HAW
HAMBURG

Bachelorarbeit

Morten Stehr

Vision-based reinforcement learning on an UR5 robot arm

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Morten Stehr

Vision-based reinforcement learning on an UR5 robot arm

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Technische Informatik*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stephan Pareigis
Zweitgutachter: Prof. Dr. Andreas Meisel

Eingereicht am: 10. Februar 2021

Morten Stehr

Thema der Arbeit

Bilddaten-basiertes Reinforcement Learning auf einem UR5-Roboterarm

Stichworte

Bilddaten, Reinforcement Learning, UR5, Roboter Arm, Belohnungs Konstruktion

Kurzzusammenfassung

Roboterarme werden für eine Vielzahl von Aufgaben eingesetzt. Im Rahmen des Testfelds Intelligente Quartiermobilität soll ein Husky-Roboter mit montiertem UR5 eine einfache Aufgabe auf dem HAW-Campus ausführen. Ziel dieser Bachelorarbeit ist es, einen Agenten mittels Reinforcement Learning für diese einfache Aufgabe zu trainieren. Hierzu ist eine Simulationsumgebung notwendig. Diese Simulationsumgebung soll die reale Welt so treffend wie möglich abbilden. Die Implementierung der Umgebung einschließlich der verwendeten Belohnung wird erläutert. Ein Agent wird mit zwei verschiedenen Ansätzen - ein- und mehrstufige Pfadplanung - trainiert und die Ergebnisse werden verglichen und ausgewertet. Dabei zeigt sich, dass der einstufige Pfadplanungsansatz sowohl bei der Annäherung an das Ziel als auch beim Erreichen des Ziels durchgängig zufriedenstellende Ergebnisse erzielt. Der mehrstufige Pfadplanungsansatz hingegen erzielt lediglich verbesserungswürdige Ergebnisse.

Morten Stehr

Title of Thesis

Vision-based reinforcement learning on an UR5 robot arm

Keywords

Vision-based, Reinforcement Learning, UR5, robot arm, Reward Engineering

Abstract

Robot arms are used in a wide variety of tasks. As part of the Test Area Intelligent Quartier Mobility project, a Husky robot with mounted UR5 should perform a simple task on the HAW campus.
This bachelor thesis aims to train an agent using reinforcement learning to perform this simple task. To fulfill this a simulation environment is necessary. This simulation environment needs to be as close to the real world as possible. The implementation of the environment including the reward engineering is explained. An agent is trained using two different approaches - single and multi-step path planning - as well as comparing and evaluating the results. These show that the single-step path planning approach achieves satisfying results getting close as well as reaching the target consistently. The multi-step path planning approach on the other hand achieves results worthy of improvement.

Contents

List of Figures

List of Accronyms

AutoSys:	Autonomous systems
HAW:	University of applied sciences Hamburg
MuJoCo:	Multi-Joint dynamics with Contact
PPO:	Proximal Policy Optimization
ROS:	Robot Operating System
TIQ:	Test Area Intelligent Quartier Mobility
TRPO:	Trust Region Policy Optimization
XML:	Extensible Markup Language

1 Introduction

In today's world robots get more important day by day. Robot arms are deployed in a wide variety of tasks including laser cutting, welding, soldering, and general assembly line work. With robot arms, there is less human risk involved when handling dangerous materials and they free up personal. That makes them indispensable in almost every factory nowadays. The advances made in research mean that autonomous robots and autonomous robotic arms are no longer just dreams of the future.

At the university of applied science in Hamburg, the AutoSys research group is dedicated, among other things, to the research project Test Area Intelligent Quartier Mobility.

The TIQ project researches the mobility of people and goods in a public space with distances less than 3 km. A part of the project includes the development of vehicles that are safe to operate in a pedestrian zone. For the project, the HAW acquired a Husky robot with a mounted UR5 robot arm. The aim is that the Husky can navigate autonomously on the campus and fulfill a simple task, like getting the mail. [Pareigis and Tiedemann, 2019]

1.1 Problem Definition

A robot arm has many fields of application. But for every task the robot arm is given, it also needs to be given instructions on how to perform this task.

The traditional way of robot arm control has been solved by hand-crafting solutions such as inverse kinematics. These hand-crafted solutions can persist of different steps such as 3D reconstruction, scene segmentation, object recognition, object pose estimation, robot pose estimation, and trajectory planning. This leads to a possible loss of information between the individual steps and thus to a possible accumulation of errors. [James and Johns, 2016]

But even without error, the traditional solution of inverse kinematics causes many difficulties because of the complexity derivation, multiple solutions, difficulty of calculation, and lack of immediacy.[Guo et al., 2019]

Another problem is that the final purpose isn't certain. The task might be changed and the robot would need to be programmed again.

In addition, the development time for the real-world training is too long. Necessary iterative improvements take a lot of time. Adding to that is that the robot is very expensive. A wrong move may damage the robot or the environment.

1.2 Objective

To control the robot, without relying on manual control, a trained controller is proposed. The controller will autonomously learn to perform specified tasks.

The proposed task is kept as simple as possible. In front of the robot is a table with a ball on top of it. The task for the robot is to move the gripper of the robot to the ball.

The control problem can be transformed into a Markov decision process. Thus, the problem can be defined as a reinforcement learning problem.[Sutton and Barto, 2018]

The reinforcement learning algorithm that is used is PPO. PPO is a comparatively new reinforcement learning algorithm that was published in 2017. PPO outperforms several other algorithms on continuous control problems.[Schulman et al., 2017]

For the training of the controller, a simulation environment is proposed. Although training could be conducted in the real world, training in a simulation environment is less time consuming and involves less risk. Learning cycles can be simulated faster than in real-time. States of the robot that would cause collisions and might cause damage in the real world don't cause any problems in the simulation.

The objective of this thesis is to build a simulation environment to train a model for a UR5 robot arm to execute the described basic task. The training will be done with different approaches. The single-step path planning approach imitates inverse kinematics and the multi-step path planning approach which takes multiple smaller steps to achieve its goal. These approaches are explained further later in the thesis. The simulation environment will be evaluated based on its ability to train an agent in general and with the different training approaches.

1.3 Structure

First, the thesis dedicates to explain the basics that are referred to in this thesis and that are necessary to understand it. After this, the implementation of the simulation environment including the reward engineering will be described. Followed by not only presenting the training process but also the final results and a discussion of it. In the penultimate chapter, the challenges that occurred while working on the thesis are presented. In closing, there is an outlook on the next steps and possible further advancements.

2 Basics

This chapter explains the fundamentals of the methods used in this thesis. In addition, the Husky robot including the UR5 robot arm is presented. The methods include the processing of the image data, the concept of reinforcement learning, and the used reinforcement learning algorithm PPO.

2.1 Husky

The Husky is an Unmanned Ground Vehicle, mounted to the Husky is an UR5 robot arm to interact with the environment. Combined with the UR5 robot arm is a RG6 flexible two-finger robot gripper with wide stroke. Sensory consists of two intel realsense D435 depth-sensing cameras, a robosense 3D-laser-scanner RD-LiDAR-16, UM7 orientation sensor, and a GPS U-Blox 7. One camera is attached slightly under the mount point of the UR5 on the Husky facing forwards and the other camera is attached at the two-finger robot gripper. [Mang and Schönherr, 2020]

For this thesis, only the UR5 robot arm is of interest. The Husky won't move or interact with the environment and can therefore be neglected. Only the image and depth data of the depth-sensing cameras will be used. The other sensory is not of importance. The Husky with mounted UR5 can be seen in 2.1.

2.1.1 UR5

The UR5 robot arm is produced by Universal Robots. It can be controlled manually by a user through an interface, or it can run autonomously.

