

SMARTBOT

ROBOTER MIT REINFORCEMENT LEARNING

MSE-VERTIEFUNGS PROJEKT 2

Autor: Joël Lutz joel.lutz@ntb.ch

Advisor: Prof. Dr. Klaus Frick klaus.frick@ntb.ch

Abgabedatum: 18. August 2018

Abstract

Robotics has ever since been an active research area for artificial intelligence. Smart drones, high-capable walking robots or autonomous cars are but some examples. A promising and fast-growing technology for AI is Reinforcement Learning (RL). This technique can naturally be used for robotics applications. The goal of this project is to gain insight to this area by developing a demonstrator for a typical industrial robotics task, which is solved with Reinforcement Learning. The task for the robot to be learned is picking up a box, based on a depth image from a Kinect sensor mounted above the box. For the robot arm, the PhantomX Pincher is used.

The training of the RL agent is done with a robot simulator. Therefore, the setup for this project consists of three main parts: OpenAI Gym, ROS and Gazebo. OpenAI Gym is a toolkit for developing RL algorithms in Python. ROS is a modular open source framework for robotics which is widely used. It serves as a middleware between the different components. Gazebo is a robot simulator with sophisticated integration of ROS. In this project, this complete tool-chain has been implemented for the Pincher arm and some RL algorithms have been tested.

In this (german) documentation, an introduction into Reinforcement Learning is given first, along with the algorithms which are later tested with the Pincher arm. Subsequently, some clarifications and decisions are discussed, e.g. which simulator or which robot shall be used for this project. Next, it is shown how the simulation of the robot and its environment is done. This will be followed by the discussion of the OpenAI Gym environment, which allows the RL agent to interact with the robot in the simulation during the training. In the next chapter, the whole setup in hardware is explained and how it can be connected to the RL agent. Finally, the tested RL algorithms and their performance are described.

Zusammenfassung

Robotik ist schon lange ein aktiver Forschungsbereich für Künstliche Intelligenz (KI). Autonom fahrende Fahrzeuge, smarte Drohnen oder leistungsfähige, gehende Roboter sind nur wenige Beispiele davon. Eine vielversprechende und schnell wachsende Technologie der KI ist das Reinforcement Learning (RL). Diese Technik kann sehr gut in der Robotik angewendet werden. Das Ziel dieses Projektes ist es, anhand eines Demonstrators mit einem Roboter einen Einblick in das Gebiet des Reinforcement Learnings zu erhalten. Der Roboter soll eine typische industrielle Aufgabe lösen, welche er mittels RL selbst erlernt hat. Die Aufgabe für den Roboter ist das Greifen eines Quaders anhand eines Tiefenbildes einer Kinect-Kamera, welche oberhalb des Quaders montiert ist. Für den Roboter-Arm wird der PhantomX Pincher verwendet.

Das Training des RL-Agenten wird mit einem Roboter-Simulator durchgeführt. Deshalb besteht das Setup in diesem Projekt aus drei Haupt-Komponenten: OpenAI Gym, ROS und Gazebo. OpenAI Gym ist ein Toolkit für RL-Algorithmen in Python. ROS ist ein modulares Open Source Framework für Robotik, welches weit verbreitet ist. Es dient als Middleware zwischen den verschiedenen Komponenten. Gazebo ist ein Roboter-Simulator mit guter Integration von ROS. In diesem Projekt wurde diese ganze Toolchain für den Pincher-Arm implementiert und ein paar RL-Algorithmen damit getestet.

In dieser Dokumentation wird zuerst ein Einstieg in das Reinforcement Learning gegeben, inklusive jener Algorithmen, welche später mit dem Pincher-Arm getestet wurden. Anschliessend sind einige Abklärungen und Entscheide dokumentiert, wie beispielsweise welcher Simulator oder welcher Roboter für dieses Projekt verwendet werden soll. Danach wird aufgezeigt, wie die Simulation des Roboters samt seiner Umgebung implementiert ist. Im darauffolgenden Kapitel wird das OpenAI Gym environment beschrieben, welches dem RL-Agenten erlaubt, mit dem Roboter in der Simulation während des Trainings zu interagieren. Danach wird der ganze Versuchsaufbau in Hardware erklärt und wie dieser mit dem RL-Agenten verbunden werden kann. Schliesslich werden die getesteten RL-Algorithmen und ihre Performance beschrieben.

Danksagung

Ich möchte mich hiermit bei allen Personen bedanken, welche mir für dieses Vertiefungsprojekt mit Rat und Tat zur Seite standen und damit einen wertvollen Beitrag geleistet haben. Ein besonderer Dank geht an:

- **Prof. Dr. Klaus Frick** für seine gute Unterstützung während des ganzen Projektes
- **B.Sc. Manuel Ilg** für seine Hilfe mit Gazebo und dem Hinweis mit dem Pincher-Arm
- **Prof. Einar Nielsen** für seine Unterstützung in Sachen Robotik
- **M.Sc. Simon Härdi** für die Unterstützung mit dem Cluster und L^AT_EX
- **Prof. Dr. Norbert Frei** für die Zurverfügungstellung des Pincher Roboter-Armes
- **dem PWO-Institut** für die Zurverfügungstellung der Kinect Kamera
- **Fabian Saccilotto** für die Hilfestellungen bei der Inbetriebnahme des Pincher-Roboter-Armes
- **Jorge Santos Simòn** für die Hilfestellung bei der Gazebo-Simulation des Pinchers
- Allen weiteren Beteiligten, sei es für die geleistete Unterstützung, Korrekturlesen oder sonstige Hilfestellungen

Inhaltsverzeichnis

1 Einleitung	1
1.1 Ziel der Arbeit	2
2 Reinforcement Learning	3
2.1 Vergleiche zu anderen lernfähigen Systemen	3
2.2 Konzept	4
2.3 Value Function & Policy	5
2.4 Exploration & Exploitation	6
2.5 Q-Learning	7
2.6 Kontinuierliche State-Spaces	9
2.7 Kontinuierliche Action-Spaces	12
3 Abklärungen & Entscheide	16
3.1 RL mit Robotik	16
3.2 Auswahl des Simulationsprogramms	16
3.3 Auswahl des RL-Toolkits	18
3.4 Auswahl des Roboters	18
3.5 Auswahl des Tasks für den Roboter	19
3.6 Auswahl des Sensors	23
3.7 Zusammenfassung & Überblick	23
4 Simulation	25
4.1 Simulation Roboter-Arm	25
4.2 Simulation Kinect-Kamera	34
5 OpenAI Gym environment	35
5.1 Aufbau der Simulation	35
5.2 Zufällige Platzierung des Quaders	36
5.3 step-Methode	37
5.4 Reward-Berechnung	38
5.5 Einlesen der Tiefenbilder	43
5.6 Definition des State- & Action-Spaces	45
6 Versuchsaufbau in Hardware	46
6.1 Inbetriebnahme realer Roboter	47
6.2 Inbetriebnahme reale Kinect-Kamera	48
7 RL-Algorithmus	53
7.1 Grundsätzlicher Trainings-Ablauf	53
7.2 Reduzierung der Action-Space Dimensionen	54
7.3 DQN-Implementierung	54
7.4 DDPG-Implementierung	55
8 Fazit	62
9 Ausblick	63
Abbildungsverzeichnis	65
Code Listings	66
Tabellenverzeichnis	67

Literatur	68
Eigenständigkeitserklärung	71

1 Einleitung

Eine der grössten Errungenschaften der Evolution ist das Gehirn, welches sich vor allem durch seine Fähigkeit auszeichnet, zu lernen. Der technische und wissenschaftliche Fortschritt der letzten Jahre hat den Menschen dazu veranlasst, diese Fähigkeit auch Computern zu verleihen – wenn auch noch nicht (immer) mit einer solchen Leistung und Universalität wie in der Natur. Dieses Gebiet wird Künstliche Intelligenz (KI) oder Artificial Intelligence (AI) genannt, welches zurzeit beachtliche Fortschritte verzeichnet. Ein Gebiet der KI ist das Reinforcement Learning, welches dem prinzipiellen Lernen von Menschen und anderen Lebewesen relativ ähnlich ist und in diesem Projekt behandelt und angewendet wird.

Im Zuge der fortschreitenden Automatisierung verschiedenster Abläufe werden häufig Roboter eingesetzt, welche (vorprogrammiert) diverse Aufgaben erledigen können, wie beispielsweise das Handling von Teilen in einer Produktionsstrasse. Die Motivation dieses Projektes ist es, die Welt der KI mit der der Automation und Robotik zu verbinden und dabei kennenzulernen. Beispiele für die Verwendung von KI mit Robotern bieten beispielsweise *Boston Dynamics*, welche verschiedenste autonome Roboter entwickeln. Einer davon ist SPOT, welcher in Abbildung 1.1 dargestellt ist.

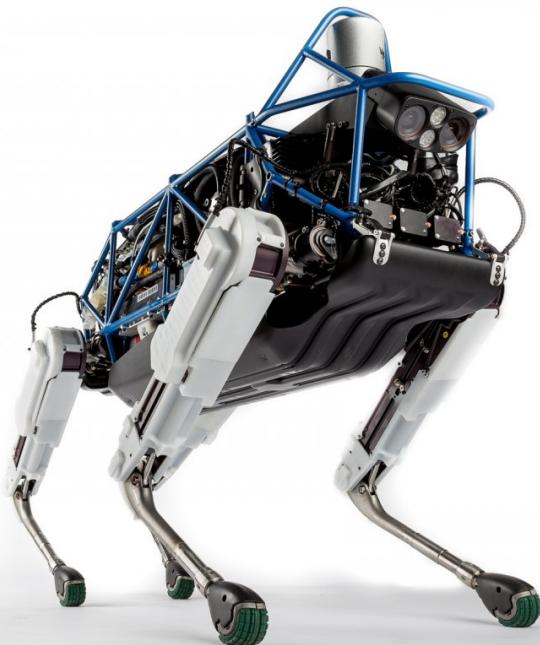


Abbildung 1.1: Der autonome Roboter SPOT von Boston Dynamics [1]

Ein Roboter, welcher nicht mehr auf eine bestimmte Aufgabe vorprogrammiert werden muss, sondern über die Fähigkeit zu lernen verfügt, eröffnet eine Vielzahl an neuen Einsatzmöglichkeiten. So könnte beispielsweise ein selbstlernender Roboter direkt mit Menschen zusammenarbeiten und auf ihre Verhaltensweisen entsprechend reagieren. Dies könnte die Integration von Robotern in unseren Alltag erleichtern. Weiter könnte ein Erkundungs-Roboter mit KI autonom bisher unbekanntes Gebiet erforschen, analysieren und sich der Umgebung entsprechend anpassen und verhalten. Es ist bereits

heute schon möglich, dass ein selbstlernender Roboter-Arm Objekte greifen und platzieren kann, welche nicht immer genau gleich aussehen, wie beispielsweise verschiedene Tassen und Flaschen (siehe [2]).

1.1 Ziel der Arbeit

Das Hauptziel dieses Vertiefungsprojekts ist es, anhand eines Beispiels mit einem Roboter einen Einblick in das Gebiet des Reinforcement Learnings zu erhalten. Dazu soll als Demonstration ein Roboter sich mittels dieser Technik eine Aufgabe selbst beibringen (daher auch der Name des Projektes, *SmartBot*).

2 Reinforcement Learning

Ein in den letzten Jahren sehr schnell gewachsenes und immer noch wachsendes Gebiet der Künstlichen Intelligenz ist das Reinforcement Learning (*bestärkendes Lernen*, RL). Obwohl das Konzept schon seit ca. Ende der 1980er Jahre bekannt ist, erfuhr RL in den letzten Jahren enormen Aufschwung, wie auch viele andere Bereiche der KI wie beispielsweise die neuronalen Netze. Auch in den Medien hat RL bereits grosses Aufsehen erregt, beispielsweise als ein mit RL trainierter Computer erstmals den amtierenden Weltmeister im Spiel Go¹ schlug [4].

In diesem Projektbericht soll nur das Grundkonzept von RL und ein paar Algorithmen vermittelt werden. Dabei werden bewusst die englischen Begriffe nicht ins Deutsche übersetzt, um mit der Literatur möglichst im Einklang zu bleiben. Gute und teils auch für die folgenden Abschnitte verwendete Quellen in Bezug auf Reinforcement Learning sind u.a. folgende:

- RL-Buch [5]
- RL-Kurs von David Silver (inkl. Videos) [6]
- Blog über RL und 2 Algorithmen [7]
- Einführung in ein paar RL-Algorithmen [8]
- Code-Beispiele von verschiedenen RL-Algorithmen [9]
- Repository mit diverser Literatur und Beispielen [10]
- RL in der Robotik [11]

2.1 Vergleiche zu anderen lernfähigen Systemen

Das RL ist vom Prinzip her dem menschlichen Lernen sehr ähnlich, was es (zumindest für uns Menschen) zu einer sehr vielversprechenden Technik macht. Im RL wird direkt anhand von Interaktionen mit der Umgebung gelernt, welche Tätigkeiten/Entscheide gut und welche eher schlecht waren.

Dies unterscheidet sich vom sog. *supervised learning*, wo ein Task durch viele vorgängig (von Menschen) etikettierte Daten (aka Daten mit Labels) eingelernt wird. Wenn beispielsweise ein Convolutional Neural Net (CNN) automatisch Katzen und Hunde anhand von Bildern unterscheiden soll, muss es mit einer (relativ grossen) Anzahl von Bildern von Katzen & Hunden trainiert (gefittet) werden, von welchen die Wahrheit bekannt ist (d.h. ob sich auf dem Bild nun ein Hund oder eine Katze befindet).

Beim *unsupervised learning* haben die Trainings-Daten keine Labels, da der Task ein anderer ist. Der Algorithmus muss nämlich hierbei selbst Muster in den Daten finden. Ein Beispiel dazu wäre eine Gruppierung von Spam-E-mails in verschiedenste Untergruppen, welche aber nicht vorgegeben sind (da dies nicht unbedingt im Voraus bekannt ist).

Eines habe jedoch alle lernfähigen Systeme – sei es der Mensch, ein Tier, ein RL-Algorithmus oder ein (un)supervised Machine Learning Algorithmus – gemeinsam: Sie alle müssen „trainiert“ werden. Dieses Training kann je nach System anders aussehen: so wird z.B. der Hund mit Futter auf seine Aufgaben trainiert oder ein Machine Learning

¹Go ist ein Brettspiel, welches ca. 10^{170} mögliche States hat (zum Vergleich: Schach hat „nur“ 10^{50} States) [3]

Algorithmus „fittet“ seine Parameter durch ein Optimierungsverfahren an die Trainings-Daten.

2.2 Konzept

Im Reinforcement Learning gibt es einen Agenten, welcher in jedem Zeitschritt t mithilfe von Actions mit seiner Umgebung (*environment*) interagieren kann. Der Agent hat einen internen State s_t , welcher seine Repräsentation der Umgebung zu dem momentanen Zeitschritt t ist. So kann beispielsweise der Agent ein fahrender Roboter sein, welcher als Actions „fahre geradeaus“, „fahre links“ und „fahre rechts“ hat. Der State des Roboters könnten alle Inputs seiner Sensoren sein, wie beispielsweise die momentanen Messwerte der Distanzsensoren. Wenn nun der Agent eine Action ausführt, reagiert die Umgebung darauf, was den Agenten in einen neuen State s_{t+1} versetzt, dem State des nächsten Zeitschritts. In dem neuen State kann der Agent nun wieder entscheiden, welche Action er als nächstes ausführen möchte. Wenn im Roboter-Beispiel der Roboter beispielsweise die Action „fahre geradeaus“ wählt, würde er eine gewisse Zeit diese Aktion ausführen, danach seine Distanzsensoren abfragen und so sich in einem neuen State befinden.

Die Umgebung gibt neben dem neuen State aber immer auch noch einen Reward (Belohnung) R zurück. Dies ist eine einzelne, skalare & reelle Zahl, welche dem Agenten mitteilt, wie „gut“ der neue State für ihn ist. Dabei sind grössere Zahlen „besser“ als kleinere. Wenn der Roboter beispielsweise eine Ziellinie überqueren soll, könnte der Reward 1 sein, wenn er die Ziellinie überquert hat, und 0 in allen anderen Fällen. Der Reward könnte aber auch als $1/d$ definiert sein, wobei d die Distanz zur Ziellinie ist. Im Allgemeinen wird mit dem Reward dem Agenten implizit gesagt, was er zu tun/zu erreichen hat. Eine schematische Darstellung des RL-Konzeptes ist in Abbildung 2.1 dargestellt.

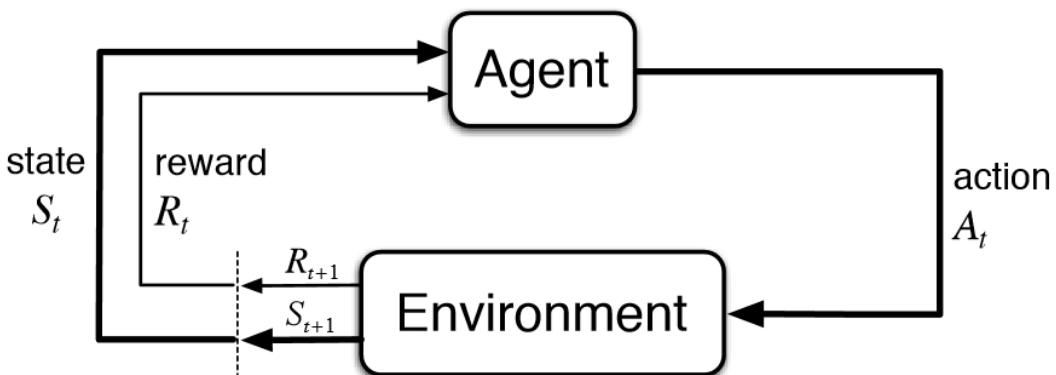


Abbildung 2.1: Konzept des Reinforcement Learnings [6]

Es ist aber wichtig zu beachten, dass eine Action in einem State nicht immer denselben Nachfolge-State und Reward mit sich bringt. Dies kann verschiedene Gründe haben, beispielsweise durch äussere, nicht im State abgebildete Umstände wie einen Windstoss, welcher den Roboter etwas verschiebt. Vielfach lässt sich das Training eines RL-Agenten auch in Episoden unterteilen, beispielsweise wenn der Agent sein Ziel erreicht hat und sich in einem sogenannten *Terminal State* (Schluss-State) befindet.

2.2.1 Vorgehensweise & Ziel des Agenten

Der Agent versucht nun während seiner Interaktion mit der Umgebung, seinen gesamten kumulativen Reward zu maximieren. Dabei führt er stets eine Action aus, welche ihn durch die Umgebung in einen neuen State transferiert und er einen neuen Reward erhält. Der Agent bewertet nun seine vergangenen Actions anhand des erhaltenen Rewards. Dies führt schliesslich dazu, dass er lernt, welche Actions in welchen States er wählen muss, damit er möglichst viel Reward erhält.

2.3 Value Function & Policy

Doch wie wählt der Agent seine Actions aus? Dies geschieht durch die sogenannte Policy π (Regel, Taktik). Die Policy weist den States die auszuführende Action zu. Sie mappt also von States auf Actions. Dies kann deterministisch oder stochastisch geschehen, also entweder wird für den gleichen State auch immer (deterministisch) die gleiche Action ausgewählt, oder für einen State werden mit gewissen Wahrscheinlichkeiten aus verschiedenen Actions ausgewählt. Im Laufe des Trainings kann der Agent seine Policy auch ändern (verbessern).

deterministische Policy: $a = \pi(s)$

stochastische Policy: $\pi(a|s) = \mathbb{P}[a_t = a | s_t = s]$

Wie kann der Agent nun aber die Actions oder die States bewerten? Dazu dienen die sogenannten *Value Functions*. Hierbei gibt es zwei Sichtweisen für den Agenten. Die *State Value Function* teilt dem Agenten mit, wie gut ein State für ihn ist, d.h. wie viel zukünftigen Reward er sich von diesem State versprechen kann. Die *Action Value Function* sagt dem Agenten, wie gut eine spezifische Action in einem spezifischen State ist. Beides sind also relativ ähnliche Konzepte. Beide Value Functions schätzen den gesamten zukünftigen Reward. Dieser wird definiert als Erwartungswert aller zukünftig erhaltenen Rewards, würde man von dem State s aus starten und danach immer der Policy π folgen. Die ferner in der Zukunft liegenden Rewards werden oftmals mit einem sog. *discount factor* $0 \leq \gamma \leq 1$ multipliziert. Durch diesen werden die Rewards, welche lange nach dem momentanen State s erhalten wurden, schwächer gewichtet als eher naheliegende Rewards.

Die State Value Function ist definiert als

$$v_\pi(s) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t = s] \quad (2.1)$$

Die Action Value Function nimmt dabei eine ähnliche Form an:

$$q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots | s_t = s, a_t = a] \quad (2.2)$$

Nach jedem Zeitschritt aktualisiert der Agent seine Value Function (welche von den beiden er auch immer verwendet). Somit werden mit der Zeit (Erfahrung) die Schätzungen der Value Function immer genauer. Wie die Value Function während der Interaktion mit der Umgebung aktualisiert werden kann, wird später anhand eines Beispiels in Abschnitt 2.5 beschrieben.

2.3.1 Value Based vs. Policy Based vs. Actor-Critic Methoden

In *Value Based* Methoden verfügt der Agent nur über eine Value Function (State oder Action Value), besitzt jedoch keine Policy Function, also kein explizites Mapping von States auf Actions. Die Policy ist somit nur implizit vorhanden. Doch wie kann der Agent dann eine Action anhand seines momentanen States auswählen? Dies geht ganz einfach, nämlich indem der Agent seine Policy aus der Value Function ableitet. Beispielsweise könnte der Agent in jedem Zeitschritt einfach stets die Action nehmen, welche den höchsten Action Value aufweist, da der Agent ja einen möglichst grossen kumulativen Reward erhalten möchte.

In *Policy Based* Methoden hingegen hat der Agent keine explizite Value Function, dafür ist seine Policy direkt als Funktion vorhanden. Der Agent konsultiert seine Policy bei jedem Zeitschritt (Input = State, Output = Action) und aktualisiert sie nach dem erhaltenen Reward und neuen State entsprechend, damit die Policy immer besser wird und so dem Agenten mehr Reward einbringt.

Es gibt aber auch Algorithmen, welche beide Ansätze kombinieren. Dies sind die sogenannten Actor-Critic Methoden, welche sowohl eine explizite Value Function als auch eine explizite Policy Function haben. Mehr dazu in Unterabschnitt 2.7.1.

2.4 Exploration & Exploitation

Ein wichtiges Konzept im RL ist der Trade-Off zwischen *Exploration* und *Exploitation*. Exploration (Erkundung) bedeutet, dass der Agent möglichst viele verschiedene States besucht und viele verschiedene Actions ausprobiert, auch wenn diese vielleicht nicht immer die besten States bzw. Actions sind (d.h. die Value Functions für diesen State/Action eher niedrig ist). Durch Exploration findet der Agent mehr Informationen über seine Umgebung heraus, jedoch auf Kosten von bekannten Rewards. Dadurch kann er aber auch States finden, welche zwar anfänglich einen schlechten Reward versprechen, sich dann aber als viel besser als alle bisher bekannten States herausstellen.

Exploitation (Ausbeutung, Ausnutzung) hingegen ist das Gegenteil von Exploration – hier wählt der Agent stets die Action aus, welche den höchste momentane Value Function Wert hat, also den höchsten zukünftigen erwarteten Reward verspricht. Dies ist eine *greedy* Policy, da „gierig“ in jedem State die Action ausgewählt wird, welche (basierend auf der aktuellen Schätzung/Erfahrung) den höchsten zukünftigen kumulativen Reward verspricht. Da die Value Function aber nur eine Schätzung ist und sich mit der Erfahrung (Training) immer verbessern sollte, kann sich der Agent durch eine reine greedy Policy auf anfänglich zu hoch eingeschätzte Actions festfahren. Er findet somit nur eine Art lokales Maximum.

2.4.1 ϵ -greedy Policy

Um einen Trade-Off zwischen den beiden Extremen zu finden, kann beispielsweise als Policy π eine ϵ -greedy Policy gewählt werden. Dabei wird in $\epsilon\%$ der Fälle einfach irgendeine Action zufällig ausgewählt (egal wie gross ihre Action Value ist), und in den restlichen $(1 - \epsilon)\%$ der Fälle wird greedy (gierig) gehandelt, d.h. es wird die Action gewählt, welche (laut der momentanen Value Function) den höchsten zukünftigen Reward verspricht. Oft wird das ϵ auch im Laufe des Trainings reduziert, damit am Anfang eine grössere Exploration stattfindet, zum Schluss hingegen mehrheitlich (oder nur noch)

greedy gehandelt wird, da dann die Value Function schon recht gut approximiert sein sollte.