It can lift up to 5 kg and can reach up to 850 mm. Furthermore, it has a 149 mm diameter footprint and weighs 18.5 kg. The UR5 has six joints, each joint has a rotation range from $-\pi$ to π and therefore a $720°$ rotation range. The joints can move with a

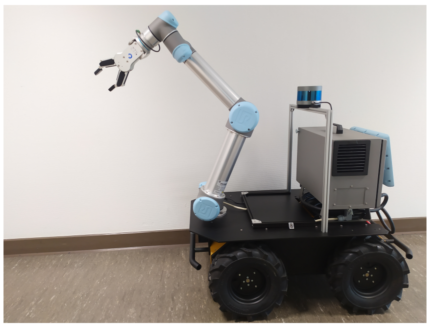
Figure 2.1: Husky robot with mounted UR5 robot arm [Mang and Schönherr, 2020]

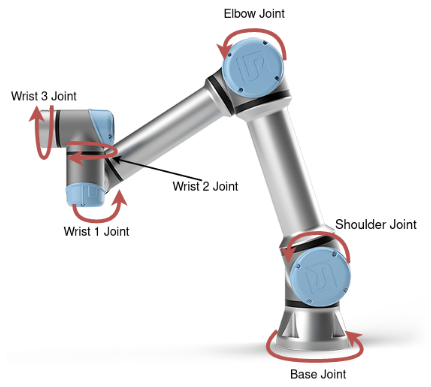

Figure 2.2: UR5 robot arm with joint specification

maximum speed of 180°/s. At the front of the arm, a RG6 flexible two-finger robot gripper with wide stroke is installed to interact with the environment. In 2.2 the UR5 can be seen with marked joints, this naming of the joints will be used throughout the thesis. [UniversalRobots, 2016]

2.2 Image Processing

Image processing describes the computer processing and extracting features of an image through algorithms. In this thesis the Sobel operator, Canny edge detection and the Hough circle transform are used.

2.2.1 Edge Detection

The goal of edge detection is to identify points in a digital image where the image brightness has discontinuities. The points where the image brightness changes sharply are typically organized in edges, therefore edge detection.

Sobel Operator

The Sobel operator is used to create an image that emphasizes edges. The operator uses two 3×3 kernels, k_1 and k_2, to calculate approximations of the derivatives through convolution. The kernel k_y is for the horizontal change and kernel k_x for the vertical change. The calculation is done for every pixel in the image. A high value indicates a big edge.

$$k_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} k_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

[Sobel, 2014]

Canny Edge Detection

After the application of the Sobel operator, the image is too noisy to be used. The Canny edge detector is a multi-stage algorithm that cleans the noise in the image and only keeps the strongest edges.

To achieve this the Canny algorithm successively applies a Gaussian filter, computes the image gradient, applies non-maximum suppression, and performs edge tracking. The Gaussian filter aims to remove noise from the image. The image gradient will identify the edges. The non-maximum suppression and edge tracking aim to remove some edges to keep only the most relevant ones. [Canny, 1986]

2.2.2 Hough Circle Transform

The Hough circle transform works on image edges that are the result of the Canny edge detection. It uses the following formula which describes a circle in the two-dimensional space.

$$r^2 = (x - a)^2 + (y - b)^2$$

where r is the radius and a and b are the coordinates of the circle center.

For each edge point, a circle with the given radius is drawn using the formula. If a circle with a specific radius is in the picture, the drawn circles will overlap in the center of that circle. This process is done for all edge points and each radius that is defined. After that, the point with the most overlapping circles is determined as the center of the circle.

The described Hough circle transform only works with a predefined radius. It's also possible to detect circles without a predefined radius, using the Hough gradient method, but this case isn't described as the input image, and therefore also the radius of the ball is known. [Kaehler and Bradski, 2008]

2.3 Reinforcement Learning

This section explains the idea of reinforcement learning. Reinforcement learning is an area of machine learning where a learning system tries to reach a given goal through interaction with the environment. In the reinforcement learning context, the learning system is called an agent. The agent receives a numeric reward for its action by the environment and tries to maximize this reward. The actions aren't given to the agent but taken randomly and are evaluated by the reward. Actions by the agent may alter the environment. In this case, the agent has to find the best actions for the new state of the environment. This behavior imitates the learning behavior of a child, that takes actions, observes the results, and adjusts its actions. [Sutton and Barto, 2018]

Besides the two terms agent and environment, four more terms characterize reinforcement learning: The policy, a reward signal, a value function, and optionally a model of the environment. [Sutton and Barto, 2018]

Policy The policy is responsible for the behavior of the agent. It maps perceived states of the environment to actions. This can be done with a lookup table or a simple function and can range to complex computations. The policy is the most important part of reinforcement learning as it alone is sufficient for the agent's behavior.

Reward Signal Reinforcement learning revolves around the reward because the reward defines the goal of a reinforcement learning problem. The reward defines which actions are good and which are bad in the given environment by assigning high rewards to good events and small rewards for bad events. After each action is taken, the agent receives a single number that represents the reward. Its only objective is to maximize the total reward.

Value Function The difference between the value function and the reward signal is, that the reward signal indicates the immediate reward, whereas the value function represents the future accumulated reward when starting in this state. The reward is easier to obtain because it is given by the environment. Whereas the value function must be calculated by the reinforcement learning algorithm. In the decision-making process, the value function is more important than the reward because it tells more

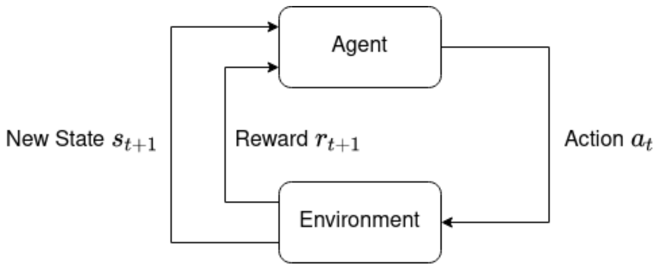

Figure 2.3: Interaction between agent and environment after [Sutton and Barto, 2018]

about the future. For example, a state can have a high reward but all the following states have low rewards, therefore the state isn't desirable. This is represented in the value function but not in the reward.

Model A model is used for planning in reinforcement learning. The model mimics the behavior of the environment. That means the model predicts the next state and the next reward, with a given start state and action. With a model, it is not necessary to error search all actions for the best one, because through the model there is the possibility of prediction and planning.
Reinforcement learning methods that use models are called model-based methods and those that don't are called model-free methods.

[Sutton and Barto, 2018]

Interaction with the Environment

The agent and the environment interact in discrete time steps $t = 1, 2, 3, 4....$ In this paper only discrete time is considered, although also continuous time is possible. At every time step the agent receives an observation of the current state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states. Based on S_t the agent chooses an action $A_t \in \mathcal{A}(S_t)$ where $\mathcal{A}(S_t)$ is the set of all possible actions in the state S_t. After this action, and one time step later, the agent receives a numerical reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and is now in the state S_{t+1}. This process can be seen in 2.3.[Sutton and Barto, 2018]

2.4 Proximal Policy Optimization

Proximal Policy Optimization, short PPO, is a policy gradient method. PPO is an on-policy algorithm, which means it has no replay buffer to store past experiences but learns directly from what the agent encounters in the environment. Once a batch of experience is used to perform a gradient update the batch is discarded and the policy moves on.

Policy Gradient

$$L^{PG}(\theta) = \hat{E}_t[\log \pi_\theta(a_t|s_t)\hat{A}_t]$$

is the loss function for a policy gradient method.