2.5 Q-Learning

Ein bekanntes Beispiel eines RL-Algorithmus ist der sogenannte Q-Learning Algorithmus. Bei diesem benutzt & aktualisiert der Agent die Action Value Function. Zur Erinnerung: Die Action Value Function sagt zu jedem State und zu jeder dazu möglichen Action aus, wie viel zukünftiger Reward man schätzungsweise erwarten kann, wenn man sich in besagtem State befindet und besagte Action ausführt (und anschliessend stets die Policy π befolgen würde). Wenn sich der Agent nun im State s_t befindet, eine Action a_t ausführt, kommt er dadurch in einen neuen (evtl. nicht immer gleichen) Nachfolge-State s_{t+1} und erhält den Reward R_{t+1} . Beim Q-Learning aktualisiert der Agent seine Action Value Function nun folgendermassen:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left(R_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right) \quad (2.3)$$

Hier ist α die *stepsize* (aka *learning rate* oder Lernrate). Gleichung (2.3) sagt also in anderen Worten aus:

$$\text{action-value}_{\text{neu}} = \text{action-value}_{\text{alt}} + \alpha (\text{Schätzung}_{\text{neu}} - \text{action-value}_{\text{alt}})$$

Dies ist ein in der Optimierung vielfach vorkommende Gleichung. Beim Q-Learning wird also der Q-Value (Action Value) ein wenig in Richtung der bestmöglichen Q-Values aktualisiert. Das bedeutet, dass der Q-Value den zukünftigen, kumulativen Reward schätzt, welcher erwartet werden kann, wenn in dem besagten State die besagte Action ausgeführt wird, und anschliessend die greedy Policy befolgt würde (also immer die Action mit dem höchsten Q-Value ausgewählt würde). Da die greedy-Policy nicht zwingend die momentan befolgte Policy (beispielsweise ϵ -greedy) sein muss, ist Q-Learning ein sogenannter Off-Policy-Algorithmus.

Der ganze Algorithmus sieht im Pseudocode wie folgt aus:

```

Initialize  $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$ , arbitrarily, and  $Q(\text{terminal-state}, \cdot) = 0$ 
Repeat (for each episode):
    Initialize  $S$ 
    Repeat (for each step of episode):
        Choose  $A$  from  $S$  using policy derived from  $Q$  (e.g.,  $\epsilon$ -greedy)
        Take action  $A$ , observe  $R, S'$ 
         $Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
         $S \leftarrow S'$ ;
    until  $S$  is terminal

```

Abbildung 2.2: Q-Learning Algorithmus [6]

Durch das Interagieren mit der Umgebung und dem Anwenden der Update-Gleichung (2.3) werden mit der Zeit jene Actions einen hohen Value erhalten, welche zu einem (evtl. erst später aufgetretenen) hohen Reward geführt haben. Schlechte Actions hingegen werden bekommen ihre Value nach unten korrigiert. Dies resultiert in einer immer

besseren Schätzung der Action Values und somit in einer besseren Performance des RL-Agenten.

2.5.1 Beispiel Windy Gridworld

Das Konzept von RL soll hier am Beispiel einer Gridworld (Rasterwelt) dargestellt werden, welche in Abbildung 2.3 abgebildet ist. Es wird der Q-Learning Algorithmus verwendet, um den bestmöglichen Weg vom Start (S) zum Goal (G) zu finden. Der Agent startet also stets auf der S-Kachel. Jede Kachel stellt einen State dar. Der Agent kann nun in jedem State eine von 4 Actions auswählen, welche die Richtung angeben, in der er sich bewegen soll. In einem State am Rande der Gridworld bleibt der Agent in dem State, sollte er eine Action „gegen die Wand“ auswählen.

Der Agent weiss nicht, wo sich sein zureichendes Ziel (die G-Kachel = der Terminal State) befindet. Er weiss auch nicht, dass ihn in gewissen Spalten ein Wind jeweils eine oder zwei Kacheln nach oben bläst. Wenn der Agent beispielsweise nach rechts geht und auf einem Feld mit Windstärke 1 kommt, wird er zusätzlich noch ein Feld nach oben transportiert, bevor der Zeitschritt vorbei ist. Der Agent erhält in jedem Zeitschritt einen Reward von -1, ausser im (Terminal) State G, wo er einen Reward von 0 erhält. Hat er das Goal erreicht, ist die Episode beendet und er startet wieder im Start-State S.

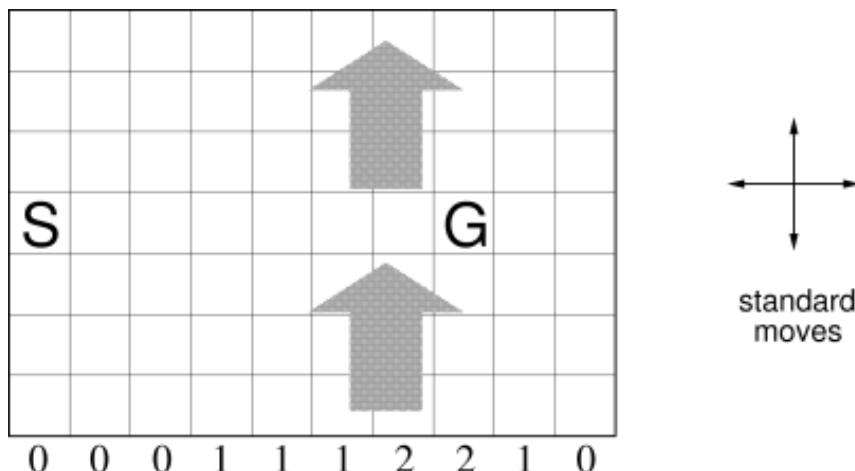


Abbildung 2.3: Windy Gridworld mit Start- und Goal-State. Die Zahlen unterhalb geben die Windstärke an. [6]

Am Anfang des Trainings wird der Agent nun verschiedenste Actions ausprobieren und planlos auf der Gridworld umherirren. Irgendwann wird er aber schliesslich sein Ziel einmal zufällig erreichen, und einen höheren Reward als sonst erhalten. Durch das Q-Learning erhält nun die Action Value des letzten State-Action-Paares unmittelbar vor dem Erreichen des Ziels einen höheren Q-Value. Wird der Agent in einer späteren Episode wieder in diesen State gelangen, wird er diese Action eher wieder auswählen, da sie ihn in der Vergangenheit zum Ziel gebracht hat. So findet der Agent in vielen durchgespielten Episoden heraus, welche Action in dem jeweiligen State ihn am wahrscheinlichsten zum Ziel führen. In Abbildung 2.4 ist die State Value Function des Agenten nach 1'000 Episoden dargestellt. Zusätzlich ist der letzte gewählte Weg des Agenten eingezeichnet. Deutlich zu sehen ist, dass dies kein planloses Herumirren mehr ist, sondern eine sichtbare Strategie (Policy), wie er sein Ziel möglichst rasch erreichen kann.

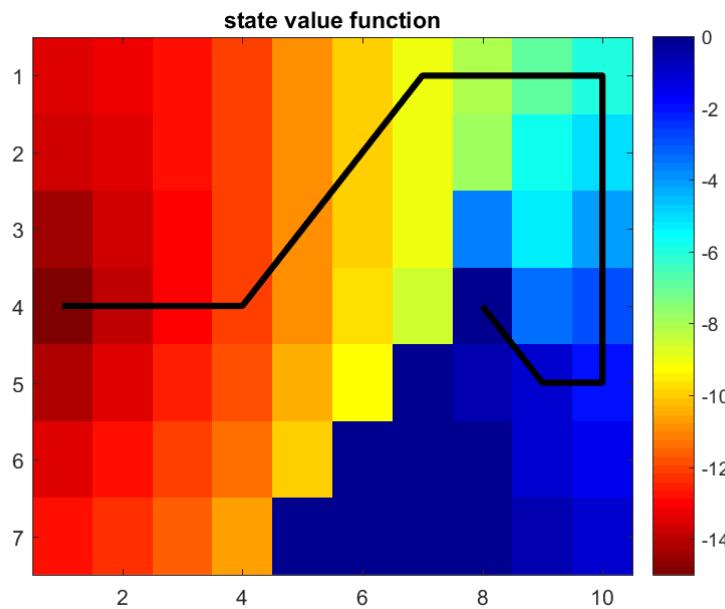


Abbildung 2.4: State Value Function der Windy Gridworld nach 1'000 Trainings-Episoden mit dem Q-Learning Algorithmus. Schwarz eingezeichnet ist der Weg des Agenten in der letzten Episode.

Bemerkung: Die State Values können aus den Action Values berechnet werden, indem für jeden State seine vier dazugehörigen Action Values aufsummiert werden. Der Code für das Beispiel stammt ursprünglich von [12] und wurde für das Q-Learning angepasst.

2.6 Kontinuierliche State-Spaces

Bisher wurden die Value Functions als diskrete Lookup-Tables behandelt, welche für jeden State bzw. für jedes State-Action-Paar einen Wert speichert, wie dies im Windy Gridworld Beispiel der Fall ist. Doch die grosse Mehrheit an Anwendungen hat entweder einen riesig grossen oder gar einen kontinuierlichen State-Space. Möchte man einen solchen Task trotzdem mit diskreten Lookup-Tables lösen, tritt das Phänomen der sogenannten „Curse of Dimensionality“ auf. Da jeder State mehrmals besucht werden muss, um seine Value Function gut approximieren zu können, würde sich die Trainingszeit bei Millionen von möglichen States exponentiell verlängern. Es muss also einen anderen Weg geben.

Die Lösung für das Problem der kontinuierlichen State-Spaces ist die *Value Function Approximation*. Hierbei muss in der Value Function nicht jeder einzelne State explizit gespeichert werden, sondern es reicht zu wissen, dass z.B. ein State 1 mm neben einem bekannten State einen ähnlichen erwarteten zukünftigen Reward haben sollte wie der des bekannten States. Die Value Function kann somit durch eine beliebige Approximation Function geschätzt werden:

$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s) \quad (2.4)$$

oder

$$\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a) \quad (2.5)$$

\mathbf{w} ist hierbei ein Gewichtungs-Vektor (z.T. auch mit θ angegeben). Der Agent muss nun im Laufe des Trainings diesen Gewichtungs-Vektor so anpassen, damit die echte Value Function möglichst gut approximiert wird. Dabei kann die Function Approximation beliebig gebildet werden, wie z.B. durch eine einfache gewichtete Linear-Kombination von Features des States oder durch ein neuronales Netzwerk.

Anstatt die Value Function direkt zu aktualisieren (wie beispielsweise beim Q-Learning), werden nun die Gewichte der Function Approximation angepasst. Das Prinzip der Update-Gleichung bleibt aber dasselbe wie beim Q-Learning. Hier hinzu kommt nun jedoch der Gradient der Function Approximation in Bezug auf dessen Gewichte, wie in Gleichung 2.6 zu sehen ist.

$$\Delta \mathbf{w} = \alpha (R_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, \mathbf{w}) - \hat{q}(s_t, a_t, \mathbf{w})) \nabla_{\mathbf{w}} \hat{q}(s_t, a_t, \mathbf{w}) \quad (2.6)$$

2.6.1 Deep Q-Learning (DQN)

Ein Algorithmus, welcher mit kontinuierlichen State-Spaces umgehen kann, ist der sogenannte Deep Q-Learning Algorithmus (DQN) von [13]. Damit konnten bereits diverse Atari-Videospiele anhand von rohen Pixel-Daten gelernt werden – zum Teil mit besserer Performance als ein menschlicher Spieler. In diesem Unterabschnitt wird das Konzept kurz vorgestellt. Für ein besseres und tieferes Verständnis wird der interessierte Leser auf die Literatur verwiesen.

Beim Deep Q-Learning wird die Action Value Function durch ein neuronales Netz approximiert (beispielsweise ein Convolutional Neural Net) mit den Parametern θ . Dabei ist der Input des Netzes der momentane State, der Output die (approximierte) Action-Value für jede mögliche Action. Ist also z.B. der State ein Bild eines Atari-Videospiels mit 84x84 Pixel (in Graustufen), so hat der Input ebenfalls diese Grösse. Wenn bei dem Videospiel 4 Actions möglich sind (z.B. nach links & rechts gehen, springen und ducken), so hat das neuronale Netz ebenfalls 4 Outputs. Dabei kann das Netz verschiedenste sogenannte *hidden layers* aufweisen, weshalb der Algorithmus auch *Deep Q-Learning* heisst.

Die Aktualisierung der Action Value Function geschieht zwar nach dem gleichen Prinzip wie beim herkömmlichen Q-Learning, wird aber aufgrund des neuronalen Netzes etwas anders formuliert. Es wird nur die aktuelle Schätzung des Q-Values für das State-Action-Paar aus der Update-Gleichung des herkömmlichen Q-Learnings (2.3) verwendet. Dieses sogenannte Target y ist also folgendermassen definiert:

$$y = R_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1} | \theta) \quad (2.7)$$

Das neuronale Netz wird dann so gefittet, dass die (quadrierte) Differenz zwischen dem Target und der durch das Netzwerk approximierten Action Value für das State-Action-Paar möglichst gering ist (siehe (2.8)). Diese Differenz bildet die sogenannte Loss-Function des Netzwerkes, welche beispielsweise mit Gradient Descent minimiert werden kann.

$$L = \frac{1}{\text{mini-batch}} \sum_{i \in \text{mini-batch}} (y_i - Q(s_i, a_i | \theta))^2 \quad (2.8)$$

Der Pseudocode des Deep Q-Learnings ist in Abbildung 2.6, eine Illustration davon in Abbildung 2.5 zu sehen.

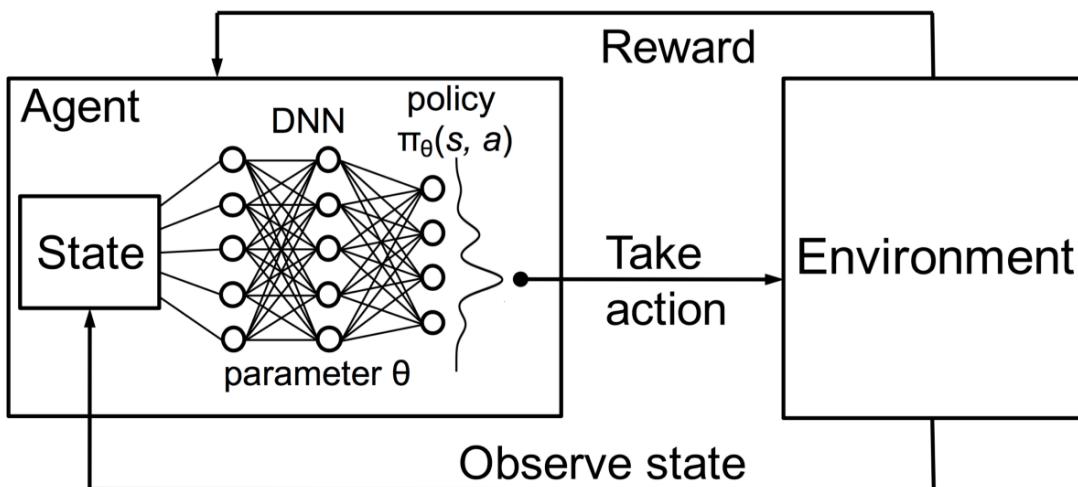


Abbildung 2.5: Illustration des DQN-Algorithmus [14]

Algorithm 1 Deep Q-learning with Experience Replay

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
    Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
    for  $t = 1, T$  do
        With probability  $\epsilon$  select a random action  $a_t$ 
        otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
        Execute action  $a_t$  in emulator and observe reward  $r_t$  and image  $x_{t+1}$ 
        Set  $s_{t+1} = s_t, a_t, x_{t+1}$  and preprocess  $\phi_{t+1} = \phi(s_{t+1})$ 
        Store transition  $(\phi_t, a_t, r_t, \phi_{t+1})$  in  $\mathcal{D}$ 
        Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
        Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_a Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
        Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
    end for
end for

```

Abbildung 2.6: Pseudocode des DQN-Algorithmus [13]

2.6.1.1 Experience Replay

Ein weiterer Trick des Deep Q-Learnings ist das Speichern der beim Training erlebten „Transitions“ (s_t, a_t, R_t, s_{t+1}) für jeden Zeitschritt t . Aus diesem sogenannten *Replay Memory* werden dann für das Fitten des neuronalen Netzes ein zufälliger Minibatch gesampelt. Dies dient dazu, dass die Trainings-Daten für das neuronale Netz nicht so stark miteinander korreliert sind, als wenn einfach jeweils die aktuellsten Transitions verwendet werden. Da ein Nachfolge-State in der Regel stark mit dem vorherigen State korreliert ist, kann die Korrelation der Trainings-Daten durch die nicht-chronologische und zufällige Zusammenstellung der Minibatches etwas reduziert werden. Eine wichtige Grundannahme für viele Machine Learning Algorithmen ist nämlich, dass die Trainings-Daten *i.i.d.* (*independent and identically distributed*, unabhängig und identisch verteilt) sind. Diese Eigenschaft wird mit dem zufälligen sampeln der Transitions zwar nicht

gänzlich, aber immerhin etwas besser erreicht als ohne. Diese Technik nennt man *Experience Replay*.

2.6.1.2 Target Networks

Ein weiterer, vielfach eingesetzter Trick ist das Verwenden zweier neuronalen Netze: ein *Target-* und ein *Fitting*-Netz. Beide Netze sind von der Struktur identisch. Das Fitting-Netz wird für das Fitten anhand des gesampelten Minibatches verwendet, d.h. dessen Gewichte werden in jedem Zeitschritt angepasst. Das Target-Netz jedoch hat seine Gewichte für eine längere Zeit fixiert. Es wird verwendet, um die nächste auszuführende Action auszuwählen. Die Gewichte des Target-Netzes werden dann periodisch mit den aktuellen Gewichten des Fitting-Netzes überschrieben. Dies führt zu einer grösseren Stabilität der Action Value Function Approximation, da das Target-Netzwerk, welches für die Interaktion mit dem Environment zuständig ist, sich nicht in jedem Zeitschritt ändert.

2.7 Kontinuierliche Action-Spaces

Eine weitere Schwierigkeit für das RL ergibt sich, wenn nicht nur der State-Space, sondern auch der Action-Space kontinuierlich ist. Gerade in Bezug auf Robotik-Anwendungen ist dies sehr häufig der Fall. Ein Lösungsansatz könnte die einfache Diskretisierung des Action-Spaces darstellen. Dieser Ansatz ist jedoch weder wirklich leistungsfähig, noch auf einen grossen Action-Space praktisch anwendbar (siehe auch Unterabschnitt 7.3.1). Deshalb muss anstatt der Value Based Methode eine Policy Based Methode oder gar eine Actor-Critic Methode angewendet werden. Da dies etwas fortgeschrittenere Algorithmen sind, wird in diesem Abschnitt nur das Konzept kurz erklärt. Für eine vollständigere und tiefere Erklärung wird wiederum auf die Literatur verwiesen. Weitere hilfreiche Quellen für den ebenfalls in diesem Abschnitt kurz gezeigten DDPG-Algorithmus sind u.a.:

- DDPG-Paper [15]
- Blog über DDPG [16]
- Lecture 14 des Stanford-Kurses Visual Recognition (Video) [17]
- Lecture 7 des RL-Kurses von David Silver [6]

2.7.1 Policy Gradient

Policy Gradient bezeichnet eine Familie von Policy Based Methods. Hierbei ist die Policy explizit als Funktion vorhanden und wird direkt während des Trainings des Agenten verbessert:

$$\pi_{\theta}(s, a) = \mathbb{P}[a | s, \theta] \quad (2.9)$$

Hier sind θ die Parameter der Policy Function, was beispielsweise Gewichte eines neuronalen Netzes sein können. Die Policy Function wird ähnlich wie die Value Function Approximation mithilfe eines Gradienten aktualisiert und somit verbessert:

$$\Delta\theta = \alpha \nabla_{\theta} J(\theta) \quad (2.10)$$

$J(\theta)$ ist die sogenannte *objective function*. Damit wird die Performance der Policy gemessen, welche es zu maximieren gilt. Es können verschiedene Ansätze für die „Messung“ verwendet werden, beispielsweise der gemittelte State Value aller besuchten States. Die Maximierung der objective function geschieht mit Gradient Ascent, weshalb der Algorithmus auch Policy Gradient heisst. Es kann gezeigt werden, dass der Gradient $\nabla_{\theta}J(\theta)$ durch einen anderen Ausdruck approximiert werden kann:

$$\Delta\theta = \alpha \nabla_{\theta} \log (\pi_{\theta}(s, a)) Q_w(s, a) \quad (2.11)$$

Zu beachten ist, das hier wieder eine Schätzung des Action Values $Q_w(s, a)$ benötigt wird. Somit ist dies keine reine Policy Based Methode, sondern eine Actor-Critic Methode. Hierbei werden zwei Function Approximations verwendet: eine für die Policy (Actor) mit den Parametern θ , und eine für die Action Value Function (Critic) mit den Parametern w . Der Actor aktualisiert also seine Policy Parameter θ in die Richtung, welche durch den Critic durch die Value Function vorgeschlagen wird. In anderen Worten, der Actor entscheidet anhand des momentanen States, welche Action auszuführen ist. Der Critic bewertet anschliessend diese Action mittels der Action Value Function.

Eine Illustration der Actor-Critic-Methode ist in Abbildung 2.7 dargestellt. In Abbildung 2.8 ist der Pseudocode eines Beispiel-Actor-Critic-Algorithmus zu sehen.

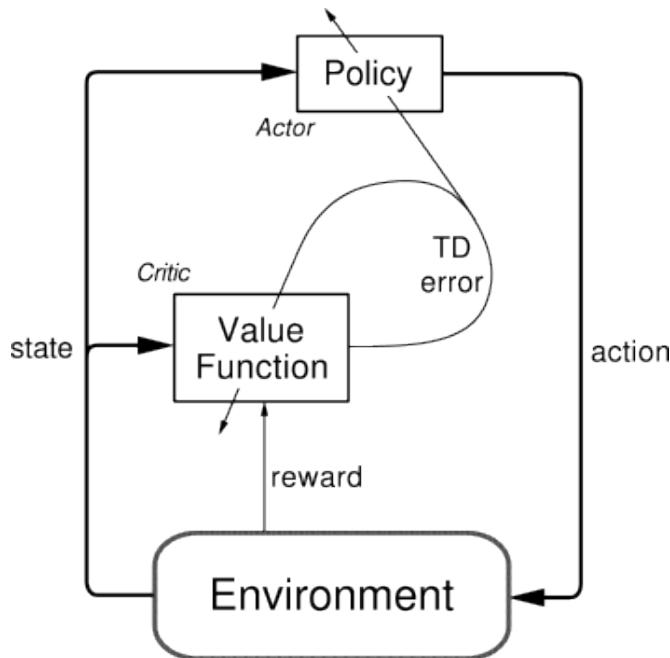


Abbildung 2.7: Illustration der Actor-Critic-Methode [5]

2.7.2 Deep Deterministic Policy Gradient (DDPG)

Der Deep Deterministic Policy Gradient Algorithmus (DDPG) ist eine Variante einer Actor-Critic Methode. Der Actor (Policy Function) und der Critic (Action Value Function) werden dabei jeweils durch ein neuronales Netz realisiert. Das Actor-Netzwerk hat als Input den State (beispielsweise ein Bild) und als Output die nächste auszuführende Action (z.B. die nächste anzufahrende Position eines Roboters). Das Critic-Netzwerk hingegen hat als Input den State und die gewählte Action, als Output die approximierte Action Value des State-Action-Paares vom Input, also eine skalare, reelle Zahl. Die Loss-Function des Critics ist dabei dieselbe wie beim DQN-Algorithmus (siehe (2.8)).