$\log \pi_\theta(a_t|s_t)$ are the log probabilities from the output of the policy network. The policy network is a neural network that takes the observed states from the environment as input and suggests actions to take as an output.

$\hat{A}_t$ is the advantage function. It is calculated by subtracting the value function from the discounted sum of rewards. The advantage function estimates the relative value of the selected action in the current state.

Discounted Sum of Rewards This is the discounted or weighted sum of all the rewards the agent got during each time step in the current episode. It's weighted by the discount factor γ because a good return now is valued more than a good return in the future. The discounted sum of rewards is calculated after the episode has been executed so all rewards are known for the calculation.

Value Function The value function is a neural network. It estimates the discounted return from this point onward. By doing so it's trying to guess the final return, starting from the given state. The neural network and therefore the value function is frequently updated during training using the experience the agent collects in the environment.

Therefore, subtracting the value function from the discounted sum of rewards answers the question of how much better the achieved reward was compared to the expected one.[Schulman et al., 2017]

The final optimization objective is the product of the log probability of the actions from the policy and the advantage function. A positive advantage function signals a better action than expected, the probabilities for these actions will be increased. If the advantage function is negative, the action was worse than expected, the probability for the action is decreased. [Schulman et al., 2017]

Trust-region

By using only policy gradient there is the risk of destructively large policy updates. The algorithm makes good actions too likely as well as excluding bad actions altogether. To prevent this the trust-region exists. The trust-region prevents the new policy from moving too far away from the old policy. [Schulman et al., 2017]

Defining $r_t(\theta)$ as the probability ratio between the new updated policy outputs and the outputs of the old policy network, $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}(a_t|s_t)}}$. Given a sequence of sampled actions and states, $r_t(\theta)$ is larger than one if the action is more likely now than in the old version of the policy. It is smaller than one if the action is now less likely. The product of $r_t(\theta)$ and the advantage function is the objective function of the Trust Region Policy Optimization, short TRPO. TRPO solves the prevention of the performance collapse with a complex second-order method. PPO solves this problem with a first-order method. [Schulman et al., 2017]

The main objective function of PPO is

$$L^{CLIP}(\theta) = \hat{E}_t[min(r_t(\theta)\hat{A}_t, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

[Schulman et al., 2017]

E_t denotes the empirical expectation over timesteps.
$r_t(\theta)\hat{A}_t$ is the already discussed objective of policy gradient methods. It pushes the policy towards actions that yield a high positive advantage over the value function.
$clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)$ is a clipped version of default policy gradient objective, applying a clipping operation between $1-\epsilon$ and $1+\epsilon$.
See 2.4 for the behavior of the objective function if the values of the advantage estimate are positive and 2.5 if they are negative. [Schulman et al., 2017]

Displayed in figure 2.4 are actions that have a better reward than expected, the advantage function is positive. The loss function flattens out when r gets too high, defined

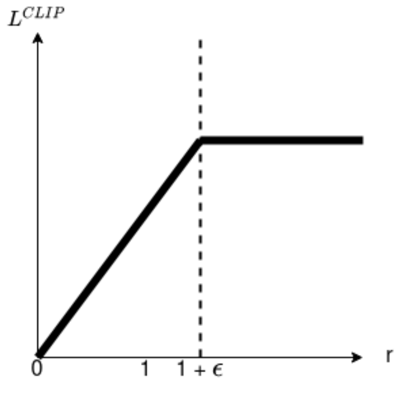

Figure 2.4: Behavior of the objective function with a positive advantage function according to [Schulman et al., 2017]

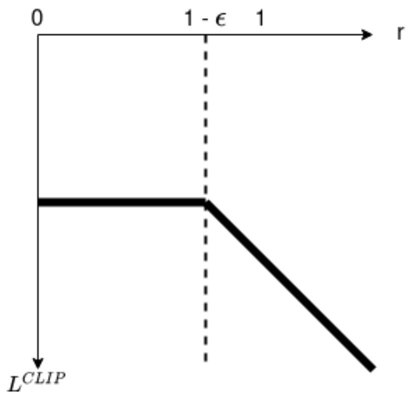

Figure 2.5: Behavior of the objective function with a negative advantage function according to [Schulman et al., 2017]

through $1 + \epsilon$. That is the case when the action is a lot more probable in the current policy than it was in the last policy. In this case, the action update shouldn't be overdone and the objective function gets clipped to limit the effect of the gradient update. [Schulman et al., 2017]

Figure 2.5 shows actions that had an estimated negative effect on the outcome. In this case, the clipping operation affects the actions that already have a small probability defined as $r < 1 - \epsilon$. They don't get their probability reduced further so that they don't end up with a probability of 0. $r > 1-\epsilon$ means the action got more probable but was also worse than expected because the advantage is negative. In this case, the last gradient step should be undone. The function is negative so the gradient will go in the other direction and make the action less likely.[Schulman et al., 2017]

Exploration

In PPO a stochastic policy is trained in an on-policy way. That means the exploration is done by sampling actions from the latest version of the policy. The randomness of action selection can be influenced by the initial conditions and the training procedure. However, throughout training the policy becomes less random as the update rule encourages already found rewards. This may cause the policy to get trapped in a local optimum. [Schulman et al., 2017]

3 Simulation Environment

In this chapter, the requirements for the simulation environment and the implementation details will be described. A great deal of attention is paid to the reward engineering.

3.1 Requirements

The purpose of the simulation environment is to represent the real world as accurately as possible so that the simulation trained model can be implemented on a real-world UR5 with as few problems as possible. To be able to achieve this, not only the robot arm needs to be accurately represented, but also the input values must be easily obtainable and the output values of the learned model must be easily usable.

Another requirement of the environment is that it needs to be able to let agents train with different approaches to reach their goal. Those approaches are single-step path planning and multi-step path planning.

Single-step Path Planning Single-step path planning imitates the process of inverse kinematics. The agent knows its state, gets the state of the environment including the goal it has to reach, and tries to reach this goal by executing one action, setting all joint angles directly into the joint target location.

Multi-step Path Planning Multi-step path planning tries to reach the goal by taking not one, but several small steps. The agent also knows its state and the state of the environment, but now takes only one small step at a time. The agent receives the information of the achieved reward and the new state of the environment and takes another step until the goal or the maximum step count is reached.

3.2 OpenAI Gym

To be able to use, test, and compare different reinforcement learning algorithms OpenAI Gym is used. OpenAI Gym acts as a platform with many existing environments for reinforcement learning. To be able to represent the exact behavior of the UR5 a custom gym environment needs to be created.
A custom OpenAI Gym environment needs to follow the gym interface and implement the following methods:

Init Function: The init function sets the parameters for the environment. Most important are the action space and the observation space.

Step Function: The step function implements the behavior of the environment when an action is taken. The function takes the action that needs to be run as the input parameter, runs the action, and returns a tuple. This tuple contains:

- **Observation:** The observation represents the state of the environment after the step was taken. Its form is defined in the observation space.

- **Reward:** The reward evaluates how good the action was performed. It can represent different things like distance to a goal or time survived. The reward can also be sparse. A sparse reward wouldn't return for example the distance to a goal, but zero if the goal isn't reached and one if the goal is reached.

- **Done Flag:** The done flag is set if either the goal of the environment is achieved or if the environment reaches a maximum number of steps without achieving the goal.

- **Additional Info:** The additional info contains additional info that might help debug or improve the learning algorithm.

Reset Function: The reset function is responsible for setting the environment into a reset state. That contains setting the agent into the initial position and if necessary set a new goal. The reset function returns an observation for the first state of the environment.

Render Function: The render function renders the behavior of the environment. Which helps to visualize the learning of the agent but slows down the computation massively.