```

function QAC
    Initialise  $s, \theta$ 
    Sample  $a \sim \pi_\theta$ 
    for each step do
        Sample reward  $r = \mathcal{R}_s^a$ ; sample transition  $s' \sim \mathcal{P}_{s,a}$ 
        Sample action  $a' \sim \pi_\theta(s', a')$ 
         $\delta = r + \gamma Q_w(s', a') - Q_w(s, a)$ 
         $\theta = \theta + \alpha \nabla_\theta \log \pi_\theta(s, a) Q_w(s, a)$ 
         $w \leftarrow w + \beta \delta \phi(s, a)$ 
         $a \leftarrow a', s \leftarrow s'$ 
    end for
end function

```

Abbildung 2.8: Pseudocode eines Actor-Critic-Algorithmus [6]

Die Gewichte des Actor-Netzwerkes werden durch die ebenfalls bereits bekannte Update-Gleichung (2.11) aktualisiert, was ebenfalls ein Gradient Descent Ansatz ist (wobei hier streng genommen Gradient Ascent verwendet wird, da ein Maximum gesucht wird).

Wie auch bereits beim DQN-Algorithmus wird beim DDQG-Algorithmus die Experience Replay Technik angewandt, wie auch das Verwenden von Target-Networks. In Abbildung 2.9 zu sehen ist das leicht andere Aktualisieren der Gewichte des Target-Netzes. Wurden beim DQN-Algorithmus die Gewichte des Target-Netzes periodisch einfach überschrieben, wird beim DDPG in jedem Zeitschritt ein Soft-Update der Gewichte durchgeführt. Dabei werden mit den (Hyper-) Parameter $\tau \in [0, 1]$ die Gewichte der beiden Target-Netze (Actor & Critic) „ein wenig in Richtung der Fitting-Netz-Gewichte“ angepasst. Wird dabei τ auf 0 gesetzt, werden die Gewichte der Target-Netze wie beim DQN-Algorithmus einfach überschrieben (hier aber in jedem Zeitschritt).

Der Pseudocode des DDPG-Algorithmus ist in Abbildung 2.9 dargestellt.

2.7.2.1 Exploration mit dem DDPG-Algorithmus

Anders als beim DQN-Algorithmus kann beim DDPG die Exploration (siehe Abschnitt 2.4) nicht mehr mit einer ϵ -greedy Policy gelöst werden, da ja die Policy als explizite Funktion durch ein neuronales Netz gegeben ist und direkt die nächste auszuführende Action anhand des momentanen States berechnet. Deshalb muss hier für einen Exploration-Effekt ein Rauschen (Noise) auf die durch den Actor vorgeschlagene Action hinzugefügt werden. So kann beispielsweise jeder Dimension einer anzufahrenden Position eines Roboters ein Gaußsches Rauschen mit Mittelwert 0 hinzugefügt werden. Je höher dabei die Varianz des Rauschens, desto grösser die Exploration. Eine leicht andere Variante fügt nicht der Action selbst einen Noise hinzu, sondern den Gewichten des Actor-Target-Netzes. Dieser Ansatz wird im Unterabschnitt 7.4.2 beschrieben.

Algorithm 1 DDPG algorithm

Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ .

Initialize target network Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q, \theta^{\mu'} \leftarrow \theta^\mu$

Initialize replay buffer R

for episode = 1, M **do**

 Initialize a random process \mathcal{N} for action exploration

 Receive initial observation state s_1

for t = 1, T **do**

 Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise

 Execute action a_t and observe reward r_t and observe new state s_{t+1}

 Store transition (s_t, a_t, r_t, s_{t+1}) in R

 Sample a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) from R

 Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$

 Update critic by minimizing the loss: $L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$

 Update the actor policy using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$$

 Update the target networks:

$$\theta^{Q'} \leftarrow \tau \theta^Q + (1 - \tau) \theta^{Q'}$$

$$\theta^{\mu'} \leftarrow \tau \theta^\mu + (1 - \tau) \theta^{\mu'}$$

end for

end for

Abbildung 2.9: Pseudocode des DDPG-Algorithmus [15]

3 Abklärungen & Entscheide

Nachdem das Reinforcement Learning im letzten Kapitel behandelt wurde, bedarf es vor der eigentlichen Implementierung des ganzen Systems mit dem Roboter zuerst einiger Abklärungen, welche in diesem Kapitel dokumentiert werden.

3.1 RL mit Robotik

Reinforcement Learning kombiniert mit Robotik ist ein vielversprechendes Gebiet, da die Konzepte von RL intuitiv auch auf Roboter anwendbar sind. Jedoch sind hier einige Probleme zu beachten. Da der Roboter über viele Trainings-Episoden mit seiner Umgebung interagieren muss, um seinen Task zu erlernen, würden bei einem realen Roboter die mechanischen Teile rasch abgenutzt. Dies hätte nicht nur grösseren Unterhalt zu folge, die Interaktionen des Roboters mit seiner Umgebung würde ebenfalls ungenauer. Zudem könnte – je nach Roboter und seinem Task – das Bewegen des realen Roboters während des Trainings potentiell gefährlich sein – sei es für Lebewesen oder Bauteile.

Eine Simulation des Roboters samt seiner Umgebung bietet sich deshalb als sinnvolle Lösung dieser Probleme an. Eine Simulation steigert nicht nur die Flexibilität, sondern der Trainings-Prozess kann zudem auch noch schneller als mit dem realen Roboter durchgeführt werden. Zusätzlich kann das Training des RL-Agenten über eine praktisch unbegrenzte Zeit und ohne Beaufsichtigung geschehen. Der Reward kann ausserdem ebenfalls über die API des Simulators berechnet werden, was ein weiterer Vorteil der Simulation ist. Deshalb gilt es nun, ein Roboter-Simulationsprogramm auszuwählen.

3.2 Auswahl des Simulationsprogramms

Es gibt eine Vielzahl von Roboter-Simulatoren. Teil dieses Projekts ist es, einen für den Roboter und den RL-Task geeigneten Simulator auszuwählen. In Tabelle 3.1 sind die in Betracht gezogenen Simulatoren mit ihren Vor- und Nachteilen aufgelistet. Dabei ist die Schnittstelle zwischen dem RL-Algorithmus und der Simulation ein wichtiger Punkt. Ein mögliches Framework für die Implementierung von RL-Algorithmen bietet beispielsweise OpenAI Gym (siehe Abschnitt 3.3), weshalb eine Anbindung an dieses Tool beispielsweise von Vorteil wäre. Aber auch die Schnittstelle zwischen Simulation & realem Roboter ist von grosser Bedeutung. Diese Schnittstelle gibt an, wie das in der Simulation ausgeführte auch auf den realen Roboter übertragen werden kann. Die meisten Simulatoren benötigen ein Unix-basiertes Betriebssystem (meistens Ubuntu 16.04).

Die Entscheidung fiel schliesslich auf das Simulationsprogramm *Gazebo*. Gazebo ist ein Roboter-Simulator, welcher eine hervorragende Integration von ROS bietet. Viele Roboter- und Sensor-Modelle sind bereits vorhanden, und mithilfe von sogenannten URDF- oder SDF-Files können eigene Roboter und Sensoren in Gazebo eingefügt werden. Ein Screenshot von Gazebo ist in Abbildung 3.1 abgebildet. Mehr zu Gazebo in Kapitel 4.

ROS (Robot Operating System) ist ein modulares Open Source Framework für Robotik-Anwendungen, welches sehr weit verbreitet ist und in zahlreichen industriellen wie

Tabelle 3.1: Vor- und Nachteile der in Betracht gezogenen Roboter-Simulatoren

Simulator	Vorteile	Nachteile
OpenRave	<ul style="list-style-type: none"> • Python-API vorhanden (gut für Schnittstelle zu RL-Bibliothek) • Erfolgreiche erste Installation & Hello-World-Programm rasch zum Laufen gebracht 	<ul style="list-style-type: none"> • kleine Community • schlechte Dokumentation • eher für Low-Level-Ansteuerung des Roboters (Gelenk-Winkel)?
Gazebo	<ul style="list-style-type: none"> • gute Dokumentation • verständliche Tutorials • Anbindung zu OpenAI Gym möglich • Ansteuerung über C++ & Python-API möglich • Schnittstelle zwischen Simulation & realem Roboter bekannt, sofern bei beidem ROS verwendet wird • bereits etwas bekannt im EMS-Institut der NTB 	<ul style="list-style-type: none"> • nur für ROS-gesteuerte Roboter? • aufwändige Installation
KUKA.Sim Pro	<ul style="list-style-type: none"> • Schnittstelle zwischen Simulation & realer Roboter wahrscheinlich einfach (sofern ein KUKA-Roboter verwendet wird) 	<ul style="list-style-type: none"> • nur relativ neue KUKA-Roboter unterstützt • nicht gratis • Schnittstelle für RL?
MuJoCo	<ul style="list-style-type: none"> • Anbindung zu OpenAI Gym sehr einfach • viele RL-Algorithmen wurden mit MuJoCo schon getestet/entwickelt 	<ul style="list-style-type: none"> • nur Low-Level-Ansteuerung des Roboters (Gelenk-Winkel)? • Installation aufwändig & mehrfach fehlgeschlagen
Webots	<ul style="list-style-type: none"> • gute Dokumentation/Tutorials • Python/Java/C/Matlab-Schnittstellen • evtl. ROS-Einbindung möglich? 	<ul style="list-style-type: none"> • nicht gratis • Schnittstelle zu realem Roboter?

akademischen Anwendungen eingesetzt wird. ROS ist eine Art Middleware für verschiedenste Komponenten (Nodes), welche untereinander über wohl-definierte ROS-messages kommunizieren, welche über TCP/IP-Verbindungen geschickt werden. Diese ROS-messages werden meistens über das Publish-Subscribe-Pattern verschickt, unter sog. ROS-topics. Dank der Modularität von ROS ist es möglich, dasselbe Interface für die Ansteuerung des simulierten wie auch des realen Roboters zu verwenden. Mehr zu ROS in Kapitel 4.

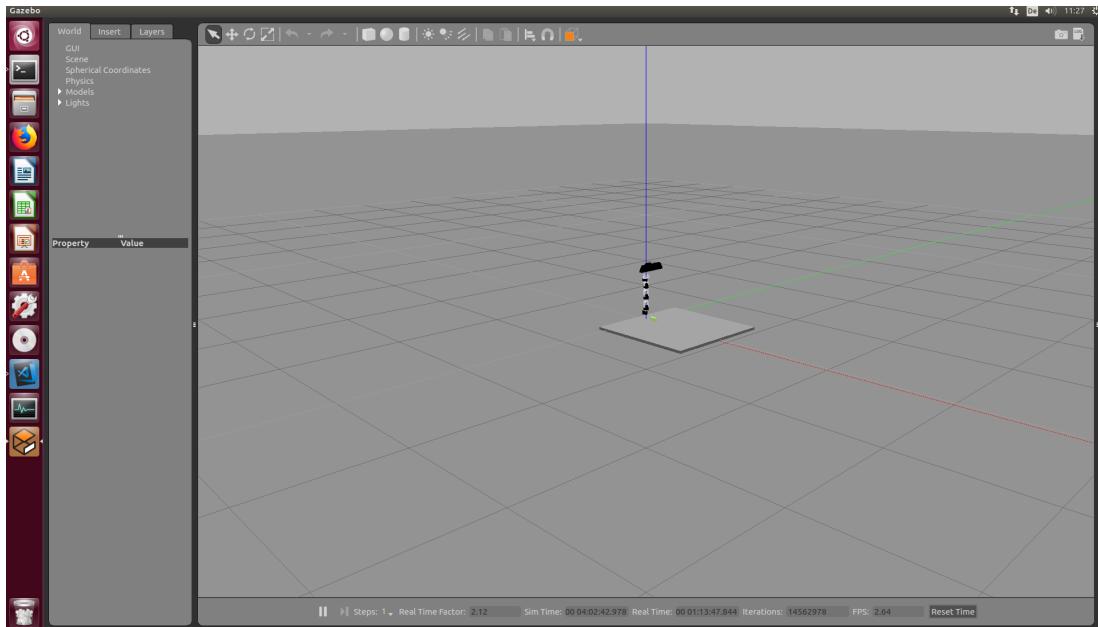


Abbildung 3.1: Gazebo-Simulator

3.3 Auswahl des RL-Toolkits

OpenAI Gym ist ein relativ neues Toolkit für das Entwickeln und Testen verschiedenster RL-Algorithmen in Python. Es definiert eine einheitliche Schnittstelle zwischen den Agenten, dem RL-Algorithmus und der Umgebung (*environment*). Da OpenAI Gym zum Zeitpunkt des Projektes das einzige RL-Toolkit mit einer genügend grossen Community war, fiel die Entscheidung rasch.

In OpenAI Gym gibt es bereits eine Vielzahl an sogenannter *environments*. Ein environment bildet die Umgebung bei einem Reinforcement Learning Task. Mehr zu OpenAI Gym in Kapitel 5.

3.4 Auswahl des Roboters

Limitiert wird der zu erlernende Task des Roboters vielmehr von den Möglichkeiten des Simulators als von der Fantasie des Projektbearbeiters. Deshalb sollte nicht nur der im Rahmen dieses Projektes verwendete Roboter im Gazebo-Simulator simulierbar (d.h. als Modell vorhanden & ansteuerbar) sein, sondern auch alle Komponenten der Umgebung wie Objekte und Sensoren. Zudem sollte der Roboter samt den benötigten Sensoren auch *physisch* vorhanden & ansteuerbar sein, damit im Falle eines erfolgreichen Trainings des RL-Algorithmus in der Simulation das Gelernte in der realen Welt umgesetzt und verifiziert werden kann. Da Gazebo als Simulator eingesetzt wird, wäre es zudem

von grossem Vorteil, wenn der Roboter und die Sensoren über ROS angesteuert bzw. ausgelesen werden könnten.

Der schliesslich in dieser Arbeit verwendete Roboter ist der *PhantomX Pincher Robot Arm* (folgend *Pincher* genannt). Dies ist ein kleiner Roboter-Arm mit 4 Freiheitsgraden (Degrees of Freedom, DOF). Die Wahl fiel auf den Pincher, da bereits ein Github-Repository existiert, mit welchem man den realen Arm über ROS ansteuern kann (siehe [18]). Dass der Roboter aber auch physisch zur Verfügung steht, mit ROS angesteuert werden kann, relativ klein und handlich ist und nicht anderweitig gebraucht wurde, waren weitere wichtige Gründe für den Pincher. Die Nachteile des Pinchers sind jedoch, dass noch kein Gazebo-Simulations-Package vorhanden ist und er durch seine geringe Anzahl an Freiheitsgraden weniger mögliche Positionen anfahren kann.



Abbildung 3.2: PhantomX Pincher Roboter-Arm [19]

3.5 Auswahl des Tasks für den Roboter

Die genaue Aufgabe (Task) für den Roboter zu definieren ist ebenfalls Teil dieser Arbeit. Dabei ist der zu lernende Task abhängig von den technischen & zeitlichen Möglichkeiten im Rahmen des Vertiefungsprojektes.

3.5.1 Möglichkeiten für Tasks

Folgend aufgelistet sind mögliche Tasks für den ausgewählten Roboter, welche in Betracht gezogen wurden:

- **Mikado:** Roboter soll lernen, aus einem ungeordneten Haufen von Stäben einen frei wählbaren Stab so zu greifen und entfernen, damit sich keiner der anderen Stäbe bewegt
- **Bottle-Flip:** Roboter soll lernen, eine Flasche/ein Objekt so zu werfen, damit sie einen Salto steht
- **Ballwurf:** Roboter soll lernen, einen Ball in ein Ziel zu werfen
- **Kegeln:** Roboter soll lernen, eine Kugel so zu rollen, damit diese möglichst viele Kegel umwirft
- **Greifen:** Roboter soll lernen, bestimmte Objekte zu greifen (welche evtl. nicht immer exakt gleich aussehen)

Andere, vor der Wahl des verwendeten Roboters in Betracht gezogene mögliche Tasks waren zudem eine Drohne, welcher autonom durch einen Wald fliegen kann und ein fahrender Roboter, welcher autonom (nicht immer gleichen) Hindernisse umfahren kann.

3.5.2 Entscheidung

Die Entscheidung fiel schliesslich auf den Greifen-Task, da dies nicht nur ein sehr oft vorkommendes Problem in der Robotik darstellt, sondern auch durch den gewählten Gazebo-Simulator am ehesten simuliert werden kann. Obwohl für diesen Task bereits zahlreiche anspruchsvolle & gut funktionierende Lösungen existieren (beispielsweise Pickit [20]), kann RL hier möglicherweise helfen, diesen Task des Greifens zu verbessern oder zu erweitern. Dies kann beispielsweise dann von Nutzen sein, wenn die zu greifenden Objekte nicht immer genau gleich aussehen, wie [2] zeigen konnten.

Der zu erlernende Task für den Roboter ist es, einen Quader aufzuheben, und zwar anhand eines Tiefenbildes von einer Kamera, welche oberhalb des Quaders montiert ist. Der Quader liegt zufällig platziert auf einer Fläche, auf welcher auch der Roboter-Arm montiert ist, wie in Abbildung 3.3 gezeigt ist.

Dabei macht es Sinn, nicht etwas mit RL lernen zu wollen, wofür es bereits eine mathematische Lösung gibt. Ein Beispiel dazu wäre das Erreichen eines Punktes im Raum mit dem Greifer des Roboter-Arms, wobei nur die Gelenk-Winkel des Armes gesteuert werden können. Dieser Vorgang kann durch die sogenannte Inverse Kinematik (IK) des Roboters berechnet werden. In diesem Projekt wird deshalb vom RL-Agenten die anzufahrende Position des Greifers vorgeschlagen, ein IK-Solver kümmert sich um die Positionierung des Greifers. Dadurch ist der gelernte Task zu einem gewissen Grade unabhängig vom verwendeten Roboter, was ein weiterer Vorteil dieser Vorgehensweise ist.

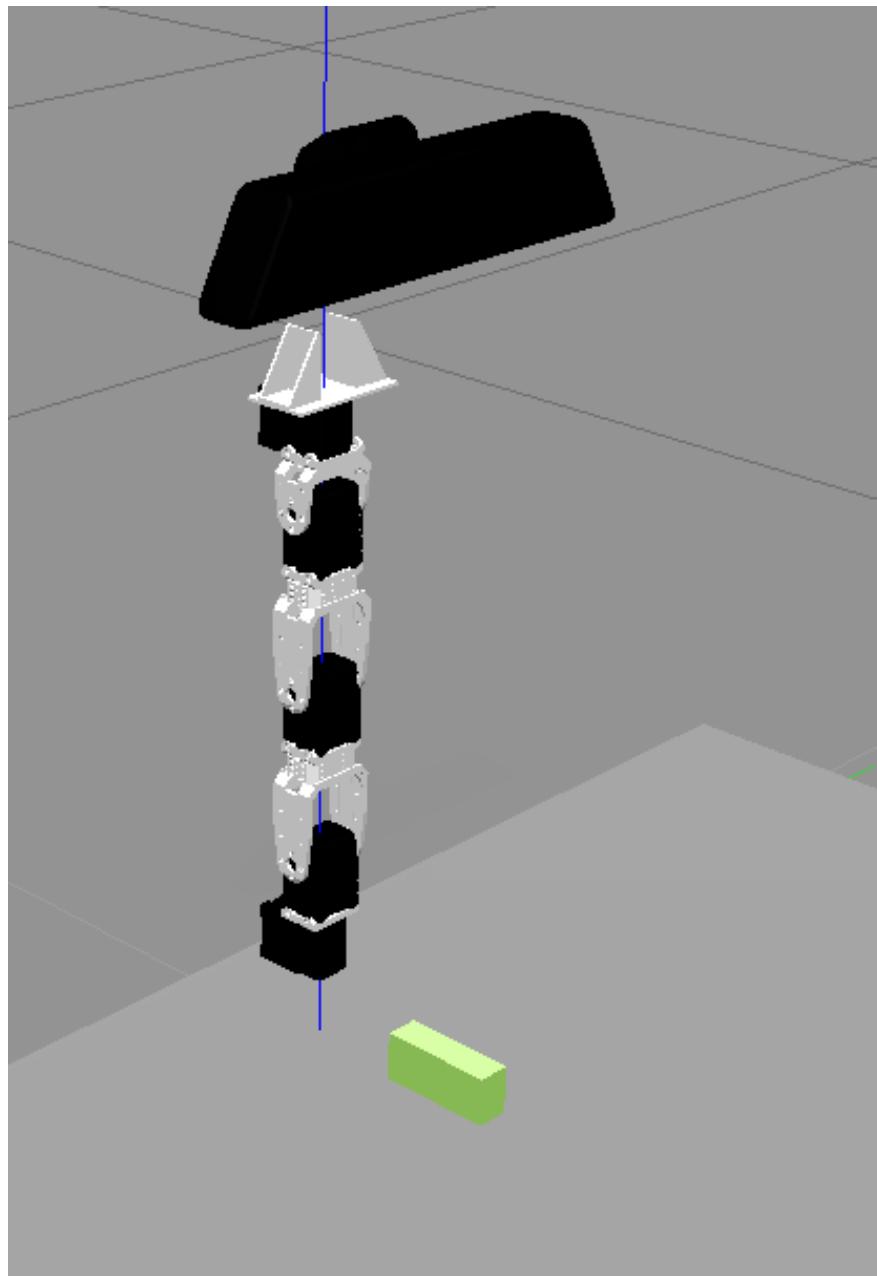


Abbildung 3.3: Simulations-Setup in Gazebo für den Greifen-Task

3.5.2.1 Exkurs Forward & Inverse Kinematics

Die sog. *Forward Kinematics* (FK) beschreibt die Position des Greifers eines Roboter-Armes im Raum mithilfe der gegebenen Gelenk-Winkel & Abmessungen des Roboters. Diese Berechnung ist nicht allzu schwierig, da die Lösung (d.h. die Position des Greifers) eindeutig bestimmbar ist. Die *Inverse Kinematics* (IK) jedoch ist viel schwieriger zu berechnen, da hier genau das Umgekehrte berechnet werden soll: damit der Greifer eine gewünschte Position im Raum erreichen kann, sollen alle dazugehörigen Gelenk-Winkel berechnet werden. Eine schematische Darstellung dazu ist in Abbildung 3.4 zu sehen. In den meisten Fällen gibt es für die IK viele verschiedene Lösungen, oder aber gar keine Lösung (wenn die Position für den Arm unerreichbar ist). Glücklicherweise gibt es aber verschiedenste Bibliotheken mit IK-Solvern, welche dieses Problem lösen. *MoveIt* [21] ist beispielsweise eine solche Bibliothek, welche in dieser Arbeit verwendet wird. Neben verschiedenensten IK-Solvern stellt MoveIt auch ganze Pfad-Planer zur Verfügung,

welche auch Hindernisse beachten können, damit der Roboter-Arm nicht auf dem Weg von Punkt A zu Punkt B mit seinen Gelenken oder dem Greifer in ein Objekt fährt.

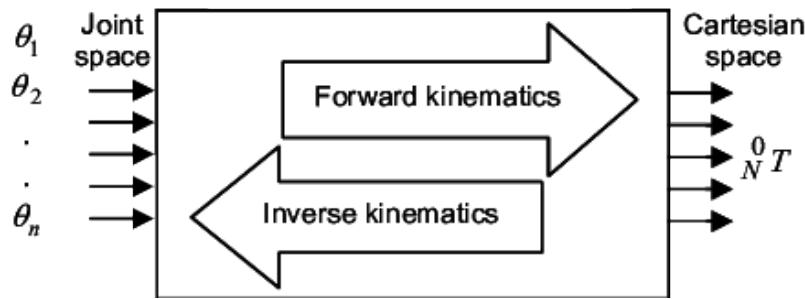


Abbildung 3.4: Schematische Darstellung der Forward & Inverse Kinematics [22]

3.5.3 Action & State-Space

Für den State des RL-Agenten ist das Tiefenbild gewählt worden. Der State-Space besteht also aus allen möglichen Tiefenbildern, was einen hoch-dimensionalen, kontinuierlichen State-Space bedeutet. Die Action, welche der Agent anhand des momentanen States (Tiefenbild) zu wählen hat, besteht aus der gewünschten Position & Orientierung (aka *Pose*) im Raum für den Greifer des Roboter-Armes. Die Position im Raum wird in Bezug auf das globale /world-Koordinatensystem der Gazebo-Simulationswelt angegeben. Die Orientierung im Raum wird durch die Winkel *roll*, *pitch* & *yaw* angegeben, wobei dies die Drehungen um die x-, y- und z-Achse bedeutet, wie in Abbildung 3.5 zu sehen ist. Das heisst, dass eine Action ein 6-dimensionaler, kontinuierlicher Vektor ist, mit den Einträgen (x, y, z, roll, pitch, yaw).

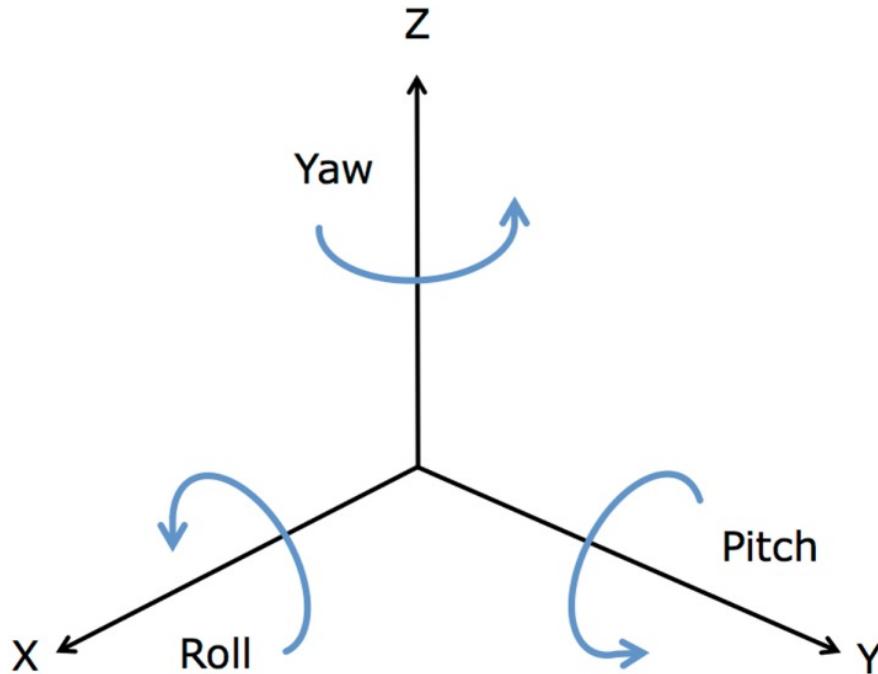


Abbildung 3.5: Roll-, Pitch- & Yaw-Winkel [23]

3.6 Auswahl des Sensors

Da nun der genaue Task des Roboters klar definiert ist, muss noch eine geeignete Kamera für das benötigte Tiefenbild gefunden werden. Dafür wird die Microsoft-Kamera *Kinect for Xbox 360* eingesetzt. Diese kann neben RGB-Bildern auch Distanz-Bilder (aka Tiefenbilder) und Punktewolken von der Umgebung erzeugen. Das Auslesen der Kamera über ROS-topics ist ebenfalls möglich, was neben der physischen Verfügbarkeit einer Kinect in der NTB ein weiterer Hauptgrund für die Verwendung in diesem Projekt darstellt.

3.7 Zusammenfassung & Überblick

Zusammengefasst wird in Abbildung 3.6 ein Überblick der verwendeten (Software-) Komponenten dieses Projektes gegeben. Der RL-Algorithmus befindet sich im Toolkit von OpenAI Gym. Für die Simulation des Roboter-Armes (Pincher) wird Gazebo verwendet. ROS fungiert als Middleware zwischen dem OpenAI Gym environment und der Gazebo-Simulation bzw. dem realen Roboter und der realen Kinect Kamera. Die gesamte Architektur läuft auf dem Linux-Betriebssystem Ubuntu 16.04 in einer virtuellen Maschine von VirtualBox.

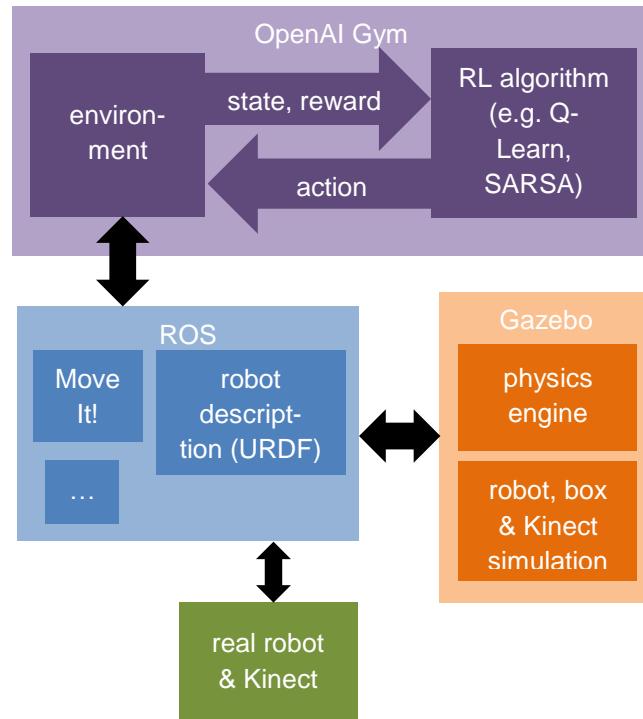


Abbildung 3.6: Überblick der Software-Komponenten

Diese Architektur ist auch bekannt als *gym-gazebo* [24] und existiert bereits für gewisse Roboter und environments. Die Aufgabe nun lautet, diese ganze Toolchain für den Roboter & Task dieses Projektes zu implementieren. Dies beinhaltet:

- eine Gazebo-Simulation des Pincher und der Kinect inkl. Ansteuerung via Python API über das ROS Framework → siehe Kapitel 4

- ein OpenAI Gym environment, welche die Simulation steuert und den Reward berechnet → siehe Kapitel 5
- eine reale Nachbildung des Simulations-Setups mit dem echten Pincher und der echten Kinect → siehe Kapitel 6
- ein RL-Algorithmus, welcher für den Task anwendbar ist → siehe Kapitel 7

3.7.1 Installation

Für die Installation aller Komponenten kann das Setup-Skript im Anhang zur Hilfe genommen werden. Dieses installiert auf einem frisch aufgesetzten Ubuntu 16.04 nicht nur ROS, Gazebo & OpenAI Gym, sondern lädt auch beispielsweise den in dieser Arbeit verwendeten Code für die RL-Algorithmen, die Files für die Simulation des Pincher-Armes und der Kinect in Gazebo oder die nötigen Treiber für den realen Pincher-Arm und die reale Kinect.

Die in diesem Projekt verwendeten Versionen sind:

- ROS: Kinetic
- Gazebo: 7.11.0
- OpenAI Gym: 0.9.6 (direkt von Github geklont)

4 Simulation

Wie bereits in Abschnitt 3.1 erwähnt, ist die Simulation des Roboters und seiner Umgebung (inkl. der Sensoren) ein essentieller Bestandteil beim Reinforcement Learning für Robotik-Anwendungen. Für die Simulation wird das Roboter-Simulationsprogramm Gazebo verwendet, welches über das Roboter-Framework ROS angesteuert werden kann. Es wird deshalb empfohlen, sich zuerst mit den Grundkonzepten von ROS & Gazebo auseinanderzusetzen, bevor auf die eigentliche Simulation des in diesem Projekt verwendeten Roboters einzugehen. Dazu wird hier auf folgende Quellen verwiesen, die einen einfachen Einstieg in die Grundkonzepte ermöglichen:

- MSE-Vertiefungsprojekt 1 „Einführung in ROS“ von Manuel Ilg [25] (siehe Anhang)
- „MME-Simulationsprojekt: ROS und Gazebo“ von Andreas Kalberer [26] (siehe Anhang)
- Offizielle Dokumentation & Tutorials ([27], [28], [29], [30])

4.1 Simulation Roboter-Arm

Für den Pincher existiert bereits ein Github-Repository [18]. Dieses beinhaltet unter anderem ein MoveIt-Package, mit welchem man den realen Arm über ROS ansteuern kann. Das Repository beinhaltet zudem auch einen Satz von URDF-Files, welche die Links & Joints des Pincher-Armes beschreiben. Jedoch existierte zum Zeitpunkt des Projektes noch kein brauchbares Gazebo-Package, weshalb dieses noch erstellt werden muss. Damit auch die Ansteuerung über MoveIt in der Simulation funktioniert, sind zudem weitere Komponenten und Änderungen nötig, welche in den nachfolgenden Abschnitten kurz erklärt werden. Eine Übersicht über die Packages des Github-Repositories (und die in diesem Projekt hinzugefügten Packages) ist in Tabelle 4.1 zu sehen.

4.1.1 Auswahl des Pincher-Armes mittels Umgebungsvariable

Eine Eigenheit des Github-Repositories ist, dass es für zwei Roboter-Arme erstellt wurde. So gelten gewisse Packages, Konfigurations-Files, Launch-Files etc. nicht nur für den Pincher, sondern auch für den (relativ ähnlichen) Turtlebot-Arm. Da der Turtlebot-Arm im Repository als Default-Arm verwendet wird, sind viele Packages `turtlebot_arm_packageName` benannt, auch wenn sie mehrheitlich nur Pincher-Arm-spezifisches beinhalten (wie z.B. das MoveIt-Package für Gazebo, siehe Unterabschnitt 4.1.5).

Um nun den einen oder anderen Arm auszuwählen, für den die Files geladen werden sollen, muss eine Linux-Umgebungsvariable auf entweder `pincher` oder `turtlebot` gesetzt werden. Damit dies nur ein einziges Mal gemacht werden muss, kann folgende Zeile dem Linux-File `~/.bashrc` hinzugefügt werden:

```
1 $ export TURTLEBOT_ARM1=pincher
```

Dies ist dieselbe Vorgehensweise wie mit den Catkin-Workspaces der ROS-Packages, welche beispielsweise mit dem Befehl

```
1 $ source ~/Documents/arbotix-devel/setup.bash
```

„gesourced“ werden müssen, damit sie von ROS gefunden werden können. Auch diese `source`-Befehle werden typischerweise im `.bashrc`-File hinzugefügt.

Tabelle 4.1: ROS-Packages des Pincher-Armes

Package	Inhalt/Zweck
turtlebot_arm	Standard-Main-Package des Catkin-Workspaces
bringup	Startet die nötigen Arbotix-Controller für den realen Pincher-Arm
control	Startet die nötigen Controller für den simulierten Pincher-Arm (neu hinzugefügt, siehe Unterabschnitt 4.1.4)
description	Beinhaltet die URDF-Files & CAD-Modelle des Pincher-Armes (und des Turtlebot-Armes), für den realen sowie den simulierten Pincher (siehe Unterabschnitt 4.1.3)
gazebo	Startet die Gazebo-Simulation des Pincher-Armes (neu hinzugefügt, siehe Unterabschnitt 4.1.2)
moveit_config	MoveIt-Konfiguration für den realen Turtlebot- & Pincher-Arm
moveit_config_4gazebo	MoveIt-Konfiguration für den simulierten Pincher-Arm (neu hinzugefügt, siehe Unterabschnitt 4.1.5)
moveit_demos	Beinhaltet verschiedene Test-Python-Skripte für MoveIt
block_manipulation	Benötigt für Pick & Place-Demo (nicht verwendet)
kinect_calibration	Benötigt für die Kalibrierung der Kinect-Kamera für die Pick & Place Demo (nicht verwendet)
object_manipulation	Beinhaltet Pick & Place-Demo (nicht verwendet)
ikfast_plugin	Beinhaltet den spezifisch für den Turtlebot- & Pincher-Arm zugeschnittene IK-Solver (wird in diesem Projekt jedoch nicht verwendet, da aufgrund von Problemen mit diesem Package ein Standard-IK-Solver verwendet wird)

4.1.2 Gazebo-Package

Das Gazebo-Package besteht nur aus zwei Files, dem Launch-File und dem World-File. Das Launch-File (Listing 4.1) ist dabei der Einstiegspunkt und wird von dem RL-Algorithmus (genauer: von dem OpenAI Gym environment) gestartet. Es startet die ganze Simulation, übersetzt das xarco-File des Pinchers in ein einzelnes URDF-File, lädt den Roboter in die Simulation und startet die Controller inkl. MoveIt-Konfigurationen, welche es ermöglichen, den Pincher in der Simulation auch über MoveIt zu bewegen.

Listing 4.1: Ausschnitt aus dem Launch-File des Gazebo-Packages

```

1  <launch>
2    ...
3    <arg name="arm_type" default="$(optenv TURTLEBOT_ARM1 turtlebot)" />
4
5    <include file="$(find gazebo_ros)/launch/empty_world.launch">
6      <arg name="world_name" value="$(find turtlebot_arm_gazebo)/worlds/
        turtlebot_arm.world"/>

```

```

7     ...
8   </include>
9
10  <!-- Load the URDF into the ROS Parameter Server -->
11  <param name="robot_description" command="$(find xacro)/xacro --
12    inorder '$(find turtlebot_arm_description)/urdf_4gazebo/${arg
13      arm_type}_arm.urdf.xacro'"/>
14
15  <!-- Run a python script to the send a service call to gazebo_ros to
16    spawn a URDF robot -->
17  <node name="urdf_spawner" pkg="gazebo_ros" type="spawn_model" respawn
18    ="false" output="screen"
19    args="-urdf -model ${arg arm_type} -param robot_description"/>
20
21  <!-- launch files -->
22  <!-- load controllers for the arm and the gripper -->
23  <include file="$(find turtlebot_arm_control)/launch/
24    turtlebot_arm_control.launch"/>
25  <!-- load moveit -->
26  <include file="$(find turtlebot_arm_moveit_config_4gazebo)/launch/
27    demo_planning.launch">
28
29  ...
30  </include>
31
32  </launch>

```

Das World-File definiert die eigentliche Simulationswelt und ist in Listing 4.2 zu sehen. Hier werden jedoch nur die beiden Standard-Objekte (eine unendlich grosse und unendlich dünne Grundplatte und eine globale Lichtquelle) eingefügt. Hier könnten zwar bereits schon beliebige andere Objekte mithilfe ihrer SDF-Files eingebunden und so der Simulation hinzugefügt werden, jedoch wird hier darauf bewusst verzichtet. Das Einfügen des Roboters geschieht durch einen separaten ROS-Node im Launch-File (Listing 4.1), und die restlichen Objekte wie die Kinect-Kamera oder den Quader werden durch das OpenAI Gym environment eingefügt (siehe Abschnitt 5.1), was eine viel grössere Flexibilität zur Folge hat. Im World-File werden aber gewisse Physik-Parameter der Simulation definiert.

Listing 4.2: Ausschnitt aus dem World-File des Gazebo-Packages

```

1  <?xml version="1.0" ?>
2  <sdf version="1.4">
3    <world name="default">
4      <include>
5        <uri>model://ground_plane</uri>
6      </include>
7      <include>
8        <uri>model://sun</uri>
9      </include>
10     <!-- see http://gazebosim.org/tutorials?tut=modifying_world -->
11     <physics type="ode">
12       <gravity>0.000000 0.000000 -9.810000</gravity>
13       <ode>
14         <solver>
15           <type>quick</type>
16           <iters>50</iters>
17           <precon_iters>0</precon_iters>
18           <sor>1.300000</sor>
19         </solver>
20         <constraints>
21           <cfm>0.000000</cfm>
22           <erp>0.200000</erp>
23           <contact_max_correcting_vel>100.000000</
24             contact_max_correcting_vel>

```

```

24      <contact_surface_layer>0.001000</contact_surface_layer>
25    </constraints>
26  </ode>
27  <real_time_update_rate>0.000000</real_time_update_rate>
28  <!-- real_time_update_rate = 0 means as fast as possible -->
29  <max_step_size>0.001000</max_step_size>
30  </physics>
31  </world>
32 </sdf>

```

4.1.3 URDF-Files

Im Launch-File des Gazebo-Packages (Listing 4.1) wird das Haupt-URDF-File des Pincher-Armes `pincher_arm.urdf.xacro` aufgerufen (sofern die Umgebungsvariable entsprechend gesetzt ist, siehe Unterabschnitt 4.1.1). Dieses befindet sich in dem `description`-Package. Da Gazebo für gewisse Dinge wie das Bewegen des Roboters in der Simulation aber noch zusätzliche Informationen (Gazebo-Tags, siehe [31]) benötigt, wurde ein neuer Unterordner im `description`-Package `urdf_4gazebo` angelegt, worin sich die für die Gazebo-Simulation angepassten URDF-Files befinden.

Im `pincher_arm.urdf.xacro`-File wird unter anderem die Grundplatte des Roboter-Armes in Bezug auf das `/world`-Koordinatensystem der Gazebo-Simulation fixiert, damit der Pincher in der Simulation nicht zu Boden fällt.

Listing 4.3: Ausschnitt aus dem Haupt-URDF-File des Pinchers

```

1  <?xml version="1.0"?>
2  <!-- Describe URDF for PhantomX Pincher Arm -->
3  <robot name="turtlebot_arm" xmlns:xacro="http://ros.org/wiki/xacro">
4
5      <!-- Pincher arm is same as Turtlebot -->
6      <xacro:include filename="$(find turtlebot_arm_description)/
7          urdf_4gazebo/turtlebot_arm.xacro"/>
8
9      <!-- As we do not have here a turtlebot base, add a world link as
10         its location reference -->
11      <link name="world"/>
12
13      <!-- Gazebo configuration for arm and cannon servos control -->
14  <gazebo>
15      <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.
16          so">
17          <robotNamespace>/turtlebot_arm</robotNamespace>
18          <robotSimType>gazebo_ros_control/DefaultRobotHWSim</robotSimType>
19      </plugin>
20  </gazebo>
21
22  <xacro:macro name="servo_transmission" params="name">
23      <transmission name="${name}_trans">
24          <type>transmission_interface/SimpleTransmission</type>
25          <joint name="${name}_joint">
26              <hardwareInterface>hardware_interface/PositionJointInterface</
27                  hardwareInterface>
28          </joint>
29          <actuator name="${name}_motor">
30              <hardwareInterface>hardware_interface/PositionJointInterface</
31                  hardwareInterface>
32              <mechanicalReduction>1</mechanicalReduction>
33          </actuator>
34      </transmission>

```

```

30  </xacro:macro>
31
32  <servo_transmission name="arm_shoulder_pan"/>
33  <servo_transmission name="arm_shoulder_lift"/>
34  <servo_transmission name="arm_elbow_flex"/>
35  <servo_transmission name="arm_wrist_flex"/>
36
37  <!-- Turtlebot arm macro -->
38  <turtlebot_arm parent="world" color="Gray" gripper_color="Gray"
   pincher_gripper="true" turtlebot_gripper="false">
39  <!-- Place the "floating" arm at the location according to the
      physical setup -->
40  <origin xyz="0 0 0.083"/> <!-- height of the PhantomX Pincher
      base = ca. 58mm + height of table plane = ca. 25mm -->
41  </turtlebot_arm>
42 </robot>

```

4.1.3.1 Probleme mit dem Greifer

Leider sind die Gazebo-spezifischen Tags nicht das Einzige, was bei den URDF-Files des Pinchers noch verändert werden muss. So war zum Zeitpunkt des Projektes der Parallel-Greifer des Pinchers nicht korrekt in den URDF-Files abgebildet. Beim Ansteuern des Greifers in der Simulation wurde nur eine der beiden Backen bewegt. Selbst beim blossen Bewegen des Armes flog die bewegliche Backe umher und befand sich so in eigentlich unmöglichen Positionen, wie in Abbildung 4.1 zu sehen ist.

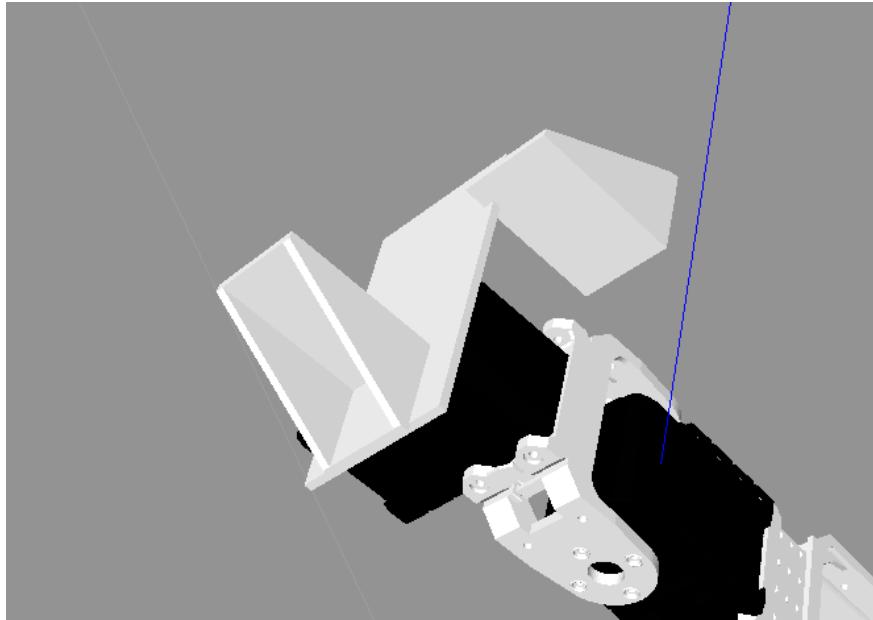


Abbildung 4.1: Von dem Roboter losgelöste Greifer-Backe in Gazebo

Auch ein Test mit einem Greifen/Festhalten des Quaders in der Simulation brachte Probleme hervor. So knickte die bewegliche Backe des Greifers beim Zusammenfahren des Greifers ein, und der Quader konnte nicht korrekt (geschweige denn realistisch) gegriffen werden, wie dies in Abbildung 4.2 zu sehen ist. Generell scheint das Greifen in Gazebo ein ziemliches Problem darzustellen, wie auch andere Benutzer feststellen mussten [32].

Zudem machte der Greifer auch Probleme mit den MoveIt-Konfigurationen und später auch im OpenAI Gym environment (siehe Kapitel 5). Dies lag an den deplatzierten

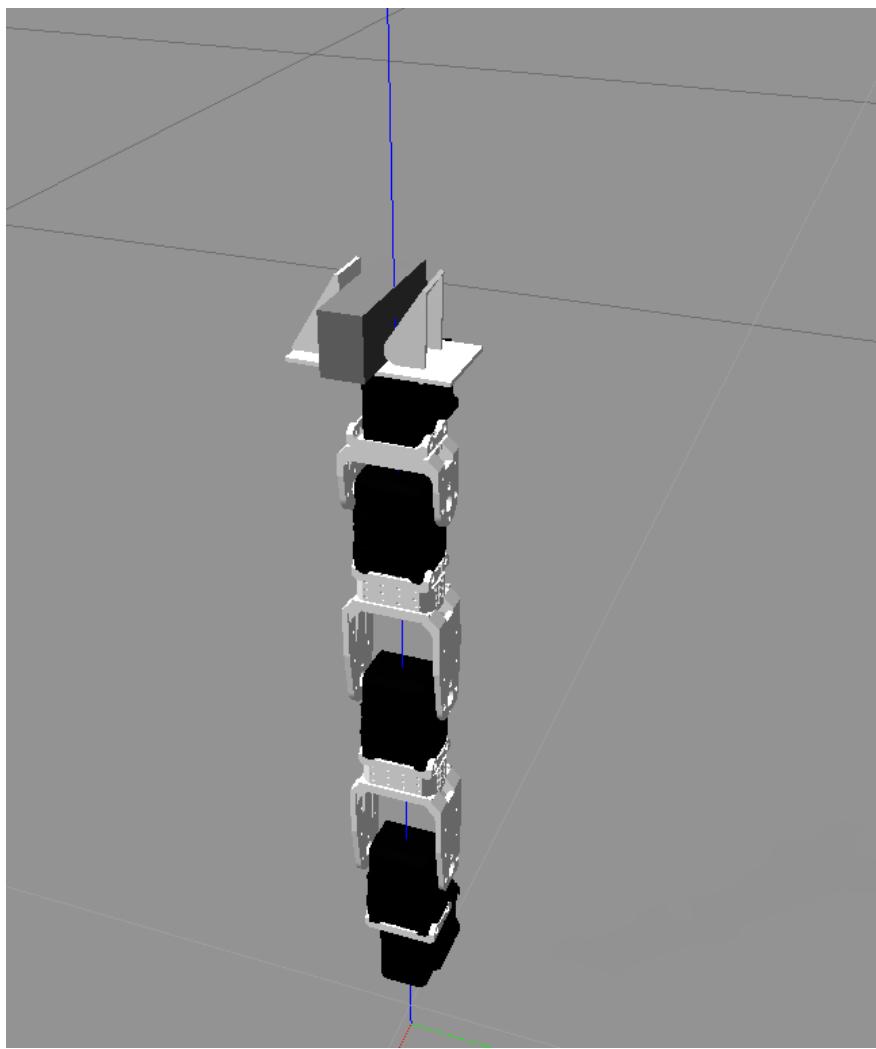


Abbildung 4.2: Probleme mit dem Greifen in Gazebo

Links, welche nicht dem Modell des Pincher-Armes entsprechen, wie dies beispielsweise in Abbildung 4.3 gut sichtbar ist.

Aus all diesen Gründen muss deshalb der ganze Greifer in der Simulation fixiert werden, damit er sich nicht mehr (relativ zu seiner Halterung) bewegen kann. Dazu sind viele Änderungen in den URDF-Files nötig, auch damit die Links der Greifer-Partie an den korrekten Orten platziert waren. Ein Pull-Request (Verbesserungs-Vorschlag) aus dem Github-Repository des Pincher-Armes (siehe [33]) dient dabei als Hilfe. Alles aus dem Pull-Request kann jedoch nicht übernommen werden, da es ansonsten nicht mehr möglich ist, den Arm in Gazebo zu bewegen. Auf eine Ausführung der nötigen Änderungen wird an dieser Stelle aber verzichtet.

Wie der Greifer nach der Bearbeitung aussieht, ist in Abbildung 4.4 abgebildet.

4.1.4 Control-Package

Das Control-Package ist nötig, damit MoveIt den Arm in der Gazebo-Simulation ansteuern kann. Das Package beinhaltet ein Launch-File und ein *yaml*-Konfigurations-File. *yaml* ist ein Dateiformat, welches einem JSON ähnelt. Darin wird u.a. der `arm_controller` definiert, welcher der ganzen kinematischen Kette des Armes einen Joint-

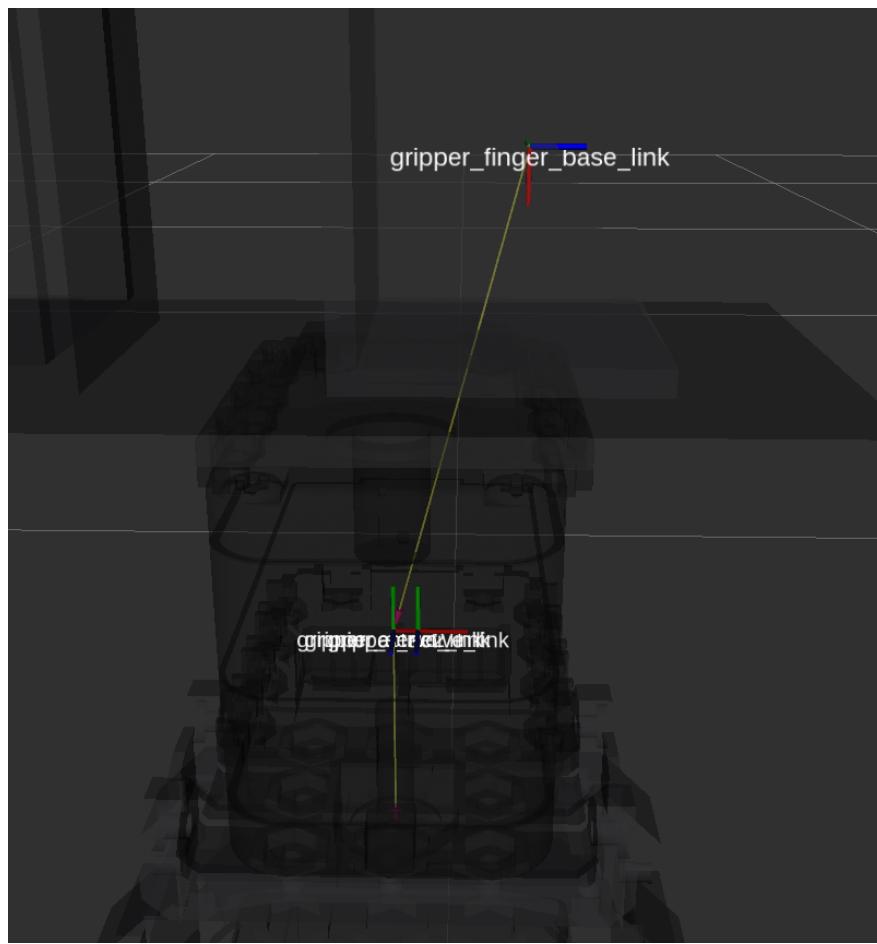


Abbildung 4.3: Falsch platzierte Links des Pincher-Greifers (visualisiert in Rviz)

TrajectoryController zuweist. MoveIt benötigt dann diesen arm_controller, um den Arm (in der Simulation) zu bewegen.

Im Launch-File des Control-Package werden u.a. die Controller, welche im yaml-File definiert sind, geladen.

Listing 4.4: Ausschnitt aus dem Launch-File des Control-Packages