Close Function: The close function closes all render windows when the simulation is over.

[Brockman et al., 2016]

3.3 MuJoCo

MuJoCo is the abbreviation for Multi-Joint dynamics with Contact. MuJoCo is a physics engine for robotics, biomechanics, graphics, and animations. The goal is to facilitate research and development in robotics in areas where fast and accurate simulation is required. It offers a combination of speed, modeling power, and accuracy. OpenAI gym uses MuJoCo as the physics engine for its robotics environments. That's why the choice in this environment also falls on MuJoCo as the physics engine. [MuJoCo, 2018]

3.4 Implementation

In this section, the actual implementation of the environment is described. This consists of the software architecture as well as the parameters that are essential for the reinforcement learning algorithm.

3.4.1 Architecture

The simulation environment consists of two main parts, the controller and the environment.

Controller

The controller is responsible for the creation of the environment and the UR5 robot arm. The controller builds the simulation environment from a given XML file using the Python implementation of MuJoCo, *mujoco_py*. The controller moves the joints into the joint target positions. This is done in compliance with physical rules and also signals if the joint target can't be reached because of invalid joint targets in the action or because something is blocking the way. The controller is also responsible for changing the goal at the beginning of a new episode. In the current configuration, the ball that represents the

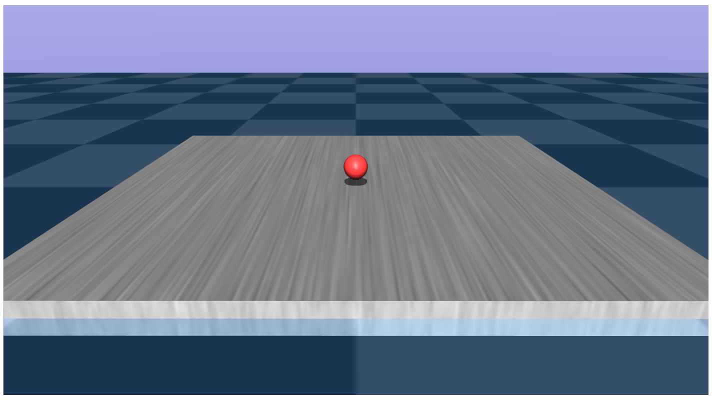
Figure 3.1: Image as seen from the lower camera

goal has no joint and therefore won't move. Additionally, the controller is responsible for rendering the simulated images that function as the observation for the environment.

Environment

The environment implements the OpenAI gym interface. It is the interface for the reinforcement learning algorithms to learn in a plug and play manner. It implements the action space illustrated in 3.4.3, gathers the information to create the observation as described in 3.4.2 and calculates the reward as in 3.4.4.

3.4.2 Observation

The observation represents the state the environment is in. The information that is given by the observation must be enough to learn for the reinforcement algorithm. In the case of this environment, it's especially important that the information gathered in the observation can be also obtained in the real world, and are not only obtainable in a simulation environment.

The obvious and also most important part that the reinforcement algorithm needs to know in this problem definition is the location of the goal. In the simulation, it would be possible to just get the location of the goal from the simulation environment and use it to train the agent. That would work very well, but this information of the goal isn't accessible in the real world. In this case, the goal must be extracted from the visual camera data.

The robot arm has two intel realsens D435 cameras that provide images with 1280x720 active stereo depth resolution. One solution to achieve the goal, which in this case is represented by the ball, would be to extract the goal using a machine learning approach. The position of the ball must be extracted using object detection that searches for the ball. However, this solution is not possible because of computation limitations further described in 6.1. As a workaround, the camera data gets preprocessed to extract the goal. Because the goal is known to be a red ball with an approximate known radius the position of the ball in the image is calculated with the Hough circle transform. By that, a two-dimensional position is obtained. By inserting that position into the depth array the corresponding depth value for this position is received. Combining these values a three-dimensional position of the goal in respect to the camera image is constructed. This is done for both cameras resulting in just two three-dimensional coordinates of the goal and six values in the observation space. By reducing the observation space from the pixel values to just the coordinates of the goal the computation time as well as the computational cost is reduced massively.

Besides the goal coordinates the observation space consists of the state of the UR5 robot arm. That state is given as the joint angles of the arm. The state of the arm would not be a piece of necessary information if the agent should only learn to reach its goal with single-step path planning. With single-step path planning, where only the final joint angles are determined, the state of the arm and the starting position are irrelevant. However multi-step path planning needs this information. With only taking little steps towards the goal the agent needs to know its current state to successfully calculate its next step.

With the state of the arm, the observation space has the size of 13. Composed of two three-dimensional goal coordinates, the six joint angles of the robot arm, and the grippers degree of opening. The latter isn't important for the objective in this thesis but is included to allow further and more complex objectives.

3.4.3 Action

The action or the action space depends on the single- or multi-step approach. In both cases, the action space consists of six values belonging to the six joints. The actions are continuous and not discrete.

For the single-step path planning approach, the action space consists of the six target values for the end position of the arm. Because this is the only action the agent takes with this approach the one action needs to take the arm to its desired position. Because of this, the actions can also range from $-\pi$ to π such as the joints. This allows the agent to set the arm to every position possible with one action. As an example: The base joint starts in position 0 and the action chosen for this joint is 1.57, or roughly $\frac{1}{2}\pi$, then the joint will move in one big step from 0 to 1.57.

$$A = \{a_0, a_1, a_2, a_3, a_4, a_5 \,|\, a \in \mathbb{Q} \wedge a \geqslant -\pi \wedge a \leqslant \pi\}$$

Multi-step path planning takes multiple small steps to get the arm to the target. The action space consists again of six values but in this case, these values don't represent the joint angles but the change of these joint angles. The actions can take continuous values from -5 to 5. These are the degrees that the joint should move. As an example: The base joint starts in position 0 and the action chosen for this joint is 2.5. The joint will move from 0 to 0.04363323129985824, which is the radiant value of 2.5 degrees.

$$A = \{a_0, a_1, a_2, a_3, a_4, a_5 \,|\, a \in \mathbb{Q} \wedge a \geqslant -5 \wedge a \leqslant 5\}$$

3.4.4 Reward

The reward signal defines the goal of the agent. The computation of the goal remains hidden for the agent, he only tries to maximize the reward per episode.

The reward persists of three components. The first component is the distance from the gripper to the goal.

$$reward_{distance} = -1 \times \sqrt{x_{distance}^2 + y_{distance}^2 + z_{distance}^2}$$

where

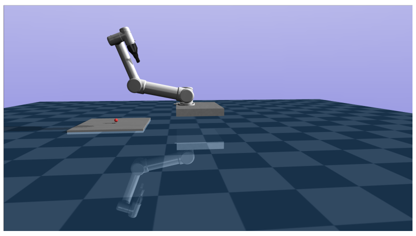
Figure 3.2: Unnatural arm position

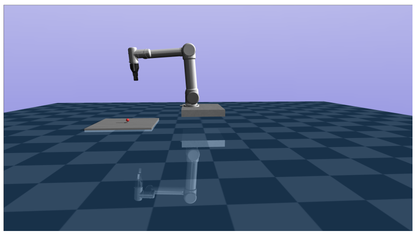
Figure 3.3: Natural arm position

$x_{distance} = x_{gripper} - x_{goal}$

$y_{distance} = y_{gripper} - y_{goal}$

$z_{distance} = z_{gripper} - z_{goal}$

Distance alone as a reward might be sufficient for the reinforcement algorithm to learn to achieve its goal, provided it does not get stuck in a local optimum described in 6.2. This is also the approach taken in the paper "A Reinforcement Learning Approach for Inverse Kinematics of Arm Robot" [Guo et al., 2019], where only the distance is used as the reward. However, to avoid getting stuck in a local optimum and to enhance general learning performance there are two additions to the reward.