```

1 <launch>
2   <!-- Load joint controller configurations from YAML file to parameter
      server -->
3   <rosparam file="$(find turtlebot_arm_control)/config/
      turtlebot_arm_control.yaml" command="load"/>
4
5   <!-- load the controllers -->
6   <node name="controller_spawner" pkg="controller_manager" type="
      spawner" respawn="false"
      output="screen" ns="/turtlebot_arm" args=" joint_state_controller
      arm_controller
      "/>
7
8   <!-- convert joint states to TF transforms for rviz, etc -->
9   <node name="robot_state_publisher" pkg="robot_state_publisher" type="
      robot_state_publisher"
      respawn="false" output="screen">
10    <remap from="/joint_states" to="/turtlebot_arm/joint_states" />
11  </node>
12
13 </launch>
```

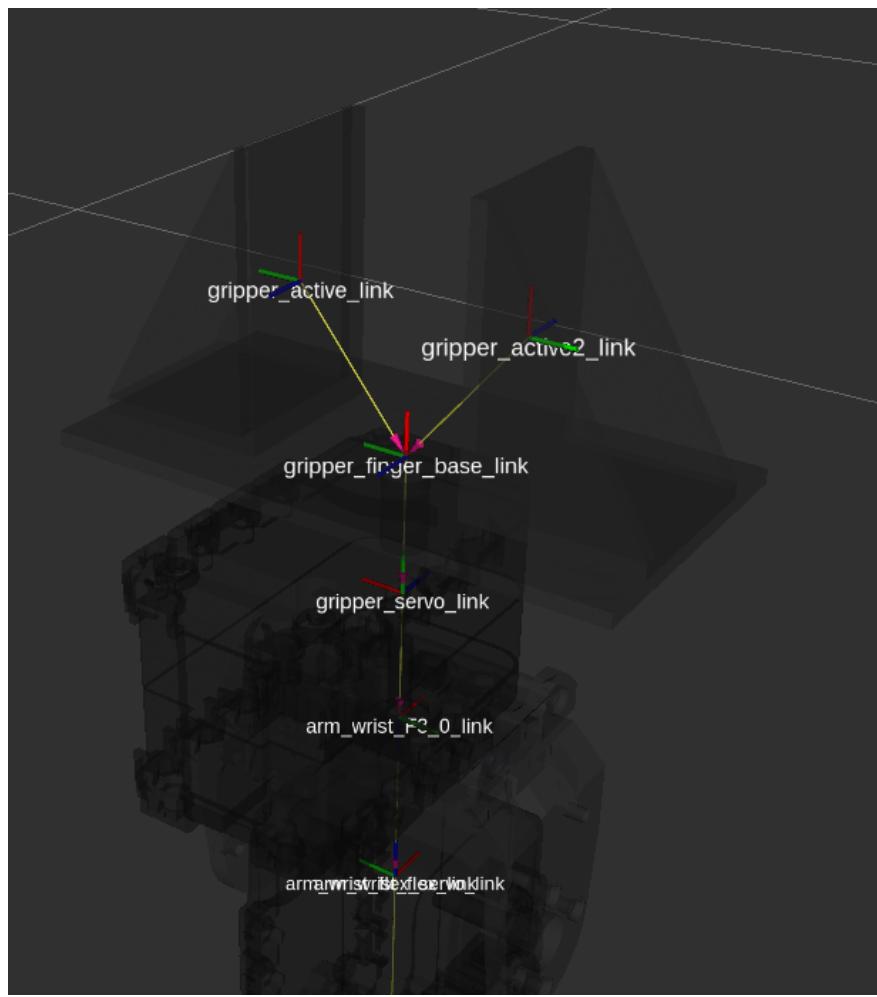


Abbildung 4.4: Links des Pincher-Greifers nach der Überarbeitung (visualisiert in Rviz)

4.1.5 MoveIt-Package

Auch das MoveIt-Package muss überarbeitet werden, damit der Roboter-Arm in Gazebo über MoveIt gesteuert werden kann. Deshalb gibt es auch ein separates MoveIt-Package für die Gazebo-Simulation, `turtlebot_arm_moveit_config_gazebo`. In einem MoveIt-Package befinden sich zahlreiche Launch- und yaml-Konfigurations-Files. Die meisten davon müssen aber nicht für jeden Roboter-Arm selbst geschrieben werden, sondern werden durch ein separates Hilfsprogramm erstellt. Dieses kann mit dem Terminal-Befehl

```
1 $ rosrun moveit_setup_assistant setup_assistant.launch
```

gestartet werden. Danach erscheint eine GUI des Hilfsprogrammes (*MoveIt Setup Assistant*, siehe Abbildung 4.5), mit welchem die verschiedenen Konfigurationen eingestellt und definiert werden können. Beispielsweise muss zuerst die *robot description* geladen werden, d.h. das (übersetzte) URDF-File des gewünschten Roboter-Armes. Anschließend können diverse Dinge wie die kinematische Kette des Armes und eventuelle weitere Dinge wie Greifer-Controller definiert werden.

Das durch den MoveIt Setup Assistant erstellte MoveIt-Package muss noch erweitert und angepasst werden, damit die Verbindung zu der Gazebo-Simulation besteht. Da dies aber relativ viele und detaillierte Änderungen sind, wird hier darauf verzichtet, genauer darauf einzugehen.

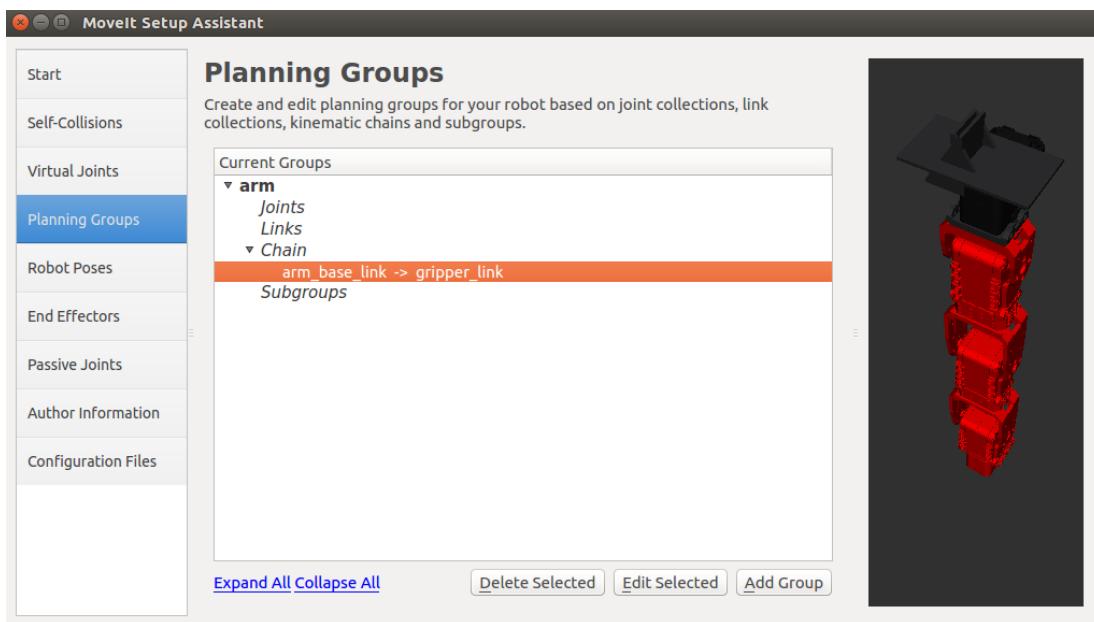


Abbildung 4.5: Screenshot des MoveIt Setup Assistant

4.1.6 Ansteuerung des Roboters über MoveIt Python API

Sind alle Packages korrekt aufgesetzt und konfiguriert, kann die Simulation gestartet werden, beispielsweise aus dem OpenAI Gym environment heraus. Dann kann der Pincher-Arm über eine von MoveIt zur Verfügung gestellter API in Gazebo bewegt werden. Dies kann ein C++ oder ein Python-Programm sein. Da der ganze RL-Algorithmus in OpenAI Gym und dadurch ebenfalls in Python geschrieben ist, wird in Listing 4.5 kurz die Python-API von MoveIt aufgezeigt.

Listing 4.5: Beispiel-Programm um über die MoveIt Python API den Pincher-Arm zu einer Position zu bewegen

```

1 # ...
2
3 moveit_commander.roscpp_initialize(sys.argv)
4 rospy.init_node("moveit_python_API_test", anonymous=True)
5 robot = moveit_commander.RobotCommander()
6 scene = moveit_commander.PlanningSceneInterface()
7 group_arm = moveit_commander.MoveGroupCommander("arm")
8
9 REFERENCE_FRAME = "world"
10
11 # Move the robot arm
12 x = 0.11
13 y = 0.0
14 z = 0.028
15 roll = 0
16 pitch = np.pi/2
17 yaw = 0
18
19 pose = self.createPose(x, y, z, roll, pitch, yaw)
20 success = self.moveArmToPose(pose)
21
22 def createPose(self, x=0.0, y=0.0, z=0.0, roll=0.0, pitch=0.0, yaw=0.0)
23     :
24     pose = Pose()
25     pose.position.x = x
26     pose.position.y = y

```

```

26     pose.position.z = z
27     quaternion = tf.transformations.quaternion_from_euler(roll, pitch,
28                 yaw)
28     pose.orientation.x = quaternion[0]
29     pose.orientation.y = quaternion[1]
30     pose.orientation.z = quaternion[2]
31     pose.orientation.w = quaternion[3]
32     return pose
33 # createPose
34
35 def moveArmToPose(self, targetPose, execute=True):
36     self.group_arm.clear_pose_targets()
37     self.group_arm.set_pose_target(targetPose)
38     planArm = self.group_arm.plan()
39     if(execute):
40         success = self.group_arm.go(wait=True)
41     return success
42 # moveArmToPose

```

4.2 Simulation Kinect-Kamera

Neben der Simulation des Pincher-Armes wird auch eine Simulation des verwendeten Sensors benötigt. Für viele verschiedene Sensoren existieren bereits Gazebo-Files (SDF) & -Plugins, wie beispielsweise Laserscan-Sensoren oder diverse Kameras. So kann auch die Kinect-Kamera relativ einfach in die Simulation eingebunden und in der Welt platziert werden. Dazu benötigt werden die entsprechende SDF-Files und die Installation des Kinect-Sensor-Plugins. Die Anleitung dazu ist unter [34] zu finden. Die Kinect nimmt dann RGB- und Tiefenbilder der Simulationswelt auf und stellt ihre Daten über ROS-topics zur Verfügung. Diese werden schliesslich von dem OpenAI Gym environment über die Python ROS API empfangen (mehr dazu im Abschnitt 5.5). Folgende Zeile empfängt eine ROS-message vom Typ Image [35], welches unter dem ROS-topic /camera/depth/image_raw von der Kinect-Kamera publiziert wird:

```

1 data = rospy.wait_for_message("/camera/depth/image_raw", Image, timeout
=5)

```

5 OpenAI Gym environment

Nachdem die Simulation des Roboters und der Kinect funktioniert, ist ein OpenAI Gym environment nötig, welche die Simulation steuert und den Reward berechnet. In einem OpenAI Gym environment wird die Umgebung beschrieben, mit welcher der RL-Agent interagieren soll. Dabei muss jedes environment eine Unterklasse von der in OpenAI Gym beschriebenen Klasse Env sein und somit bestimmte Methoden implementieren. Dies sind beispielsweise `reset()` oder `step()`. Die step-Methode wird vom RL-Agenten einmal pro Zeitschritt aufgerufen. Ihr wird eine Action mitgegeben. Das environment führt dann die erhaltene Action aus, und gibt neben dem State (observation) und dem Reward u.a. auch noch einen boolean zurück, welcher indiziert, ob es sich um einen Terminal-State handelt. Das environment wird vom Hauptprogramm des RL-Algorithmus erzeugt, instandgesetzt und für das Training verwendet. Ein Beispiel eines solchen Programms ist in Listing 5.1 zu sehen. Dort findet jedoch noch kein Training statt, da bloss jeweils eine zufällige Action ausgewählt wird.

Listing 5.1: Beispiel-Programm in OpenAI Gym

```

1 import gym
2 env = gym.make('CartPole-v0')
3 for i_episode in range(20):
4     observation = env.reset()
5     for t in range(100):
6         env.render()
7         action = env.action_space.sample()
8         observation, reward, done, info = env.step(action)
9         if done:
10             print("Episode finished after {} timesteps".format(t+1))
11             break

```

Es ist auch möglich, eigene environments zu programmieren. Da in diesem Projekt eine Gazebo-Simulation mit dem environment gestartet werden muss, kann als Vorlage das gym-gazebo [36] genommen werden. Dort wird unter anderem eine Oberklasse für alle Gazebo-environments GazeboEnv definiert, welche beispielsweise den Gazebo-Server und den roscore startet. In den folgenden Abschnitten wird das für dieses Projekt erstellte environment beschrieben. Dieses wird für das Training mit der Gazebo-Simulation verwendet.

Möchte man das reale Setup (siehe Kapitel 6) verwenden, ist ein anderes environment nötig, da dieses beispielsweise kein Gazebo braucht. Die Ansteuerung des Pinchers und das Auslesen der Tiefenbilder der Kinect-Kamera sind aber bei beiden environments nahezu identisch, weshalb in diesem Kapitel nur auf das environment für die Simulation eingegangen wird.

5.1 Aufbau der Simulation

Wenn das environment erzeugt wird (mit `gym.make(env_name)`), wird zuerst in der Oberklasse GazeboEnv der roscore und der Gazebo-Server gestartet. Danach wird ein ROS-Node erzeugt, welcher das angegebene Launch-File startet. Dieses ist das Launch-File des Gazebo-Packages vom Unterabschnitt 4.1.2. Dabei wird der Pincher in die Gazebo-Simulation geladen und MoveIt mit den nötigen Controllern gestartet.

Danach werden im environment über die Python APIs von ROS und MoveIt verschiedenste Dinge initialisiert, wie beispielsweise einen Service, welcher die Gazebo-Simulation pausieren und wieder weiterlaufen lassen kann. In der init-Methode (Klassen-Konstruktor) werden zudem auch die restlichen nötigen Objekte in die Simulation einge-fügt, wie beispielsweise die Kinect oder den Quader. Die init-Methode sowie die Methode für das Einfügen eines Objektes in die Simulation sind in Listing 5.2 gezeigt.

Listing 5.2: init-Methode des Pincher-environments

```

1  def __init__(self):
2      # Launch the simulation with the given launch file name
3      # launch file is in gym-gazebo/gym_gazebo/env/assets/launch/
4      gazebo_env.GazeboEnv.__init__(self, "GazeboSmartBotPincherKinect_v0"
5          .launch")
6      self.link_state_pub = rospy.Publisher("/gazebo/set_link_state",
7          LinkState, queue_size=5)
8      self.unpause_proxy = rospy.ServiceProxy("/gazebo/unpause_physics",
9          Empty)
10     self.pause_proxy = rospy.ServiceProxy("/gazebo/pause_physics",
11         Empty)
12     self.reset_proxy = rospy.ServiceProxy("/gazebo/reset_simulation",
13         Empty)
14
15     # (define observation & action space incl. boundaries)
16
17     # setting up the scene
18     print("inserting table plane")
19     self.insertObject(0.4, 0, 0.025/2, "path/to/tablePlane.sdf", "tablePlane")
20
21     print("inserting kinect sensor")
22     self.insertObject(0.03, 0, 0.5, "path/to/kinect.sdf", "kinect_ros",
23         roll=0, pitch=np.pi/2, yaw=0)
24
25     print("inserting object to pick up")
26     self.insertObject(0.1, 0, 0.1, "path/to/objectToPickUp.sdf", "objectToPickUp")
27
28     # (initializing MoveIt variables)
29
30     # __init__
31
32     def insertObject(self, x, y, z, path_to_sdf, name, roll=0.0, pitch=0.0,
33         yaw=0.0, namespace="world"):
34         initial_pose = self.createPose(x, y, z, roll, pitch, yaw)
35         f = open(path_to_sdf, "r")
36         sdff = f.read()
37         rospy.wait_for_service("gazebo/spawn_sdf_model")
38         spawn_model_prox = rospy.ServiceProxy("gazebo/spawn_sdf_model",
39             SpawnModel)
40         spawn_model_prox(name, sdff, "", initial_pose, namespace)
41
42     # insertObject

```

5.2 Zufällige Platzierung des Quaders

Ist das environment einmal initialisiert, wird es (wie vor jeder neuen Episode) in den Anfangszustand zurückgesetzt. Dies geschieht über die reset-Methode, welche in Listing 5.3 zu sehen ist. Darin wird nicht nur die Gazebo-Simulation zurückgesetzt, sondern es wird auch (alle paar Episoden) der Quader neu platziert. Dabei werden zufällig die x-, y- und

z-Koordinaten des Quaders sowie seine Drehung um die z-Achse (yaw-Winkel) gesetzt. Die roll- und pitch-Winkel sind stets auf 0 fixiert, d.h. der Quader liegt immer hochkant dort. Dies wurde aufgrund der Reward-Berechnung (siehe Abschnitt 5.4) so gewählt, damit diese etwas einfacher ausfällt.

Listing 5.3: reset-Methode des Pincher-environments

```

1  def reset(self):
2      """ Resets the state of the environment and returns an initial
       observation."""
3      self.reset_simulation()
4
5      # Unpause simulation to set position of ObjectToPickUp & make
       observation
6      self.unpause_simulation()
7      if(self.resetCount % self.randomPositionAtResetFrequency == 0):
8          x = random.uniform(0.06, 0.22)
9          y = random.uniform(-0.2, 0.2)
10         z = random.uniform(0.05, 0.1)
11         roll = 0
12         pitch = 0
13         yaw = random.uniform(0, np.pi)
14         self.currentPoseOfObject =
               self.createPose(x,y,z,roll,pitch,yaw)
15         self.setPoseOfObjectToPickUp(self.currentPoseOfObject)
16      self.resetCount += 1
17
18      # get depth image from kinect camera
19      image = self.read_kinect_depth_image(saveImage=True)
20      self.pause_simulation()
21
22      self.state = image
23      return self.state
24
25 # reset

```

5.3 step-Methode

In der step-Methode wird die vom RL-Agenten vorgeschlagene Action ausgeführt. Die Methode ist in Listing 5.4 zu sehen. Darin wird versucht, den Pincher-Arm über die MoveIt Python API an die gewünschte Position zu bringen. Ist die Position für den Pincher jedoch nicht erreichbar (d.h. der IK-Solver hat keine Lösung gefunden), wird ein tiefer Reward zurückgegeben. Als nächster State wird ein neues Tiefenbild mit der Kinect-Kamera aufgenommen. Dieses wird sehr ähnlich sein wie jenes, welches der RL-Agent einen Zeitschritt zuvor erhalten hat, da sich in der Simulation ja nichts geändert hat. Würde dem RL-Agenten jedoch das exakt gleiche Bild (d.h. der exakt gleiche State) zurückgegeben, resultierte dies in einer sehr schlechten Exploration, wie dies Experimente in diesem Projekt ergeben haben. Eine schlechte Exploration bedeutet, dass der Agent in allen Zeitschritten stets die beinahe identische Position auswählt.

Kann der Pincher-Arm jedoch an die gewünschte Position gebracht werden, wird der Reward berechnet (siehe Abschnitt 5.4) und anschliessend der Arm wieder in die Start-Position gebracht. Danach wird ein neues Bild der Kinect-Kamera aufgenommen und als nächster State dem RL-Agenten zurückgegeben.

Listing 5.4: step-Methode des Pincher-environments

```

1  def step(self, action):
2      self.unpause_simulation()
3
4      # reset position of object in case it got moved
5      self.setPoseOfObjectToPickUp(self.currentPoseOfObject)
6
7      # get the position of the ObjectToPickUp
8      pickup_pose_old, gripper_right_position, gripper_left_position =
9          self.getPoseOfObjectAndGripper()
10
11     # Move the robot arm
12     x = action[0].astype(np.float64)
13     y = action[1].astype(np.float64)
14     z = action[2].astype(np.float64)
15     roll = action[3].astype(np.float64)
16     pitch = action[4].astype(np.float64)
17     yaw = action[5].astype(np.float64)
18     pose = self.createPose(x, y, z, roll, pitch, yaw)
19     success = self.moveArmToPose(pose)
20     self.printMovingArmSuccess(success, printSuccess=True)
21
22     if(not success):
23         # unreachable position has been selected by the RL algorithm
24         reward = self.rewardUnreachablePosition
25         done = False
26         info = {"couldMoveArm" : False}
27         # take a new picture (should be very similar like the one
28             # before) but don't return previously captured image (= same
29             # state as before), as this results in very bad exploration
30         self.state = self.read_kinect_depth_image(saveImage=True)
31         return self.state, reward, done, info
32
33     pickup_pose, gripper_right_position, gripper_left_position = self.
34         getPoseOfObjectAndGripper()
35
36     # determine if gripper could grasp the ObjectToPickUp
37     reward = self.calculateReward(pickup_pose_old, pickup_pose,
38         gripper_right_position, gripper_left_position)
39
40     # return arm to home position
41     success = self.moveArmToHomePosition()
42     self.printMovingArmSuccess(success, printFailure=True)
43
44     # set the ObjectToPickUp where it was before (at the beginning of
45         # the step() function)
46     self.setPoseOfObjectToPickUp(pickup_pose_old)
47
48     # get depth image from kinect camera (= next state)
49     image = self.read_kinect_depth_image(saveImage=True)
50
51     self.pause_simulation()
52     self.state = image
53     done = False
54     info = {"couldMoveArm" : True}
55
56     return self.state, reward, done, info
57     # step

```

5.4 Reward-Berechnung

Ein sehr wichtiger Teil des environments ist die Berechnung des Rewards, da mit dem sogenannten *reward shaping* ein RL-Algorithmus stehen und fallen kann. Nach dem

Anfahren der durch den RL-Agenten vorgeschlagenen Position könnte beispielsweise der Greifer zugemacht und mit dem Pincher-Arm an eine definierte Position gefahren werden. Danach könnte einfach über die ROS Python API geprüft werden, ob sich der Quader an der definierten Position befindet. Wenn ja, konnte der Arm den Quader greifen und der Reward würde höher ausfallen als wenn nicht. Diese einfache Reward-Berechnung ist in diesem environment aber leider nicht möglich, da wie bereits in Unterunterabschnitt 4.1.3.1 erwähnt das Greifen in Gazebo sehr unrealistisch ist und der Greifer des Pincher-Armes aufgrund diverser Probleme unbeweglich gemacht werden muss. Aus diesen Gründen muss der Reward anders berechnet werden, wie folgend beschrieben wird.

5.4.1 Binärer Reward

Da der Greifer des simulierten Roboters starr ist und sich seine Backen nicht bewegen können, wird ein hoher Reward zurückgegeben, wenn der Greifer in einer Position ist, in der er den Quader greifen *könnte*, wenn er seine Backen schliessen *würde*. Dies wird wie folgt berechnet: Nachdem der Pincher-Arm die vom RL-Agenten vorgeschlagene Position erreicht hat, wird die momentane Position & Orientierung (aka Pose) des Quaders über die ROS Python API geholt, zusammen mit den Positionen der beiden Greifer-Backen. Wenn sich nun die eine Backe in einem bestimmten Gebiet (*bounding box*) rechts des Quaders und die andere Backe in einem bestimmten Gebiet links des Quaders befinden, wäre der Greifer in der Lage, den Quader aufzuheben. Die bounding box wird dabei mithilfe von Einheitsvektoren berechnet. Eine Skizze der Situation ist in Abbildung 5.1 abgebildet.

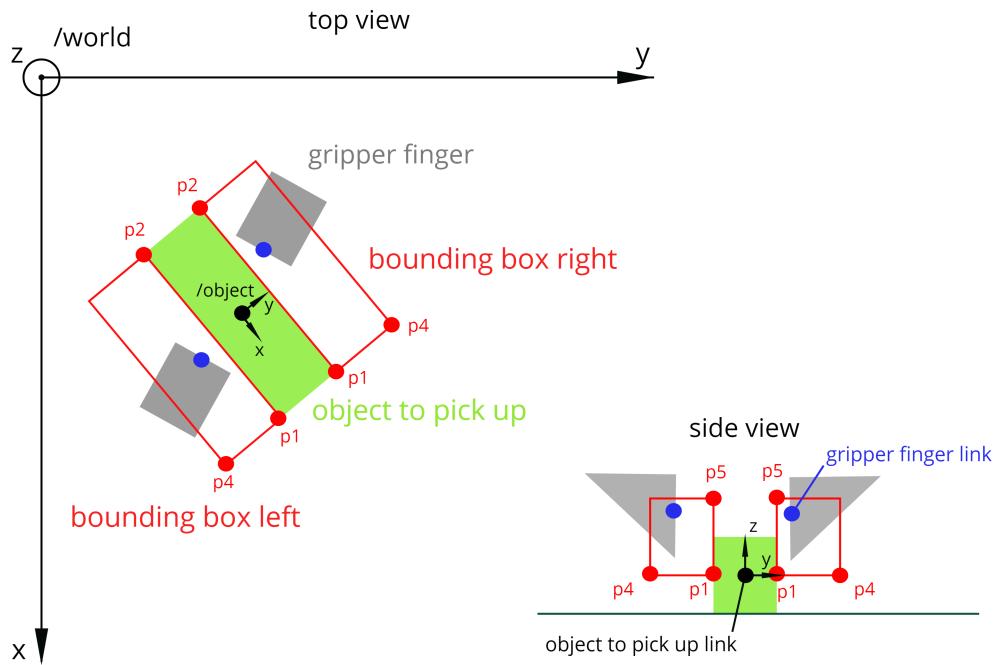


Abbildung 5.1: Reward-Berechnung im Pincher-Environment. Ein Greifen wird dann als erfolgreich gewertet, wenn die beiden `gripper_finger_links` in den gegenüberliegenden bounding boxen befinden. Die Bezeichnung der Eckpunkte einer bounding box (p1, p2, p4, p5) entspricht derjenigen aus dem Code und [37].

Wenn die gewünschte Position von dem Pincher-Arm nicht erreicht werden kann, wird ein tiefer Reward zurückgegeben. Betrachtet man nur die Fälle, in welchen die gewünschte Position erreicht werden konnte, erhält der RL-Agent entweder einen hohen oder einen tiefen – also binären – Reward. Ein binärer Reward hat aber zur Folge, dass in den meisten Fällen der RL-Agent jedoch keinen Erfolg erfahren wird. Dies führt bei den meisten der RL-Algorithmen dazu, dass das Training entweder sehr lange dauert oder der Agent gar nicht zu einer akzeptablen Performance konvergiert. Ein relativ neuer Algorithmus, welcher mit sehr spärlichen Rewards umgehen kann, wäre der HER-Algorithmus [38], welcher in diesem Projekt aus Zeitgründen aber nicht weiter verfolgt werden konnte (siehe Kapitel 9).

5.4.2 Kontinuierlicher Reward

Deshalb ist neben dem binären Reward noch eine leicht andere Variante der Reward-Berechnung implementiert. Bei dieser wird der Reward aus dem Kehrwert der Distanz des Greifers zu der bounding box des Quaders gebildet:

$$reward_{cont} = \begin{cases} 5 * \frac{1}{\text{distance}}, & \text{if } graspSuccess \\ \frac{1}{\text{distance}}, & \text{otherwise} \end{cases} \quad (5.1)$$

Zusätzlich wird, falls die Greifer-Backen in der Lage wären, den Quader aufzuheben, der Reward noch mit 5 multipliziert, um diese wünschenswerte Situation noch stärker zu belohnen. Durch eine Klassenvariable kann zwischen der binären und der kontinuierlichen Reward-Berechnung gewechselt werden.

Die Methode, welche den Reward anhand der Pose des Quaders und der Positionen der Greifer-Backen berechnet, ist in Listing 5.5 zu sehen.

Listing 5.5: Auszug aus der Reward-Berechnung im Pincher-environment