The first addition to the reward for the enhanced learning performance is that the robot arm gets a reward for a "natural arm position". This can be seen in figure 3.2 and figure 3.3. The gripper of the robot arm has reached the same point in figure 3.2 like the robot arm in figure 3.3, but the agent for the robot arm in figure 3.2 will get a worse reward than the agent for the robot arm in figure 3.3. That is because in 3.2 the robot arm needs a lot of adjustments in more than one joint. Whereas in the state the robot arm in 3.3 is in, an adjustment to the shoulder joint might be enough to get close to the goal. Even if the robot arm in 3.2 would be able to reach the goal the process of grabbing and lifting the ball is much more complex than it is for the robot arm in 3.3.

$$[\, j_0 \leqslant 0 \wedge j_2 \geqslant 0 \rightarrow reward_{natural\_arm} = -1 \times (j_4 + \frac{1}{2}\pi)\,]$$

$$[\, j_0 \geqslant 0 \wedge j_2 \leqslant 0 \rightarrow reward_{natural\_arm} = -1 \times (j_4 - \frac{1}{2}\pi)\,]$$

where j_0 is the angle of the base joint, j_2 is the angle of the elbow joint and j_4 is the angle of the wrist 2 joint.

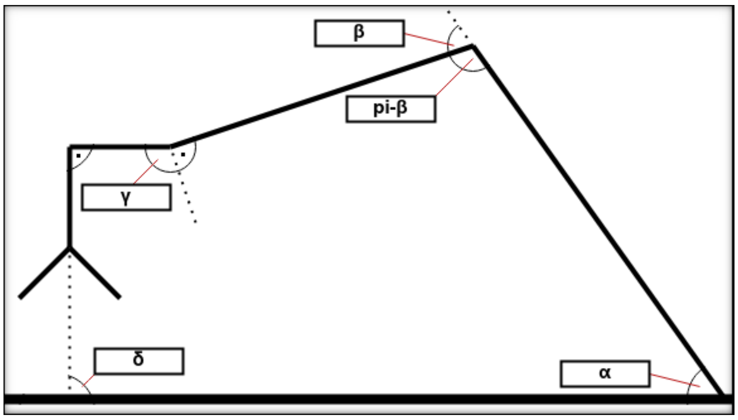

Figure 3.4: UR5 modeled as a pentagon with its interior angles

These are the cases in which the robot arm has a "natural arm position". In this case, the reward is the difference of the wrist 2 joint angle to $\frac{1}{2}\pi$, respectively $-\frac{1}{2}\pi$, because at this angle values the gripper points straight downward.

The second addition for an enhanced learning performance is that the wrist 1 joint is oriented so that the gripper is perpendicular to the ground. This is done through the internal sum of angles of a pentagon. In figure 3.4 the robot arm is shown as a pentagon. The internal sum of angles of a pentagon is 540°or 3π.

$$3\pi = \delta + \frac{1}{2}\pi + \frac{1}{2}\pi + \gamma + (\pi - \beta) + \alpha$$

For the gripper to be straight to the ground, δ must be 90°. That's why the reward is the difference to $\frac{1}{2}\pi$.

$$\delta = 2\pi - \gamma - (\pi - \beta) - \alpha$$

$$reward_{angle} = \frac{1}{2}\pi - (2\pi - \gamma - (\pi - \beta) - \alpha)$$

In figure 3.5 to 3.7 the angular circle for alpha, beta and gamma can be seen. Because of the orientation of the angle of the elbow joint the opposite angle $\pi - \beta$ must be used instead of β.

The reward is calculated with these three parts. Stephen James and Edward Johns [James and Johns, 2016] proposed a case distinction is used for the reward. The function

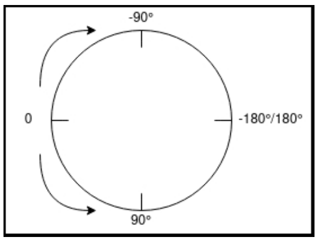

Figure 3.5: Angular circle for alpha

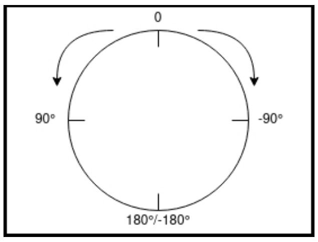

Figure 3.6: Angular circle for beta

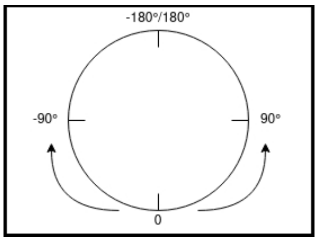

Figure 3.7: Angular circle for gamma

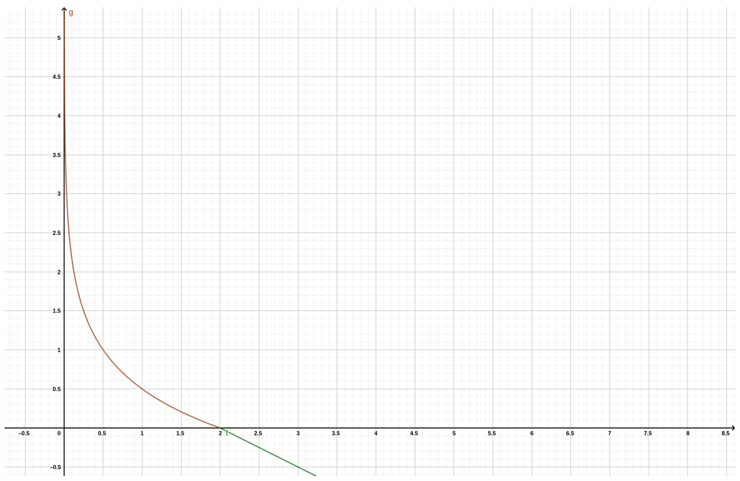

Figure 3.8: The reward function

of r_d can be seen in 3.8.

$$r_d = \begin{cases} -\frac{1}{2}reward_{distance} + 1 & \text{if } reward_{distance} \geqslant 2 \\ -\frac{1}{2}log_2(reward_{distance}) + 0.5 & \text{if } reward_{distance} < 2 \end{cases}$$

$$r_e = |reward_{natural\_arm}| + |reward_{angle}|$$

$$r = r_d - r_e$$

There are special cases for the reward. If the goal is, reached which is defined through a distance smaller than 0.01, the agent receives a reward of 10 minus r_e. This also covers the case of a distance of 0 where the log_2 is undefined. If this reward is higher than 9.85, meaning the combined negative reward of $reward_{natural\_arm}$ and $reward_{angle}$ is smaller than 0.15, the target is considered as reached and the agent receives a reward of

20. If the action is not possible to be executed the reward the agent receives is -100. If the conditions from the "natural arm position" reward aren't matched, this part of the reward is -25.

3.4.5 Episode

At the beginning of the episode, the robot arm is in a reset state. This is always the same state. This ensures that the bottom camera isn't blocked by the arm and that the top camera is aimed at the goal. From this point, the reinforcement learning algorithm will calculate an action as described in 3.4.3. Depending on the approach taken the environment behaves differently.

For single-step path planning the robot arm will execute the action in two steps. In the first step, the robot arm will try to move every joint into the given position except the shoulder joint. All the other joints move at the same time. The shoulder joint is responsible for moving the gripper up and down. By delaying this movement it's excluded that the arm gets stuck at the table. By moving the shoulder joint separately, the gripper ultimately always comes from above, which is considered good behavior. Because the agent only gets to execute one action and all joints move at the same time it would be impossible for the agent to learn to come from above. After the action is executed the reward is computed like described in 3.4.4. Also, the done flag is set, indicating that either the goal or the maximal step number of the environment is reached. As a result, the robot arm can and should only learn the direct way from the reset position to the goal. The set done flag also indicates to the reinforcement learning algorithm that the environment should be reset, which sets a new random goal and resets the robot arm to the initial position.

With multi-step path planning, there are a few differences in the procedure. The robot arm will move in multiple small steps. As a result the robot arm no longer has to move in two separate steps. All joints move at the same time. In addition, the done flag is only set after 100 actions have been taken. This is enough time for the robot arm to reach its target.

4 Training

4.1 Runtime Environment

The agent was locally trained on an Asus Zenbook with the following specifications:

Memory	7,7 GiB
Processor	Intel Core i5-6200U CPU @ 2.30GHz x 4
Graphics	Intel HD Graphics 520 (SKL GT2)
Operating System	Ubuntu 18.04.5

Unfortunately, because a MuJoCo license is required to run the environment with the MuJoCo engine, the learning process couldn't be done on a much more powerful and suited cloud solution. Under these circumstances, one learning process needed approximately 25% CPU and about 350MiB of Memory.

4.2 Algorithm

The training is done with the implementation of PPO from OpenAI. It was decided to use a professional implementation of a reinforcement learning algorithm to be able to exclude the algorithm as a source of error.

The training was done with the following parameters that are explicitly set and that differ from the standard value:

Parameter	Value single step path planning	Value multi step path planning
Gamma	0	0.99
N-Steps	128	4096
Learning rate	0.001	0.00025

For single-step path planning, gamma is set to 0. Gamma is the discount factor that ensures that actions in the future are valued less than actions now. By setting gamma to 0 only the first actions count because the other actions are set to 0. This is not necessary because the environment only allows one step before being reset but gamma is set to 0 to clarify and be safe nonetheless. N-Steps signals PPO after which amount of steps a gradient update is necessary. In this case after 128 steps. This value is set relatively low but because one step equals one complete episode in single-step path planning the gradient update takes 128 complete episodes for one update. This is especially important at the beginning of the training as the agent needs to learn fast which actions are not only bad but also impossible. The learning rate follows the same thought. The learning rate is four times as high as the standard learning rate in OpenAIs PPO implementation. The risk of the model converging too quickly to a non optimal solution is kept as small as possible by the way the reward is designed.

Multi-step path planning uses the standard value of gamma which is 0.99. Because multi-step path planning uses multiple smaller actions to achieve the goal there is the need to discount future rewards to encourage the agent to get to the goal as fast as possible. N-Steps is set to 4096 which is significantly higher than for single-step path planning. This still only takes into account 40 episodes of 100 steps when performing the gradient update but with a higher value for N-Steps, there would be fewer gradient updates during the training process. 4096 was chosen as a middle value. The learning rate is kept at the standard value of the OpenAI PPO implementation. There is a higher risk for the agent to converge to an imperfect solution because of the small steps.

5 Results

In this chapter, the results of the training in the simulation environment with both approaches are displayed and evaluated.

5.1 Single-step Path Planning

The results of the training for single-step path planning are very encouraging. In 5.1 the episode reward return is plotted. The rewards start to consistently get higher than 0 at about 20k steps. At around 50k steps the agent starts to receive a consistent reward of close to / or higher than 2. As in 3.8 can be seen this corresponds to a distance of 0.1 but with nothing else impacting the reward. Considering that r_e won't be 0, the agent achieved an even smaller distance than 0.1 and therefore a quite good result. Also worth mentioning is that there are more outliers as the learning progresses. The negative outliers are those where the agent received a reward of -100, which means that the target state could not be reached. The accumulations towards the end are a cause of the agent being very close and by trying to get closer getting stuck at the table. The outliers in the positive direction are the episodes in which the agent reached its target or at least got close enough but r_e was not small enough. This is better visualized in 5.2. It displays

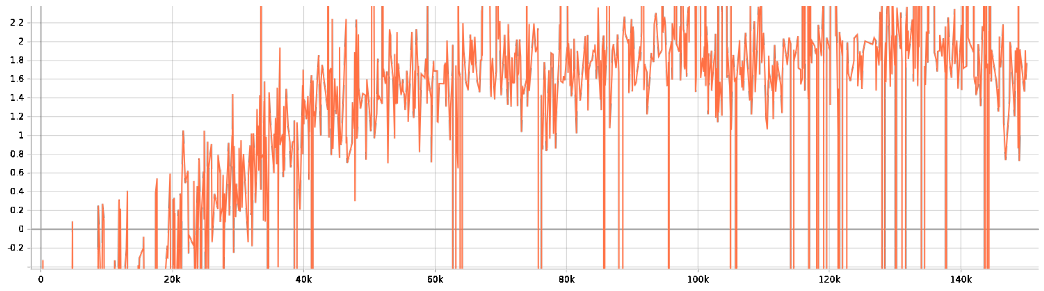

Figure 5.1: Episode reward for 150k episodes, single-step path planning

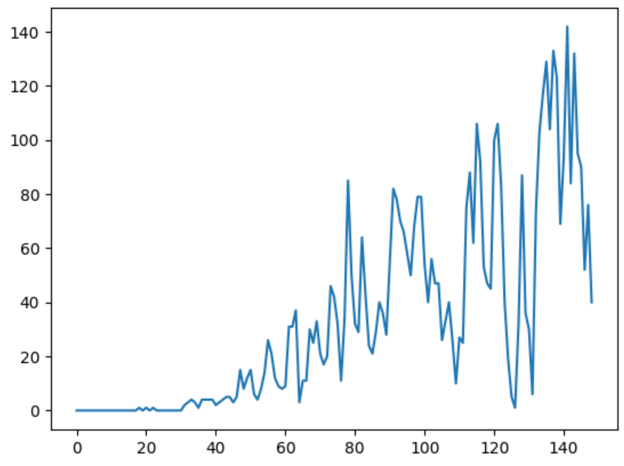

Figure 5.2: Reached goal per 1000 episodes, single-step path planning

the times the agent reached the target in 1000 episodes. This shows that even though it seems like the agent stagnates with its learning process starting at 50k episodes it gets progressively better with reaching its goal.

5.2 Multi-step Path Planning

The training of multi-step path planning wasn't as successful. Although in 5.3 can be seen that the agent indeed learns. It shows the episode rewards the agent receives. One episode now consists of 100 timesteps. This results in a different possible reward for the agent. Taking 5.1 into account the agent should be able to at least achieve results around the reward of 2. This considered one episode would achieve roughly $\sum_{n=0}^{100} 2 \times 0.99^n = 126.79$ as an accumulated reward. Looking at the achieved reward the values don't differ much but the case of the reached goal isn't considered. The agent would receive a much higher reward hence the result isn't as good as it might look. The fluctuations are probably originating in a learning rate that is too high. The agent takes a gradient update too fast. Because PPO uses no old data these updates can hinder training performance. 5.4 shows the times the agent achieved his goal per 1000 steps. The maximum comes quite early between 30k and 40k steps. This matches with the first maximum of 5.3. But the total tally is quite low especially considering that the episode doesn't necessarily end