```

1 def calculateReward(self, pickup_pose_old, pickup_pose,
2     gripper_right_position, gripper_left_position):
3     pickup_position_old = np.matrix([[pickup_pose_old.position.x],
4         [pickup_pose_old.position.y], [pickup_pose_old.position.z]])
5
6     pickup_position = np.matrix([[pickup_pose.position.x],
7         [pickup_pose.position.y], [pickup_pose.position.z]])
8     # e.g. pickup_position = [[0.1], [0.0], [0.04]]
9
10    # (check if the gripper has crashed into the ObjectToPickUp)
11
12    # check if gripper is in the correct position in order to grasp the
13    # object
14    # dimensions of the ObjectToPickUp & the gripper (see the
15    # corresponding sdf/urdf files)
16    pickup_xdim = 0.07
17    pickup_ydim = 0.02
18    pickup_zdim = 0.03
19    gripper_width = 0.032 # between the fingers
20
21    # calculating the unit vectors (described in the /world coordinate
22    # system) of the objectToPickUp::link
23    ex, ey, ez = self.getUnitVectorsFromOrientation(pickup_pose.
24        orientation)
25
26    # corners of bounding box where gripper_right_position has to be
27    p1 = pickup_position + pickup_xdim/2 * ex + pickup_ydim/2 * ey

```

```

21      # e.g. p1 = [[0.135] [0.01] [0.04]]
22      p2 = p1 - pickup_xdim * ex
23      # e.g. p2 = [[0.065] [0.01] [0.04]]
24      p4 = p1 + ey * (pickup_ydim/2 + gripper_width - pickup_ydim)
25      # e.g. p3 = [[0.135] [0.032] [0.04]]
26      p5 = p1 + ez * pickup_zdim
27      # e.g. p4 = [[0.135] [0.01] [0.07]]
28      # the vectors of the bounding box where gripper_right_position has
29      # to be
30      u = np.transpose(p2 - p1)
31      v = np.transpose(p4 - p1)
32      w = np.transpose(p5 - p1)

33      # (same for the bounding box to the left of the ObjectToPickUp)
34
35      graspSuccess = False
36      # check if right gripper is on the right and left gripper is on the
37      # left of the ObjectToPickUp
38      gripperRightIsRight = self.isPositionInCuboid(
39          gripper_right_position, p1, p2, p4, p5, u, v, w)
40      gripperLeftIsLeft = self.isPositionInCuboid(gripper_left_position,
41          p1_left, p2_left, p4_left, p5_left, u_left, v_left, w_left)
42      # check if right gripper is on the left and left gripper is on the
43      # right of the ObjectToPickUp
44      gripperLeftIsRight = self.isPositionInCuboid(gripper_left_position,
45          p1, p2, p4, p5, u, v, w)
46      gripperRightIsLeft = self.isPositionInCuboid(gripper_right_position
47          , p1_left, p2_left, p4_left, p5_left, u_left, v_left, w_left)
48      # if one of the two scenarios is true, the grasping would be
49      # successful
50      if((gripperRightIsRight and gripperLeftIsLeft) or (
51          gripperLeftIsRight and gripperRightIsLeft)):
52          print("grasping would be successful!")
53          graspSuccess = True
54      else:
55          graspSuccess = False

56      if(self.binaryReward):
57          if(graspSuccess):
58              return self.rewardSuccess
59          else:
60              return self.rewardFailure
61      else:
62          # calculate reward according to the distance from the gripper
63          # to the middle of the bounding box
64          # pM = middle of the box where gripper_right_position has to be
65          pM = p1 + 0.5 * np.transpose(u) + 0.5 * np.transpose(v) + 0.5 *
66              np.transpose(w)
67          # e.g. pM = matrix([[0.1], [0.021], [0.055]])
68          distance = np.linalg.norm(pM - gripper_right_position)

69          # invert the distance, because smaller distance == closer to
70          # the goal == more reward
71          reward = 1.0 / distance
72          # scale the reward if gripper is in the bounding box
73          if(graspSuccess):
74              reward = 5 * reward
75          return reward
76      # if
77      # calculateReward

```

5.4.3 Position von Quader & Greifer-Backen erhalten

Die Pose des Quaders über die ROS Python API herauszufinden stellt kein Problem dar, da die Positionen & Orientierungen vieler Links von Objekten in der Simulation über das ROS-topic `/gazebo/link_states` publiziert werden – unter anderem auch die Pose des Quaders (`ObjectToPickUp`). Leider werden die Positionen der beiden Links der Greifer-Backen nicht über dieses Topic publiziert, weshalb diese Positionen mit einem Workaround herausgefunden werden müssen.

Es können zwar nicht die Positionen der Greifer-Backen durch das ROS-topic empfangen werden, dafür aber u.a. die Pose eines Links, welcher sich nicht relativ zu den Greifer-Backen bewegt. Dies ist der `arm_wrist_flex_link` und in Abbildung 5.2 abgebildet. Durch das ROS-Package `tf (transformation)` kann die Translation von diesem Link zu den linken und rechten Greifer-Backen herausgefunden werden. Mithilfe von Einheitsvektoren des `arm_wrist_flex_link` und der Translation werden dann die Positionen der rechten und linken Greifer-Backen (beschrieben in dem `/world`-Koordinatensystem) berechnet. Dies ist in Listing 5.6 zu sehen.

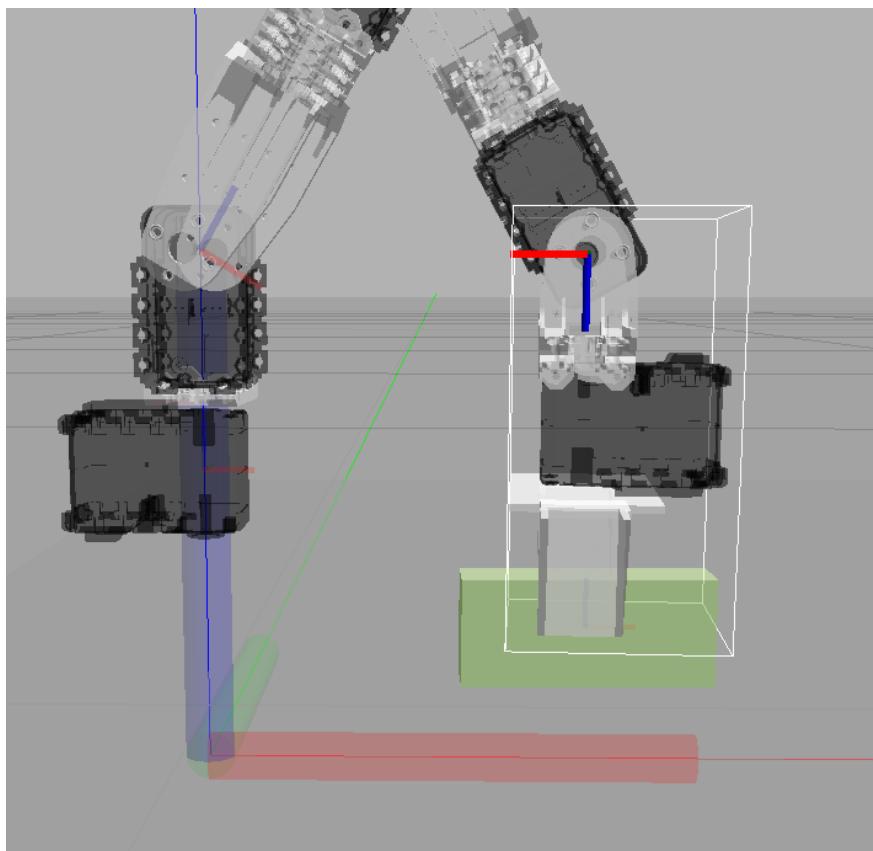


Abbildung 5.2: Pincher-Arm mit markiertem `arm_wrist_flex_link`, welcher sich nicht relativ zu den Greifer-Backen-Links bewegt

Listing 5.6: Code für das Berechnen der Greifer-Backen-Positionen & der Pose des Quaders im Pincher-environment (Auszug)

```

1 def getPoseOfObjectAndGripper(self):
2     object_positions = rospy.wait_for_message("/gazebo/link_states",
3                                             LinkStates, timeout=5)
4     indices = [i for i, s in enumerate(object_positions.name) if "
5         objectToPickUp" in s]
6     indexOfObjectToPickUp = indices[0]

```

```

5     pickup_pose = object_positions.pose[indexOfObjectToPickUp]
6
7     # We can only get the arm_wrist_flex_link via the /gazebo/
8     # link_states topic, so we get the pose of this link (which doesn
9     # t move relative to the finger links) and use this
10    # arm_wrist_flex_link pose to calculate the position of the two
11    # finger links
12    indices = [i for i, s in enumerate(object_positions.name) if "
13        arm_wrist_flex_link" in s]
14    indexOfGripper = indices[0]
15    arm_wrist_pose = object_positions.pose[indexOfGripper]
16
17    # Getting translations from arm_wrist_flex_link to gripper_active
18    # link (we don't care for rotation, as we are only interested in
19    # the position of the gripper_active link (and not its
20    # orientation as well)). As mentioned above, these two coordinate
21    # systems don't move relative to each other.
22    listener = tf.TransformListener()
23    listener.waitForTransform("/arm_wrist_flex_link", "/"
24        gripper_active_link", rospy.Time(), rospy.Duration(4.0))
25    (trans, rot) = listener.lookupTransform("/arm_wrist_flex_link", "/"
26        gripper_active_link", rospy.Time())
27
27    # calculating the unit vectors (described in the /world coordinate
28    # system) of the arm_wrist_flex_link
29    ex, ey, ez = self.getUnitVectorsFromOrientation(arm_wrist_pose.
30        orientation)
31
32    # calculating the positions of the finger links (described in the
33    # world coordinate system)
34    # position of gripper_active_link (gripper_right_position)
35    gripper_right_position = np.matrix([[arm_wrist_pose.position.x], [
36        arm_wrist_pose.position.y], [arm_wrist_pose.position.z]])
37    gripper_right_position = gripper_right_position + trans[0]*ex +
38        trans[1]*ey + trans[2]*ez
39    # print(gripper_right_position) # e.g. [[0.09] [0.013] [0.058]]
40
41    # (same for position of gripper_active2_link (gripper_left_position
42    # ))
43
44    return pickup_pose, gripper_right_position, gripper_left_position
45    # getPoseOfObjectAndGripper

```

5.5 Einlesen der Tiefenbilder

Das Einlesen des aktuellen Tiefenbildes der Kinect-Kamera geschieht über das ROS-topic /camera/depth/image_raw (siehe Abschnitt 4.2). Dies ist auch bei der realen Kinect der Fall, jedoch sind die Tiefenbilder der realen Kinect in einer anderen Codierung als die der simulierten. Deshalb ist es nötig, das erhaltene Tiefenbild zuerst in eine definierte Codierung umzuwandeln. Danach wird das Bild zugeschnitten, damit nur die Region abgebildet ist, in welcher sich der Quader befinden kann. Da beim realen Tiefenbild die Entfernung in mm, beim simulierten Tiefenbild jedoch in m gemessen wird, wird das Bild nach dem Zuschneiden noch normalisiert, damit die Pixel nur noch Werte zwischen 0 und 255 annehmen können. Dies hat auch den Vorteil, dass die Bilder als Graustufen-Bilder abgespeichert und betrachtet werden können.

Damit das Tiefenbild der Simulation einem realen Bild möglichst ähnelt, ist es ausserdem nötig, einen Noise (Rauschen) auf das Bild zu geben. Obwohl Gazebo für speziell

verrauschte Sensorwerte wirbt, gilt dies nicht für Tiefenbilder oder Point Clouds, sondern nur für „normale“ Kamerabilder oder Laserscan-Sensoren. Deshalb wird nach der Normalisierung des *simulierten* Tiefenbildes ein additives Gaußsches Rauschen hinzugefügt. Der Mittelwert dieses Rauschens ist 0, für die Standardabweichung wurde 10 gewählt. Damit sieht das simulierte Bild dem echten etwas ähnlicher. Es könnten auch eine gewisse Prozentzahl an Pixel zufällig auf 0 gesetzt werden, was ein fehlender Messwert simulieren könnte. Da dann jedoch das Bild nicht wirklich einem realen Tiefenbild ähnelt, ist diese Option deaktiviert. Ein Bild der simulierten Kinect-Kamera ist in Abbildung 5.3 zu sehen.

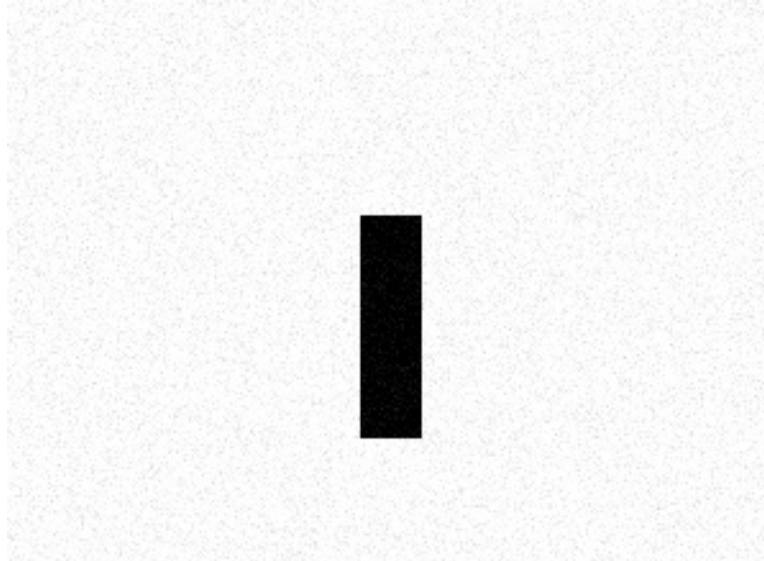


Abbildung 5.3: Tiefenbild der simulierten Kinect-Kamera mit Gaußschem Rauschen

Bevor das Tiefenbild schliesslich als State zurückgegeben wird, werden die Pixelwerte noch auf 8-bit Integer gerundet, was Speicherplatz spart. Der Code für das Empfangen und Nachbereiten des simulierten Tiefenbildes ist in Listing 5.7 nachzulesen.

Listing 5.7: Code für das Einlesen der simulierten Tiefenbilder im Pincher-environment (Auszug)

```

1  def read_kinect_depth_image(self, setRandomPixelsToZero=False,
2                               saveImage=False, folder="/home/joel/Pictures/", saveToFile=False):
3      data = rospy.wait_for_message("/camera/depth/image_raw", Image,
4                                    timeout=5)
5      # encoding of simulated depth image: 32FC1, encoding of real depth
6      # image: 16UC1
7      # converting & normalizing image, so that simulated & real depth
8      # image have the same encoding & value range
9      # see http://wiki.ros.org/cv_bridge/Tutorials/
10     # ConvertingBetweenROSImagesAndOpenCVImagesPython
11     cv_image = self.bridge.imgmsg_to_cv2(data, desired_encoding="32FC1"
12                                         )
13
14     image = np.array(cv_image, dtype=np.float) # shape = (480, 640)
15
16     # cutting image so that it only contains region of interest
17     image = image[240-170:240+50, 320-150:320+150] # shape = (220, 300)
18     # normalizing image
19     cv2.normalize(image, image, 0, 255, cv2.NORM_MINMAX)
20     # adding noise to simulated depth image
21     image = image + np.random.normal(0.0, 10.0, image.shape)
22     # round to integer values and don't allow values <0 or >255 (
23     # necessary because of the added noise)

```

```

17     image = np.clip(np.round(image), 0, 255).astype(np.uint8)
18
19     if(setRandomPixelsToZero):
20         # set approx. 5% of all pixels to zero
21         mask = (np.random.uniform(0,1,size=image.shape) > 0.95).astype(
22             np.bool)
23         image[mask] = 0
24
25     # (saving image if requested)
26
27     return image
# read_kinect_depth_image

```

5.6 Definition des State- & Action-Spaces

Dem RL-Algorithmus kann über environment-Eigenschaften (*properties*) mitgeteilt werden, welche Dimensionen der State- und der Action-Space aufweist. Zusätzlich können dazu auch noch die Minimal- und Maximal-Werte jeder Dimension gesetzt werden. Damit kann nicht nur der RL-Algorithmus sich dem jeweiligen environment anpassen (z.B. durch andere neuronale Netze), durch die Eingrenzungen des Action-Spaces können auch im Vorhinein klar schlechte Actions ausgeschlossen werden. Im environment dieses Projektes werden so mit der Einschränkung der möglichen x-, y- und z-Koordinaten der anzufahrenden Position die Actions viel wahrscheinlicher zum Erfolg führen, als wenn der Algorithmus alle möglichen wilden Positionen anzufahren versucht. So werden in der init-Methode des environments (siehe Listing 5.2) die 6 Dimensionen des Action-Spaces auf die ungefähren möglichen Positionen des Quaders eingeschränkt. Wie dies in der init-Methode gemacht wird, ist in Listing 5.8 zu sehen.

Listing 5.8: Definition des State- & Action-Spaces in der init-Methode des Pincher-environments

```

1 self.observation_space = spaces.Box(low=0, high=255, shape=[220,300],
2                                     dtype=np.uint8)
3
4 # the action space are all possible positions & orientations (6-DOF),
5 # which are bounded in the area in front of the robot arm where an
6 # object can lie
7 boundaries_xAxis = [0.04, 0.3]          # box position possiblities x-axis:
8                 (0.06, 0.22)
9 boundaries_yAxis = [-0.25, 0.25]        # box position possiblities y-axis:
10                (-0.2, 0.2)
11 boundaries_zAxis = [0.015, 0.05]        # box z-position: ca. 0.04
12 boundaries_roll = [0, 2*np.pi]
13 boundaries_pitch = [0, 2*np.pi]
14 boundaries_yaw = [0, 2*np.pi]
15
16 low = np.array([boundaries_xAxis[0], boundaries_yAxis[0],
17                 boundaries_zAxis[0], boundaries_roll[0], boundaries_pitch[0],
18                 boundaries_yaw[0]])
19 high = np.array([boundaries_xAxis[1], boundaries_yAxis[1],
20                  boundaries_zAxis[1], boundaries_roll[1], boundaries_pitch[1],
21                  boundaries_yaw[1]])
22 self.action_space = spaces.Box(low=low, high=high, dtype=np.float32)
23
24 self.reward_range = (-np.inf, np.inf)

```

6 Versuchsaufbau in Hardware

Da das OpenAI Gym environment mit dem Simulations-Setup nun definiert ist, kann dieses Setup auch in echt (also in Hardware) aufgebaut und installiert werden. Der reale Versuchsaufbau besteht aus einer Grundplatte, auf welcher der Pincher und die Halterung für die Kinect montiert sind. Ein Bild des Versuchsaufbaus ist in Abbildung 6.1 zu sehen (ohne aufzunehmenden Quader).



Abbildung 6.1: realer Versuchsaufbau mit Pincher-Arm und Kinect-Kamera

6.1 Inbetriebnahme realer Roboter

Damit der reale Pincher Roboter-Arm über die MoveIt Python API aus der virtuellen Maschine angesteuert werden kann, ist neben der Einrichtung des realen Pinchers auch eine andere Konfiguration der Launch-Files notwendig. Ein Teil der Inbetriebnahme des Roboters ist u.a. ebenfalls auch in dem MSE-Vertiefungsprojekt „Innenraumerfassung mit dem Turtlebot“ von Fabian Saccilotto [39] im dortigen Anhang A beschrieben.

6.1.1 Treiber in Ubuntu

Für den realen Pincher-Arm ist ein anderes MoveIt-Package nötig als wie für den simulierten Pincher-Arm. Dieses entspricht mehrheitlich demjenigen aus dem Github-Repository des Pincher-Armes [18]. Es werden aber andere Controller benötigt, nämlich eine spezielle (inoffizielle) Version der Arbotix-Controller [40]. Dieses Package kann wie alle anderen Packages von Github heruntergeladen werden, in einem Catkin-Workspace installiert und schliesslich „gesourced“ werden, damit das Package auch von ROS gefunden werden kann (siehe Unterabschnitt 4.1.1). Letzteres sollte in dem `~/.bashrc`-File ebenfalls gemacht werden, damit dies nur einmal gemacht werden muss (und nicht bei jedem Start eines neuen Terminals). Im Launch-File des MoveIt-Packages für den realen Pincher muss zudem das Argument `simulation` auf `false` gesetzt werden, wie in Listing 6.1 gezeigt. Dies bewirkt, dass die Arbotix-Controller den realen Pincher über die serielle Schnittstelle `/dev/ttyUSB0` anzusprechen versuchen.

Listing 6.1: Ausschnitt aus dem Launch-File des MoveIt-Packages für den realen Pincher-Arm