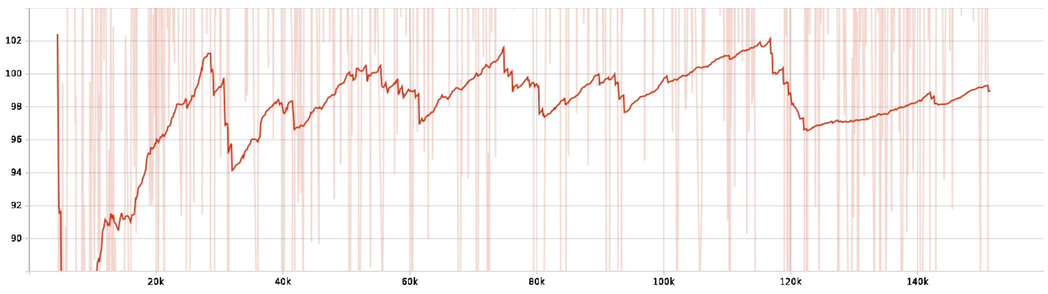

Figure 5.3: Smoothed episode reward for 150k steps, multi-step path planning

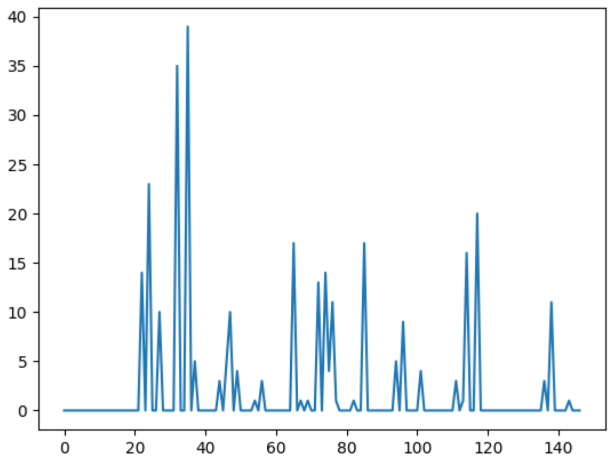

Figure 5.4: Reached goal per 1000 steps, multi-step path planning

after the agent came close, only if also r_e is small enough. The consequence is that in one episode multiple close results can be achieved. In 5.5 one example run of the training process is plotted. This run is taken from the first optimum at around 25k total steps. The distance reduces significantly from around 0.5 to 0.03 at around 90 steps.

5.6 shows the discounted rewards the agent receives. This again shows that the agent does learn. The plot visualizes the gradient update quite well.

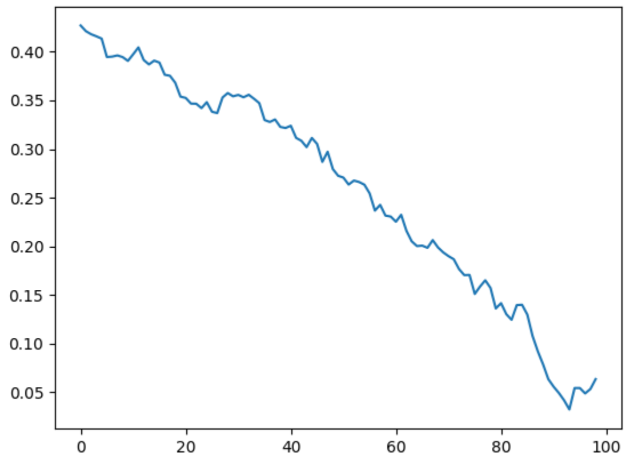

Figure 5.5: Distance of one Episode at 25k total steps, multi-step path planning

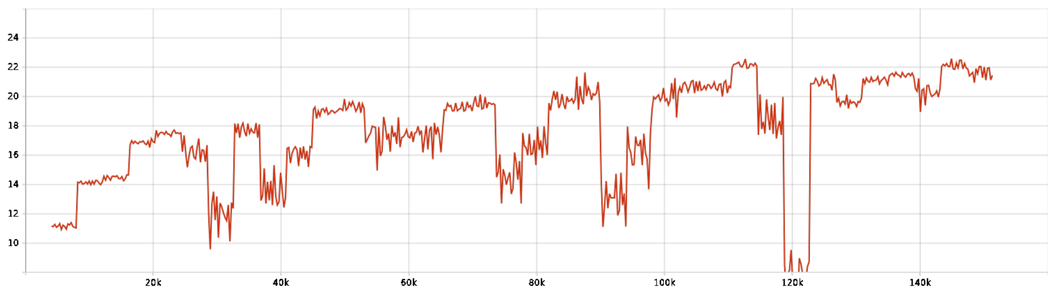

Figure 5.6: Discounted rewards with N-Steps 4096, multi-step path planning

5.3 Comparison

The question arises why single-step path planning performs notably better than multi-step path planning.
One part of the problem is that multi-step path planning had a lot fewer computations than single-step path planning. Although both simulations were executed for 150k timesteps, single-step path planning was able to simulate 150k full episodes whereas multi-step path planning was in the worst case only able to run 1500 full episodes. The worst case is that the episode never ended prematurely because the goal was reached. That means single step path planning had 100 times the full episodes to learn. Adding to that is the time of the gradient update. Single-step path planning updates the gradient after 128 timesteps. But this is after 128 full episodes whereas multi-step path planning updates the gradient after 4096 timesteps which corresponds to only roughly 41 full episodes. To get the equivalent runtime of single-step path planning, multi-step path planning would need to run for 1.92 billion timesteps with a gradient update step size of 12800. This is not realizable.

6 Challenges

6.1 Necessity of Image Preprocessing

As already slightly discussed in 3.4.2 the ideal solution for the camera data in the observation space would be to hand all pixel values, probably in downsized version, but with no other processing, to the reinforcement learning algorithm and let it learn on the pixel data. This solution will be referred to as the pixel solution in the following. The second-best solution would be to identify the target in a pre-step with object detection through machine learning. Both solutions were not possible due to their computational overhead.

Instead, the already discussed solution is implemented. The goal, known to be a red ball, is identified with the Hough circle transform. This solution works well for this particular task and in the simulation. However, this solution has some problems.

If the Hough circle transform can't detect a ball there is no goal for that episode. This could also happen with the object detection solution but not with the pixel solution. Also if the goal would have a different shape than a ball it couldn't be detected. In this case, there would be the need for another algorithm. The object detection solution would only need to be trained to detect the different shape, whereas the pixel solution will still work without changes. These problems could already appear in the simulation environment. If the process is transferred to a real-world scenario even more problems might occur. In the real world, it can't be rejected that another ball shape is found in the background of the image. In that case, the algorithm might find the wrong goal. And it can't be guaranteed that the lighting is perfect like in the simulation. That might also obstruct the detection of the goal.

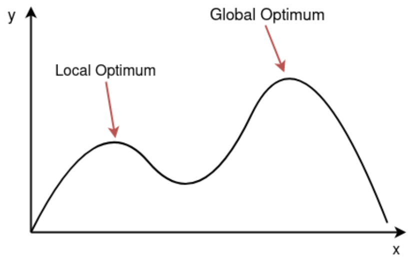

Figure 6.1: Local optimum

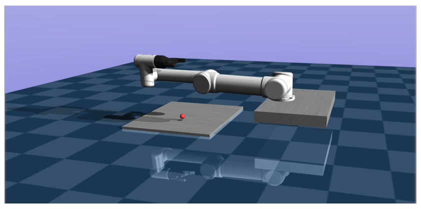

Figure 6.2: Example of the UR5 being trapped in a local optimum

6.2 Local Optima

The environment seems quite susceptible to get stuck in a local optimum. The concept of a local optimum can be seen in 6.1. The value is on a local high and all possible solutions in the direct surroundings are worse. PPO is already susceptible to get stuck in a local optimum because it explores by sampling actions from the last version of the policy. In the simulation environment, one problem is that especially at the beginning of the learning process most of the actions the agent takes are invalid actions, resulting in a return of -100. That makes the few valid solutions very good, even though they might not be. In 6.2 an example can be seen for such a state. It's apparent that the state of the robot arm isn't good. But it's not easy to escape from that state. If the arm moves up or to the sites, the reward gets worse because the distance to the goal got bigger. If the arm tries to move down it's an invalid action and the reward is -100. The additional parts of the reward aim to prevent that the agent gets stuck in a local optimum. By enforcing the arm to be in a natural position and rewarding the agent for the gripper being straight, it's much more unlikely for the agent to be trapped in a local optimum. Now there is the actual possibility for an improved reward when the arm moves. There are other exploration possibilities to avoid getting trapped in a local optimum. Epsilon greedy for example solves the problem by periodical taking a completely random action. This solution would also have problems in this environment as the complete random action would probably be an invalid one, as a consequence of the very big continuous action space with only a small part being valid actions.

6.3 Joint Angle Inaccuracy

A continuous action space has the consequence that the actions, the agent selects, can be very precise. In the simulation and the reality, it might not be possible to set the joint angles to the exact angles given by the action. But it could be that the rough angle is a valid solution and therefore the return value of the environment should not be one of the impossible angles, but the environment allows a slight deviation in setting the angles.

The tolerance in the environment is set to 0.01. This value was determined by testing. It's the smallest value where the environment still gets the joint angles close to the joint angle target without failing to do so. The consequence can be seen in an example. The action given by the agent is 1.35742, in this case, every angle in the range [1.34742, 1.36742] is accepted as a successful movement of the joint angles to the joint angle target. This can happen for all of the six joints on the robot arm.

The result is that the environment can get two quite different states from the same action. This isn't beneficial for the learning process as this generates certain randomness.

Another problem with the joint angle inaccuracy arises with multi-step path planning. If the allowed step size is too small the robot arm might not move at all because the joints already are in the accepted range. $1°$ is 0.02 as radian. So every change that is bigger than $0.5°$ will move the joint but the impact is still bigger than with single-step path planning. With the action space of $-5°$ to $5°$, there is always the possibility of a deviation of $0.5°$ which is at least 10% of the action that's taken.

6.4 Reward Calculation in the Real World

To train the agent in the real world or even adjust an already trained agent it is necessary to get the information that is available in the simulation. This is easily possible for the observation space but not as easy for the reward. The additional reward parts that are combined in r_e are easily converted to the real world because their computation is exactly as in the simulation and only the joint angles are necessary for the calculation and they are easy to access.

The problem is to calculate the distance. In the real world, there are no coordinates for the goal to just calculate the reward. A simple solution would be to just measure the distance after each action. This is a complex task but the possibility exists.

The second solution works with MoveIt! which is the motion planning framework in ROS, the robot operating system. It is possible to get the coordinates of the robot arm gripper through MoveIt! as well as it is possible to calibrate coordinates for the goal which are relative to the camera. This way it is possible to access both goal and gripper coordinates and calculate the distance like in the simulation. If configured correctly this is a good way of accessing the data in the real world.

7 Discussion

7.1 Need for Research

To evaluate further the trained model and therefore the environment the model would need to be implemented on the real-world UR5 robot arm. To test the agents ability to reach the goal in the real world and also inspect the agents capability to relearn if the model is a little off. This can be done with the training results that were achieved in this thesis to inspect if the robot arm behaves as in the simulation even if this behavior might not be ideal.

Despite encouraging results there are further things to improve. Single-step path planning showed that it is possible to train an agent in the environment and multi-step path planning already showed promising results. Nevertheless, it needs better training including hyperparameter tuning. With better hyperparameters and a longer training time, it will be possible to train an agent using multi-step path planning. Training the agent for a longer time is easier to realize with better hardware. If the training is done with better hardware there is no need for image preprocessing. The reinforcement learning algorithm would only use the pixel data of the cameras. This should improve the results and needs to be tested.

A next step to advance the simulation environment would be to complicate the goal the agent tries to achieve. Currently, the goal is as easy as possible. The next goal could be to reach and grab or move the ball. For this purpose, the reward function would need to be adjusted. The reward function doesn't have to change, a bonus reward for a successful grab would be sufficient.

Another thing for the future is to test and train with the environment using a different reinforcement learning algorithm. In this thesis, only PPO was used but different algorithms can also achieve good results for example Deep Deterministic Policy Gradient. There is a possibility that another algorithm is better in interaction with this environment.

7.2 Conclusion

The goal of this thesis was to implement a simulation environment for an UR5 robot arm to learn to fulfill tasks through reinforcement learning.

As the results show this can be considered as successful. Single-step path planning was able to reach its target in a short time of learning, although not consistent. Multi-step path planning still needs work to achieve these results. The ability of both approaches to be trained proves that the environment is capable and suitable to train a reinforcement learning agent. Enhancing the training behavior is a reinforcement learning problem where the results can be improved by changing hyperparameters as well as training the agent for a longer time.

There are different layers to the improvable results. Some of the challenges can't be fixed easily like the joint angle inaccuracy. That is a limitation given by the physics environment.

Another part of the objective for this work was to design the simulation environment to be as easy as possible to transfer to the real world. This was managed well for the input values, which can be taken from the cameras of the UR5. Also, the output values can be used directly to move the real-world UR5 robot arm. This objective wasn't achieved for the reward. The reward uses information that is complicated to observe in the real world. It is possible but inconvenient.

In conclusion, the goal of the thesis was reached. A simulation environment that is able to train a reinforcement learning agent to perform a simple task. This environment was supposed to be as close to the real world as possible, this has been achieved with the exception of the reward. Nevertheless, there is still room for improvement for the environment and the trained model.

Bibliography

[Brockman et al., 2016] Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., and Zaremba, W. (2016). Openai gym.

[Canny, 1986] Canny, J. (1986). A computational approach to edge detection. *IEEE TRANSACTIONS ON PATTERN ANALYSIS AND MACHINE INTELLIGENCE*.

[Guo et al., 2019] Guo, Z., Huang, J., Ren, W., and Wang, C. (2019). A reinforcement learning approach for inverse kinematics of arm robot. pages 95–99.

[James and Johns, 2016] James, S. and Johns, E. (2016). 3d simulation for robot arm control with deep q-learning.

[Kaehler and Bradski, 2008] Kaehler, A. and Bradski, G. (2008). *Learning OpenCV*.

[Mang and Schönherr, 2020] Mang, M. and Schönherr, N. (2020). Husky. URL https://autosys.informatik.haw-hamburg.de/platforms/2020husky/ [Last accessed on 2021-02-10].

[MuJoCo, 2018] MuJoCo (2018). *MuJoCo advanced physics simulation*. URL http://www.mujoco.org/ [Last accessed on 2021-02-10].

[Pareigis and Tiedemann, 2019] Pareigis, S. and Tiedemann, T. (2019). Test area intelligent quartier mobility (tiq). URL https://autosys.informatik.haw-hamburg.de/project/smartmobility/ [Last accessed on 2021-02-10].

[Schulman et al., 2017] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., and Klimov, O. (2017). Proximal policy optimization algorithms.

[Sobel, 2014] Sobel, I. (2014). An isotropic 3x3 image gradient operator. *Presentation at Stanford A.I. Project 1968*.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement Learning: An Introduction*. The MIT Press. URL http://incompleteideas.net/book/the-book-2nd.html.

[UniversalRobots, 2016] UniversalRobots (2016). *UR5 Technical specifications*. URL https://www.universal-robots.com/media/50588/ur5_en.pdf [Last accessed on 2021-02-10].

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ______________________

Vorname: ______________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Bilddaten-basiertes Reinforcement Learning auf einem UR5-Roboterarm

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

		[illegible]
Ort	Datum	Unterschrift im Original