```

1 <!-- Load arm description, state and controllers -->
2 <include file="$(find turtlebot_arm_bringup)/launch/arm.launch">
3   <arg name="simulation" value="false"/>
4 <!-- when set to false, arbotix drivers try to open /dev/ttyUSB0
     connection -->
5 </include>
```

6.1.2 Hardware verkabeln & anschliessen

Der reale Pincher-Arm wird über ein FTDI-USB-Kabel mit dem PC verbunden, auf welchem auch die Ubuntu-VM mit den ganzen Komponenten wie OpenAI Gym läuft. Eine schematische Darstellung ist in Abbildung 6.2 abgebildet. Damit die Energieversorgung des Pinchers nicht über das FTDI-Kabel geschieht, sondern der Roboter über die externe Stromversorgung für die 230V-Steckdose gespiesen wird, muss der Power-Jumper auf dem Pincher-Board auf die VIN-Position gesetzt werden, wie in Abbildung 6.2 rechts neben dem Steckplatz für das FTDI-Kabel dargestellt ist. Die Speisung des Pinchers über die externe Stromversorgung ist nötig, damit die Servomotoren angesteuert werden können.

6.1.3 Pincher ROS-tauglich machen

Ist der Pincher über das FTDI-Kabel mit dem PC verbunden, muss auf dem Host-Betriebssystem (Windows 10) mithilfe der (vorher) installierten Arduino IDE das ROS-Sketchbook ausgewählt werden, wie dies in Abbildung 6.3 dargestellt ist. Um dieses Sketchbook (Programm) auf das Pincher-Board zu laden, muss anschliessend der

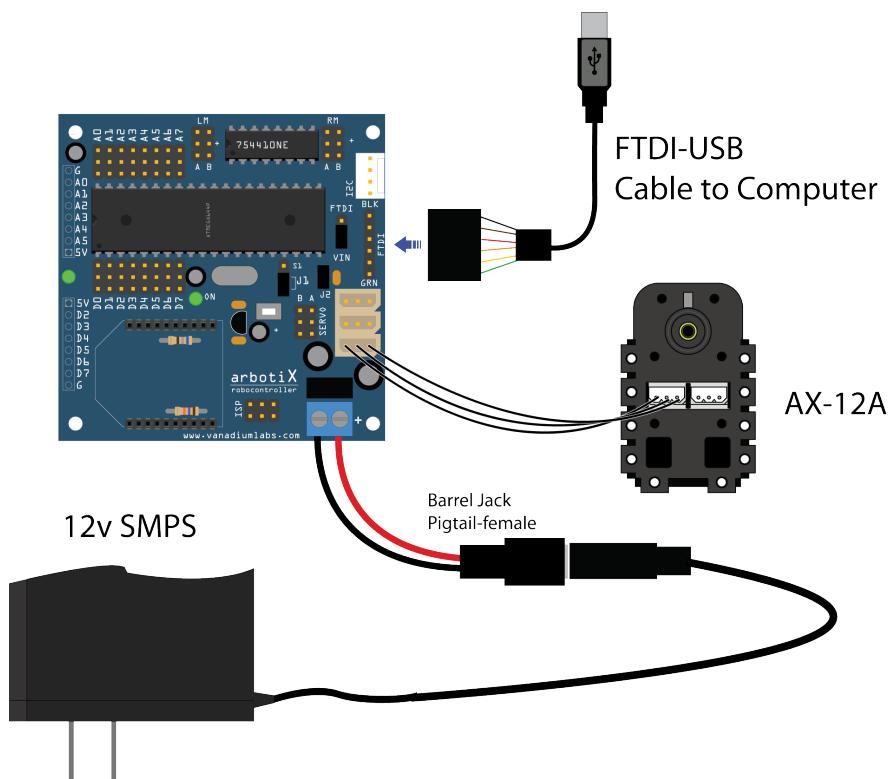


Abbildung 6.2: Verkabelung des Pincher-Armes [41]

Upload-Button der Arduino IDE gedrückt werden. Dies ermöglicht es dem realen Pincher-Arm, ROS-messages entgegenzunehmen und entsprechend auszuführen.

Solange kein anderes Sketchbook auf das Pincher-Board geladen wird, ist dieser Schritt nur einmal nötig, da das Pincher-Board beim Hochfahren jeweils das letzte geladene Sketchbook automatisch wieder ausführt.

6.1.4 Verbindung zu VM herstellen

Sobald der reale Pincher-Arm für ROS-messages empfänglich ist, kann die USB-Verbindung in die virtuelle Maschine überbrückt werden. Dies kann in VirtualBox während des Betriebes der VM über das Geräte-Menu einfach gemacht werden, wie dies in Abbildung 6.4 gezeigt ist.

Wenn nun das korrekte MoveIt-Package gelaunched wird, kann der reale Pincher-Arm mit den exakt gleichen Python-Befehlen wie beim simulierten Pincher angesteuert werden. Ein solches Beispiel-Programm ist in Listing 4.5 im Unterabschnitt 4.1.6 zu sehen.

6.2 Inbetriebnahme reale Kinect-Kamera

Damit die Daten der realen Kinect in der Ubuntu-VM über ROS-topics zur Verfügung stehen, müssen die richtigen Treiber installiert werden. Es gibt dazu mehrere in Frage kommende Treiber-Pakete, wie beispielsweise *Openni* [42]. Bei Openni besteht jedoch das Problem, dass es in einer virtuellen Maschine nicht richtig funktioniert. Deshalb ist das in diesem Projekt schliesslich verwendete Treiber-Paket *freenect* [43]. Die

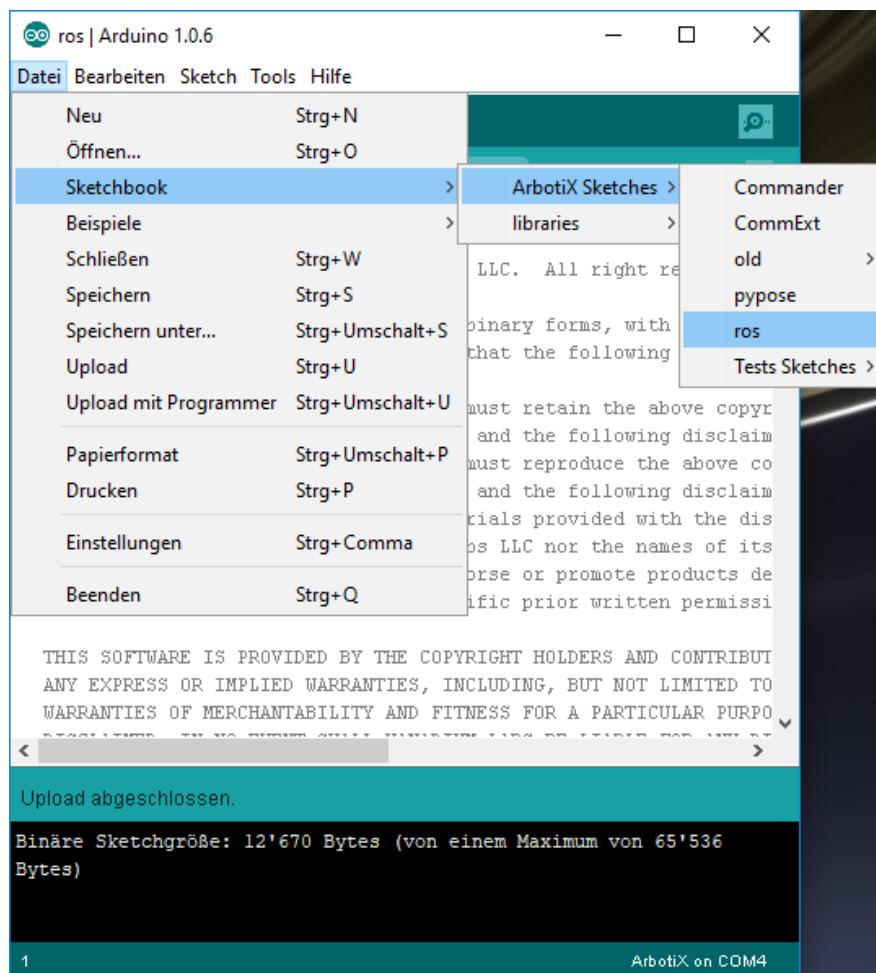


Abbildung 6.3: Auswahl des ROS-Sketchbooks in der Arduino IDE (auf dem Host-Betriebssystem)

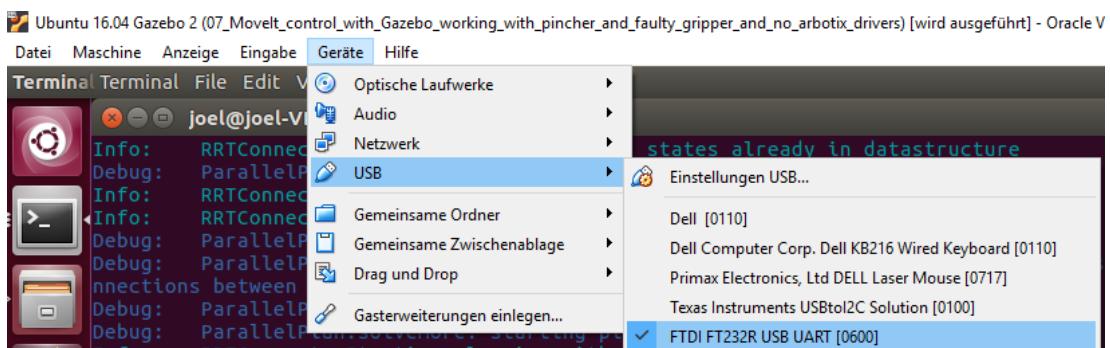


Abbildung 6.4: USB-Bridge für den realen Pincher-Arm in die Ubuntu-VM (von VirtualBox)

Installations-Anleitung ist u.a. auch in dem MSE-Vertiefungsprojekt 1 „Einführung in ROS“ von Manuel Ilg [25] im dortigen Kapitel 3.4 zu finden.

6.2.1 USB 3.0 Controller für VM

Damit nach der Installation von freenect in der Ubuntu-VM die Daten der realen Kinect auch darin empfangen werden können, muss in den Einstellungen der VM ein USB

3.0 Controller ausgewählt sein. Dies ist in Abbildung 6.5 dargestellt. Damit dies funktioniert, ist das VM VirtualBox Extension Pack nötig, welches beispielsweise von [44] heruntergeladen werden kann.

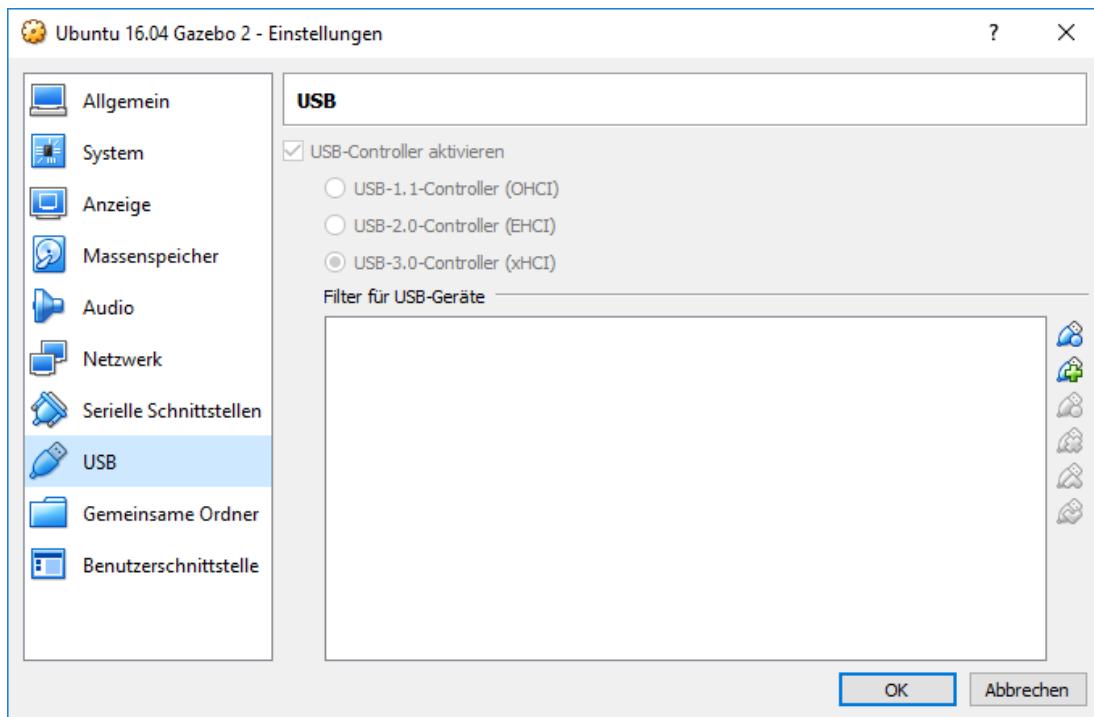


Abbildung 6.5: USB-Settings in VirtualBox für die Verbindung der realen Kinect-Kamera mit der Ubuntu-VM

6.2.2 Verbindung zur VM herstellen

Sobald die reale Kinect mit einer USB 3.0-Schnittstelle des PCs verbunden ist, muss wie bei dem realen Roboter-Arm die Verbindung in die VM überbrückt werden. Bei der Kinect stehen insgesamt drei verschiedene USB-Devices zur Verfügung, welche alle über das Geräte-Menu der laufenden VM angewählt werden müssen, wie in Abbildung 6.6 dargestellt. Es ist dabei empfehlenswert, zuerst das *Microsoft Xbox NUI Motor Device* auszuwählen, da nach der Brückenbildung für dieses Device die beiden anderen kurz verschwinden und erst nach ein paar Sekunden wieder auftauchen.

Nach dem Starten des freenect-Launch-Files stehen die Daten der realen Kinect-Kamera über diverse ROS-topics zur Verfügung und können beispielsweise in einem Python-Programm ausgelesen werden. Ein Beispiel-Code dazu ist im Abschnitt 4.2 zu sehen.

Die Daten können zu Testzwecken auch mit dem ROS-Visualisierungstool Rviz betrachtet werden. Ein Beispiel einer realen Punktewolke, welche in Rviz visualisiert ist, ist in Abbildung 6.7 abgebildet.

6.2.3 Einlesen der Tiefenbilder

Wie bereits im Abschnitt 5.5 erwähnt, funktioniert das Einlesen der realen Tiefenbilder prinzipiell gleich wie das der simulierten Tiefenbilder. Die Daten werden ebenfalls über das ROS-topic `/camera/depth/image_raw` empfangen. Die Nachbereitung der realen Bilder unterscheidet sich aber von dem Simulations-Fall, da hier beispielsweise

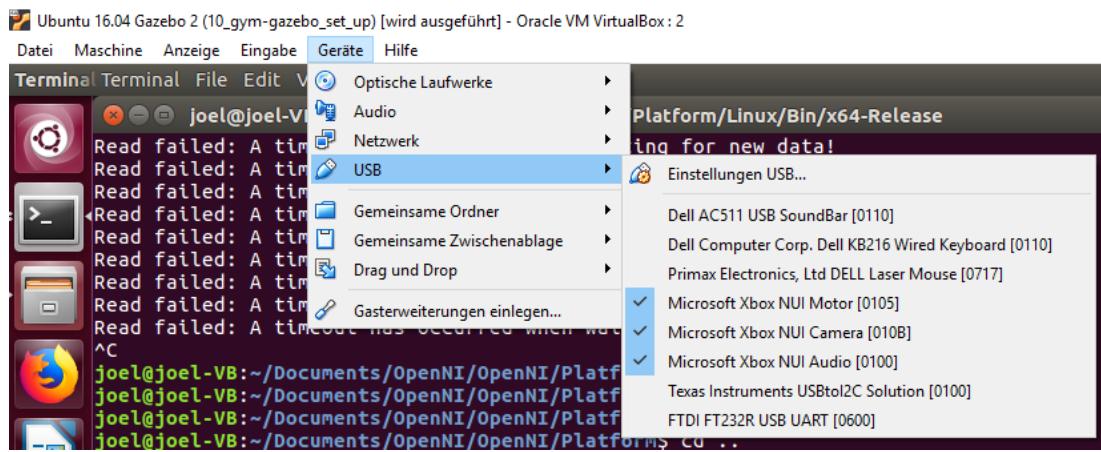


Abbildung 6.6: USB-Bridge für die reale Kinect-Kamera in die Ubuntu-VM (von VirtualBox)

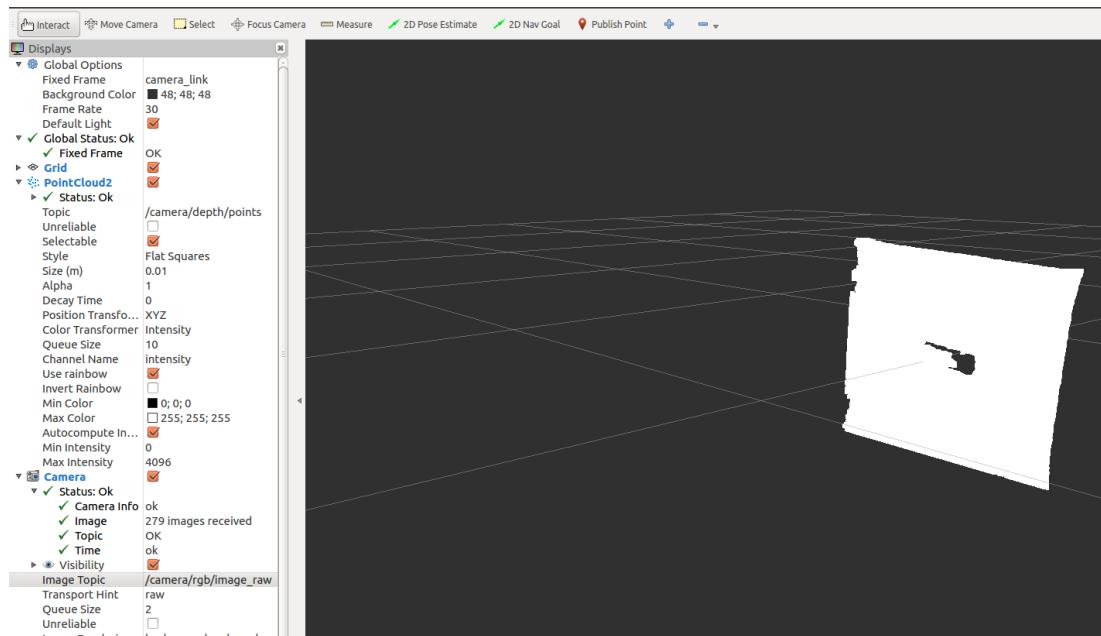


Abbildung 6.7: In Rviz visualisierte Punktewolke einer realen Kinect-Kamera

kein Noise mehr auf das Tiefenbild gegeben werden muss. Um zudem ein möglichst gutes Bild zu erhalten, werden 6 einzelne Bilder hintereinander eingelesen und (nach dem Zuschneiden auf den relevanten Bildbereich) gemittelt. Ein Tiefenbild der realen Kinect-Kamera ist in Abbildung 6.8 abgebildet. Das simulierte Tiefenbild ähnelt dem realen noch nicht sehr stark. Die Qualität des realen Tiefenbildes könnte durch eine höhere Montage der Kinect-Kamera noch etwas verbessert werden (siehe Kapitel 9). Dies wurde in diesem Projekt aufgrund der fehlenden Befestigungsmaterialien und Zeit jedoch nicht mehr durchgeführt.



Abbildung 6.8: Tiefenbild der realen Kinect-Kamera (Mittelwert von 6 Aufnahmen, mit einem etwas anderem Quader als schliesslich in der Simulation verwendet)

7 RL-Algorithmus

Nachdem die Gazebo-Simulation fertiggestellt und das OpenAI Gym environment implementiert ist, kann der RL-Algorithmus in Angriff genommen werden. Der State des RL-Agenten ist dabei das von der Kinect-Kamera aufgenommene Tiefenbild, auf welchem der zufällig platzierte Quader zu sehen ist. Obwohl dies ein Pixel-Array mit Integer-Werten und somit streng genommen ein diskreter State-Space ist, ist die Anzahl der Möglichen States riesig. Deshalb wird der State-Space als kontinuierlich angesehen, was eine Value Function Approximation (siehe Abschnitt 2.6) voraussetzt macht. Der Action-Space ist ein kontinuierlicher, 6-dimensionaler Vektor, welcher auf einen bestimmten Bereich eingeschränkt ist (siehe Abschnitt 5.6). RL-Tasks mit kontinuierlichen Action-Spaces sind jedoch viel schwieriger zu lösen als solche mit diskreten Actions. Nichtsdestotrotz gibt es RL-Algorithmen, welche auch mit kontinuierlichen Actions umgehen können, wie beispielsweise der DDPG-Algorithmus, welcher im Unterabschnitt 2.7.2 beschrieben ist.

7.1 Grundsätzlicher Trainings-Ablauf

Jede Algorithmus-Implementation hat ihre eigenen Methoden, wie die RL-Konzepte wie Policy oder die Function Approximation implementiert wird. Bei allen gleich ist hingegen ein grundsätzlicher Trainings-Ablauf, welcher hier kurz beschrieben wird:

1. Erzeuge environment, initialisiere alles Notwendige
2. Für jede Episode
 - a) Reset Simulation
 - i. Setze zufällige Position des Quaders (sofern nötig)
 - ii. Nehme Tiefenbild auf → State
 - b) Für jeden Zeitschritt in der Episode
 - i. Wähle eine Action aufgrund des momentanen States aus
 - ii. Führe `env.step()` aus mit gewählter Action
 - A. Fahre mit Roboter-Arm an die Position
 - Position ist erreichbar: Berechne ob Quader greifbar wäre (→ Reward), anschliessend fährt der Arm wieder auf Start-Position. Der Quader wird wieder wie vorhin platziert (nötig, falls Greifer in den Quader gefahren ist).
 - Position ist nicht erreichbar: Reward = niedrig
 - B. Neues Tiefenbild aufnehmen → nächster State (ist dann ähnlich wie vorheriger State)
 - C. environment gibt neuen State, erhaltener Reward & einen boolean (`done`) dem RL-Agenten zurück. `done` ist immer `False`, da es kein episodischer Task ist.
 - iii. Lerne anhand des erhaltenen Rewards (bewerte vergangene Actions)

7.2 Reduzierung der Action-Space Dimensionen

Um die Chancen zu erhöhen, dass ein RL-Algorithmus (in angemessener Trainings-Zeit) zu einer zufriedenstellenden Performance konvergiert, werden bei allen Algorithmen zusätzlich zu der bereits im environment erfolgten Einschränkung des Action-Spaces (siehe Abschnitt 5.6) der roll- und der pitch-Winkel der anzufahrenden Position auf 0 bzw. $\frac{\pi}{2}$ fixiert. Dadurch schaut der Greifer des Pincher-Armes stets senkrecht nach unten, was die Wahrscheinlichkeit eines erfolgreichen Greif-Prozesses erhöht. Dies hat zur Folge, dass der eigentliche noch durch den Algorithmus zu erkundende Action-Space nur noch 4-dimensional ist.

In den folgenden Abschnitten wird kurz auf verschiedene, in diesem Projekt getestete Algorithmen eingegangen.

7.3 DQN-Implementierung

Eine Variante, wie mit kontinuierlichen Actions umgegangen werden kann, ist die simple Diskretisierung des Action-Spaces (wie bereits im Abschnitt 2.7 erwähnt). Dabei wird jede Dimension der Action (in dem Pincher-Beispiel 4 Freiheitsgrade) in eine bestimmte Anzahl Bereiche unterteilt, beispielsweise 10. Jeder dieser Bereiche stellt dann eine diskrete Action dar. Dies resultiert jedoch rasch in einer sehr grossen Anzahl an Actions. Bei den 4 Dimensionen der Action für den Pincher-Arm (x, y, z, yaw) und je 10 Bereiche ergibt das $10^4 = 10'000$ verschiedene Actions! Im Deep Q-Learning Algorithmus (DQN, erklärt im Unterabschnitt 2.6.1) hat so beispielsweise das neuronale Netz, welches die Action-Value jeder möglichen Action anhand des momentanen States approximiert, ganze 10'000 Outputs. Dies resultiert in einer sehr grossen Anzahl an zu trainierenden Parameter des Netzwerks (> 200 Mio.), wie dies in Listing 7.1 zu sehen ist. Wenn vom Algorithmus eine Action ausgewählt wird, wird für jede Action-Dimension jeweils die Mitte des jeweiligen Bereichs genommen und diese Position dann als Action der step-Methode übergeben.

Das Template der für dieses Projekt verwendeten DQN-Implementierung stammt von [45]. Der Code wurde für das Pincher-environment angepasst und die Action Value Function Approximation als ein CNN anstatt ein Fully-Connected Neural Net implementiert. Zudem wurde im Hauptprogramm die vorhin erwähnte Diskretisierung des Action-Spaces durchgeführt.

7.3.1 Resultate

Der DQN-Algorithmus wurde einmal über Nacht laufen gelassen, wobei die Position des Quaders fixiert wurde. Da jedoch ein einziger Zeitschritt ungefähr ganze 27s benötigte, konnten nur ca. 2'000 Zeitschritte simuliert werden. Ein Auszug aus dem Logging des Algorithmus ist in Listing 7.1 zu sehen. In einem Zeitschritt benötigte dabei die step-Methode des environments (Simulation in Gazebo inkl. der Reward-Berechnung) ca. 1.5s, das Zusammenstellen des Mini-Batches 17s und das Fitten des CNNs mit diesem Mini-Batch ca. 9s. Somit wurde die meiste Zeit eines Zeitschrittes durch das Handling der Function Approximation benötigt. Dies liegt an der sehr grossen Anzahl an Netzwerk-Parametern und der Tatsache, dass in einer VM kein GPU-Support für Tensorflow¹ möglich ist. Somit laufen alle Algorithmen „nur“ auf den 4 in der VM zur Verfügung stehenden CPU-Kernen.

¹Tensorflow ist ein Open Source Machine Learning Framework [46].

Das in keinem der Zeitschritte des Trainings dabei die korrekte (immer gleich gebliebene) Position angefahren werden konnte, überrascht deshalb an dieser Stelle nicht. Ein Zeitschritt dauert viel zu lange, die Anzahl der Parameter des neuronalen Netzes sowie die Anzahl an möglichen (diskreten) Actions sind viel zu gross und die Diskretisierung des Action-Spaces ist generell keine gute Herangehensweise für dieses Problem. Somit ist ein DQN-Algorithmus für dieses Problem ungeeignet.

Listing 7.1: Ausschnitt aus der Logging des DQN-Algorithmus

1	Layer (type)	Output Shape	Param #
<hr/>			
3	conv2d_1 (Conv2D)	(None, 218, 298, 16)	160
<hr/>			
5	conv2d_2 (Conv2D)	(None, 216, 296, 16)	2320
<hr/>			
7	flatten_1 (Flatten)	(None, 1022976)	0
<hr/>			
9	dense_1 (Dense)	(None, 200)	204595400
<hr/>			
11	dense_2 (Dense)	(None, 10000)	2010000
<hr/>			
13	Total params:	206,607,880	
14	Trainable params:	206,607,880	
15	Non-trainable params:	0	
<hr/>			
17	resetting environment		
18	1		
19	calling env.step... 0		
20	moving arm to position 0.157, -0.175, 0.425, 0.000, 1.571, 5.341		
21	calling env.step... 1		
22	moving arm to position 0.261, 0.025, 0.225, 0.000, 1.571, 5.969		
23	calling env.step... 2		
24	moving arm to position 0.079, 0.125, 0.425, 0.000, 1.571, 5.341		
25	calling env.step... 3		
26	moving arm to position 0.053, -0.125, 0.375, 0.000, 1.571, 4.084		
27	...		
28	calling env.step... 1936		
29	moving arm to position 0.209, 0.075, 0.475, 0.000, 1.571, 2.199		
30	learning on mini-batch		
31	fitting model		
32	calling env.step... 1937		
33	moving arm to position 0.209, -0.075, 0.175, 0.000, 1.571, 0.942		
34	learning on mini-batch		
35	fitting model		

7.4 DDPG-Implementierung

Wie bereits im Abschnitt 2.7 erwähnt, können Policy Gradient Algorithmen besser mit kontinuierlichen Action-Spaces umgehen. Aus diesem Grund wurden verschiedene Implementationen des Deep Deterministic Policy Gradient Algorithmus (siehe Unterabschnitt 2.7.2) an das Pincher-environment angepasst und deren Performance analysiert.

Eine der DDPG-Implementierung stammt aus der RL-Algorithmen-Sammlung *baselines* [47], welche von OpenAI Gym selbst zur Verfügung gestellt wird. Da diese im Vergleich zu anderen die Implementierung mit der besten Exploration ist (siehe Unterabschnitt 7.4.2), wird in diesem Abschnitt nur auf diese Implementierung eingegangen. Darin werden die meisten Berechnungen mit Tensorflow gemacht, was den Code (gerade für Tensorflow-Anfänger) eher schwierig zu lesen macht. Nichtsdestotrotz konnte

die Implementierung auf das Pincher-environment angepasst werden, wobei der State (d.h. das Tiefenbild) anstatt als 2D-Array in ein 1D-Array gestreckt werden musste, da die neuronalen Netze des Actors und des Critics „nur“ Fully-Connected Neural Nets sind. Ob das Tiefenbild gestreckt werden soll, kann im environment mit der booleschen Klassenvariable `flattenImage` eingestellt werden.

Ein weiterer zu beachtender Punkt sind die Kompatibilitätsprobleme von Python 2 und Python 3. Da ROS (Kinetic Version) nur mit Python 2 angesteuert werden kann, viele Implementierungen von RL-Algorithmen (wie jene von baseline) aber für Python 3 geschrieben sind, sind unter Umständen Anpassungen des RL-Algorithmus-Codes vonnöten, damit alles mit Python 2 laufen lassen werden kann.

7.4.1 Skalierung der Action-Bereiche

In der Standard-Version des DDPG-Algorithmus kann die durch den Actor ausgewählte Action nur Werte zwischen -1 und +1 annehmen (für jede Dimension der Action). Deshalb nehmen quasi alle DDPG-Implementierungen symmetrische Action-Spaces an. Um trotzdem (ein wenig) environment-unabhängig zu sein, wird die vom Actor ausgewählte Action in vielen Implementierungen vor dem Ausführen mit der oberen Grenze des Bereiches der jeweiligen Action-Dimension multipliziert. Dies funktioniert jedoch nur mit symmetrischen Bereichen für die Action-Space-Dimensionen.

Um trotzdem mit den asymmetrischen Bereichen des Pincher-environments (siehe Abschnitt 5.6) arbeiten zu können, können die Dimensionen der vom Actor ausgewählten Action einfach linear von dem [-1, 1] Bereich auf denjenigen des Pincher-environments skaliert werden. Die Implementation dazu ist in Listing 7.2 gezeigt.

Listing 7.2: Ausschnitt aus dem DDPG-Algorithmus mit der Skalierung der Action-Dimensionen

```

1 ...
2 for t_rollout in range(nb_rollout_steps):
3     # Predict next action.
4     action, q = agent.pi(obs, apply_noise=True, compute_Q=True)
5     # e.g. action = array([ 0.02667301,  0.9654905 , -0.5694418 , -0.
6         40275186], dtype=float32)
7
8     # scale for execution in env (as far as DDPG is concerned, every
9     # action is in [-1, 1])
10    target = np.insert(action, 3, [0.0, 0.0])
11    target = scale_range(target, -1, 1, env.action_space.low, env.
12        action_space.high)
13    # e.g. target = array([0.17346749, 0.24137263, 0.10763955, 3.
14        1415926, 3.1415926, 1.8763105 ], dtype=float32)
15    # we keep the roll & pitch angle fixed
16    target[3] = 0.0
17    target[4] = np.pi/2
18
19    # Execute next action.
20    assert target.shape == env.action_space.shape
21    new_obs, r, done, info = env.step(target)
22    t += 1
23
24    ...
25
26    def scale_range(x, x_min, x_max, y_min, y_max):
27        """ Scales the entries in x which have a range between x_min and
28            x_max
29            to the range defined between y_min and y_max. """
30
31        # y = a*x + b

```

```

25      # a = deltaY/deltaX
26      # b = y_min - a*x_min (or b = y_max - a*x_max)
27      y = (y_max - y_min) / (x_max - x_min) * x + (y_min*x_max - y_max*x_
28          x_min) / (x_max - x_min)
29      return y

```

7.4.2 Parameter-Noise anstatt Action-Noise

In herkömmlichen DDPG-Implementierungen wird die Exploration durch einen hinzugefügten Noise auf die vom Actor-Netzwerk ausgewählte Action gelöst. In der DDPG-Variante von baseline wird jedoch ein anderer Ansatz verwendet, welcher bessere Exploration verspricht [48]. Hier wird nämlich nicht beim Output des Actor-Netzwerkes ein Noise hinzugefügt, sondern bei allen Parametern des Actor-Netzwerkes selbst (siehe Abbildung 7.1).

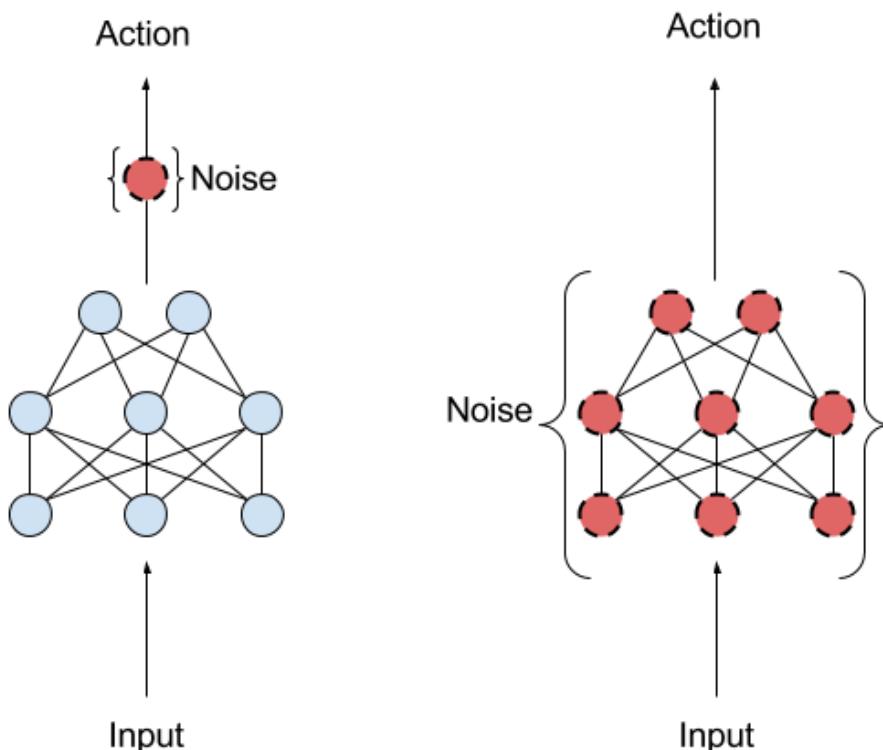


Abbildung 7.1: Action-Noise (links) vs. Parameter-Noise (rechts) [48]

Die bessere Exploration durch den Parameter-Noise hat sich auch in den Testläufen mit dieser DDPG-Implementierung bestätigt. Im Vergleich zu anderen, ebenfalls an das Pincher-environment angepasste und getestete Implementierungen ([49] & [50]) wurde dank des Parameter-Noise nicht immer von Anfang an die gleiche Position angefahren.

7.4.3 Speichern & Laden eines Modells

Reinforcement Learning ist relativ unnütz, wenn ein gelerntes Modell nicht gespeichert und bei Bedarf wieder geladen werden kann. Da in der DDPG-Implementierung von baseline zum Zeitpunkt des Projektes jedoch noch keine solche Funktionalität vorhanden war, musste diesbezüglich der Code ebenfalls etwas erweitert werden. Details dazu sind unter [51] zu finden.

7.4.4 Resultate

Obwohl der DDPG-Algorithmus (und speziell die etwas angepasste Implementierung von baseline) am besten funktionierten, konnte bis zur Abgabe des Projektes leider auch hier keine gute Performance verzeichnet werden. So konnte beispielsweise nur jeweils ca. einmal in 1'000 Versuchen (Zeitschritten) die korrekte (immer gleich gebliebene Position) angefahren werden. Dies hat verschiedene Gründe:

- **Unerreichbare Positionen:** In den meisten Fällen konnte der Roboterarm die vom Algorithmus vorgeschlagene Position nicht erreichen, da der Pincher-Arm nur über 4 Freiheitsgrade verfügt. Sehr viel Rechenzeit & Versuche gehen deswegen „ins Leere“.
 - Ein Roboter-Arm mit 6 Freiheitsgraden würde eine grössere Anzahl an vom Algorithmus vorgeschlagenen Positionen erreichen. Diese Option wurde aber nicht mehr getestet, da dies (je nach Roboter) ein beachtlicher Aufwand bedeuten würde.
- **Lange Trainingszeit:** 2'000 Zeitschritte benötigten in der Ubuntu-VM mit 4 CPU-Kernen ca. 1.5h (inkl. dem Fitzen der Actor- & Critic-Netzwerke alle 100 Zeitschritte). Das ist zwar bereits um ein Vielfaches schneller als bei der DQN-Implementierung (siehe Abschnitt 7.3), jedoch bedeutet dies trotzdem, dass so eine sehr lange Zeit trainiert werden müsste, bevor der Algorithmus zu einer akzeptablen Performance konvergieren würde (wenn überhaupt).
 - Siehe dazu Unterabschnitt 7.4.6.
- **Absturz von Gazebo:** Unglücklicherweise passierte es sehr oft, dass nach einer Weile im Training der Gazebo-Server abstürzte. Dies führte zusammen mit der langsamen Simulation dazu, dass kein Training bis zum Abschluss laufen gelassen und die Performance analysiert werden konnte.
 - Siehe dazu Unterabschnitt 7.4.5.
- **Wahl der Hyper-Parameter:** In einem DDPG-Algorithmus (wie auch in vielen anderen RL-Algorithmen auch) gibt es zahlreiche Hyper-Parameter wie die Architektur der Function Approximation, die Anzahl Trainings-Episoden, die Exploration etc. Viele der Algorithmen reagieren relativ empfindlich auf diese Einstellungen, und es wird viel Erfahrung benötigt, um diese Hyper-Parameter für einen Task und ein environment richtig einzustellen.
 - Aufgrund der letzten beiden Problemen konnten keine anderen Hyper-Parameter mehr ausgetestet werden. Möglichkeiten, wie der DDPG-Algorithmus sonst noch verbessert werden könnte, sind deshalb in Kapitel 9 aufgelistet.

7.4.5 Gazebo-Abstürze

Die im vorherigen Unterabschnitt erwähnten Abstürze von Gazebo sind das grösste Problem, da durch die Abstürze kein ernsthaftes Training des RL-Agenten durchgeführt werden kann. Deshalb wurde das ganze System mit ROS, Gazebo etc. (Übersicht siehe Abschnitt 3.7) auf verschiedenen Plattformen neu aufgesetzt und wieder getestet. Dank des in diesem Projekt erstellen Setup-Skripts (siehe Anhang) konnte der Aufwand dafür aber etwas verringert werden.

So wurden alle Komponenten in einer neuen Ubuntu-VM, auf einem Cluster (ebenfalls in einer VM) und auf einem Rechner mit nativ installierten Ubuntu neu installiert und

das Training gestartet. Leider traten in allen Fällen die Abstürze ebenfalls wieder auf, auch bei Testläufen mit einem einfachen Random-Agenten. Dies lässt stark vermuten, dass das Problem in Gazebo selbst liegen muss.

Die Abstürze traten zu verschiedenen Zeitpunkten während dem Training auf, ohne erkennbaren Anlass. Aufgrund dessen könnte die Ursache der Abstürze an einer *race condition* in dem Source Code von Gazebo liegen. Abhilfe dafür könnte eine Änderung an verschiedenen Stellen im Source Code von Gazebo schaffen (evtl. von [52]), oder aber eine Installation einer neueren Version von Gazebo, wie z.B. Gazebo 8.6.0. Letzteres wurde auch ausgetestet, und die Abstürze scheinen dadurch behoben zu sein. Natürlich ist das Nicht-Vorhandensein eines solchen Fehlers (gerade bei race conditions) sehr schwierig nachzuweisen - wie generell alle Nicht-Existenzen nur sehr schwierig oder gar nicht nachzuweisen sind². Aber mit der neueren Version von Gazebo konnte der Algorithmus mehrere Stunden ohne Probleme laufen gelassen werden, ohne dass die Simulation abstürzte.

7.4.6 Lange Trainingszeiten

Ein weiteres Problem ist die vergleichsweise lange Zeit, welche für das Training benötigt wird. Ein Zeitschritt in der Simulation kann von ca. 0.5s bis 18s dauern. Diese Zeit hängt zum einen davon ab, auf welchem Rechner das ganze System läuft und wie viel Ressourcen der Rechner zur Verfügung hat. So macht es einen Unterschied, ob das System in einer VM, auf dem Cluster oder auf einem Rechner mit nativ installiertem Ubuntu läuft. Zum anderen spielt es auch eine sehr grosse Rolle, ob die vom RL-Algorithmus vorgeschlagene Position erreicht werden konnte oder nicht. Denn wie eine Laufzeitanalyse (*profiling*) ergab, wird die meiste Zeit des Trainings für das Bewegen des Pincher-Armes in der Simulation benötigt, wie in Abbildung 7.2 zu sehen ist.

Deshalb wurde versucht, das Bewegen des Pincher-Armes in der Simulation möglichst noch ein wenig zu Beschleunigen. Dies wurde mithilfe eines Skalierungsfaktors gelöst, mit welchem die Geschwindigkeit für die von MoveIt geplante Trajectory für alle Gelenk-Winkel des Armes multipliziert wurde. Dadurch können sich aber andere Probleme ergeben: das Zurückkehren in die Startposition muss teils oft wiederholt werden, bis der Arm sich auch wirklich dort befindet. Nichtsdestotrotz konnte eine (erste) Verkürzung der für das Bewegen benötigte Zeit erreicht werden, wie dies die Laufzeitanalyse in Abbildung 7.3 zeigt.

7.4.6.1 Weitere Massnahmen

Das Fitten der neuronalen Netze benötigt ebenfalls eine Weile (z.B. bei der DDPG-baseline-Implementierung: ca. 60s), da (zumindest in einer VM) alles auf der CPU gerechnet werden muss. Eine Tensorflow-Version mit GPU-Support könnte hier eine Beschleunigung bieten, sofern der Rechner natürlich auch über eine entsprechende, kompatible Grafikkarte verfügt.

Eine andere Möglichkeit zur Beschleunigung der Berechnungen könnte ein High-Performance Cluster-Computer sein, auf welchem die Berechnungen parallel ablaufen könnten. Dazu müsste aber auch das dort ausgeführte Python-Skript parallelisierbar sein, beispielsweise durch die Technik des *Message Passing Interface* (MPI). Für ein (dazu geeignetes und angepasstes) Python-Skript können so z.B. mit dem Kommandozeilen-Befehl

²Die Nicht-Existenz eines den Jupiter umkreisenden Teekessels ist beispielsweise äusserst schwierig nachzuweisen.

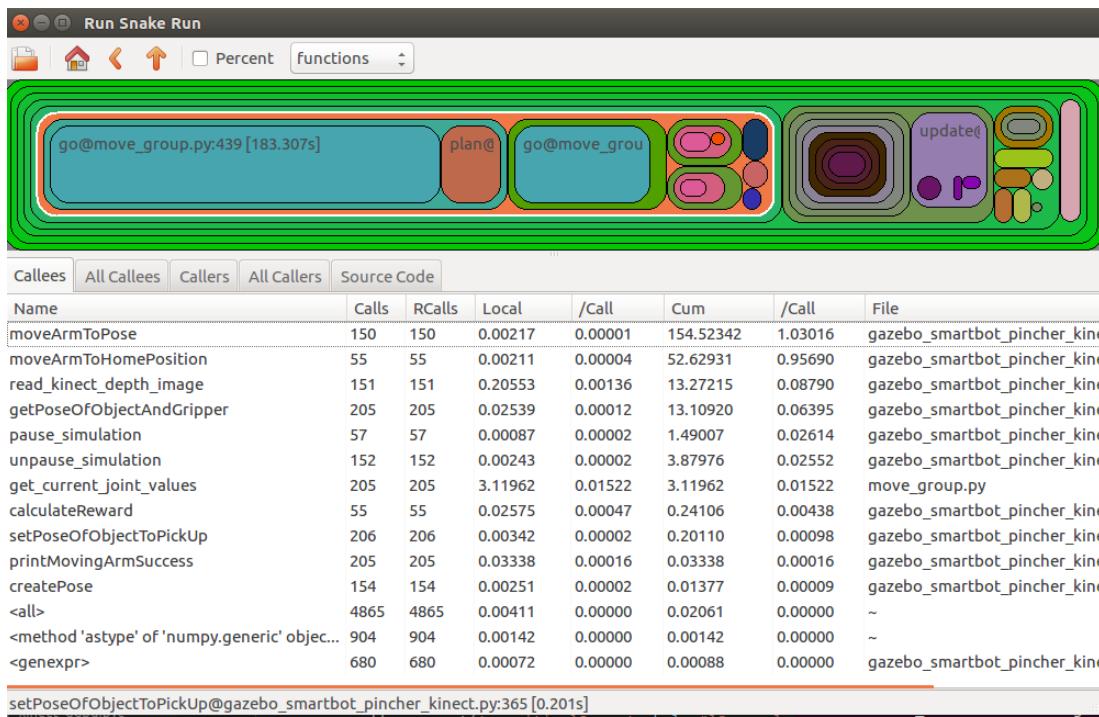


Abbildung 7.2: Laufzeitanalyse eines (verkürzten) Trainings mit dem DDPG-Algorithmus. Je grösser ein Bereich in der Grafik, desto mehr Zeit wurde für die entsprechende Methode benötigt. Gut zu erkennen ist, dass die meiste Zeit für das Bewegen des Pinchers (`go@move_group`) benötigt wird.

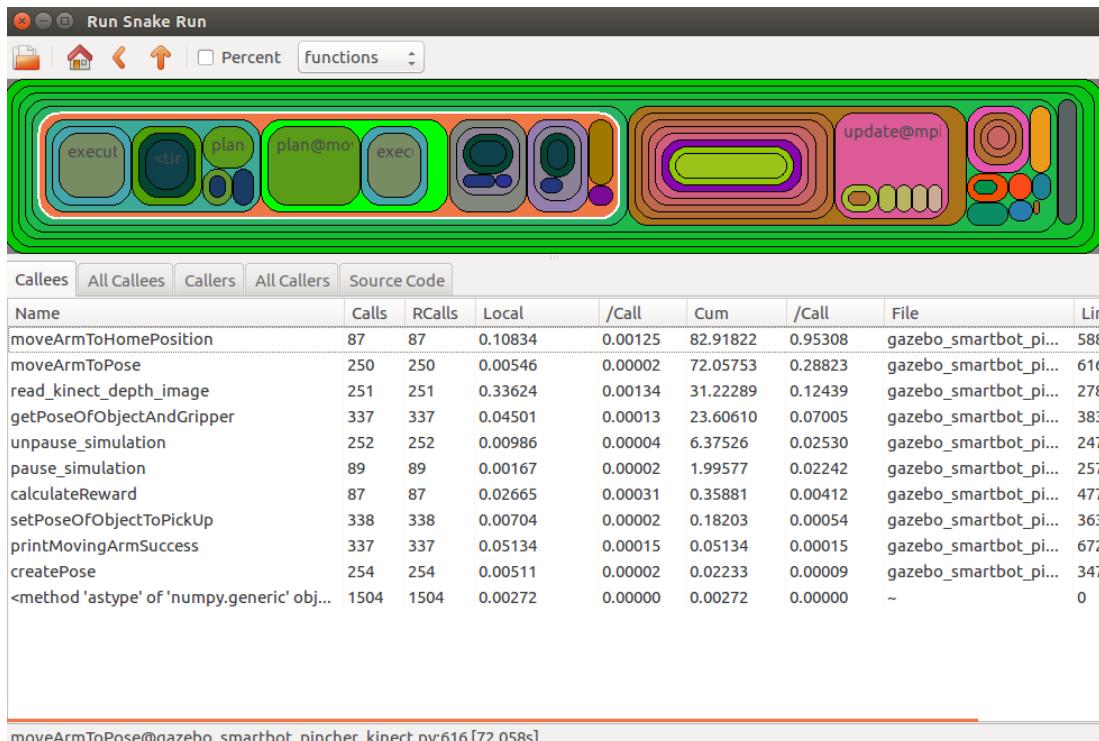


Abbildung 7.3: Laufzeitanalyse eines (verkürzten) Trainings mit dem DDPG-Algorithmus nach der Beschleunigung von MoveIt. Wie zu erkennen ist, ist die benötigte Zeit für das Bewegen des Pinchers deutlich geschrumpft.

```
1 $ mpirun -n 16 python myscript.py
```

16 separate, parallel auf je einem Prozessor-Kern laufende Prozesse gestartet werden, welche aber auch untereinander Daten austauschen können (sofern im Programmcode entsprechend vorgesehen). Dies ist sogar bereits in der DDPG-Implementierung von baseline implementiert. Jedoch funktioniert diese Art der Parallelisierung (noch) nicht für gym-gazebo environments, wie das in diesem Projekt verwendete Pincher-environment eines ist. Für eine (echte) Parallelisierung müssten mehrere Gazebo-Simulationen und ROS-master gestartet werden, was aber aufgrund von ROS nicht möglich ist.

Aufgrund des bereits fortgeschrittenen Zeitbudgets konnten bis zur Abgabe des Projektes diese beiden weiteren möglichen Massnahmen gegen die langen Trainingszeiten leider nicht mehr umgesetzt werden.

8 Fazit

Die Arbeit an diesem Vertiefungsprojekt war sehr lehrreich und interessant. Durch das Aufsetzen des ganzen Systems mit OpenAI Gym, ROS & Gazebo konnte eine anwendbare Grundlage für Reinforcement Learning mit Robotern getestet werden - vor allem für den in diesem Projekt verwendeten Pincher-Arm. Das Projekt konnte sehr gut als Einarbeitung in die verschiedenen Bereiche verwendet werden.

Das Gebiet des Reinforcement Learnings ist überaus spannend. Das Konzept, dass Algorithmen nur anhand der Interaktion mit der Umgebung etwas lernen können ist äusserst faszinierend. Wenn dann auch noch solche Algorithmen teils übermenschliche Performance entwickeln, ist dies umso aufregender.

Neben der Einarbeitung in das Reinforcement Learning konnten auch gewisse Dinge aus der Welt der Robotik gelernt werden. Gerade im Bereich der Roboter-Simulation mit Gazebo und ROS wurde ein guter Einblick gewonnen.

Durch das Nicht-Vorhandensein einer Gazebo-Simulation des Pinchers ging jedoch ein sehr grosser Teil der Projektzeit dadurch verloren - auch für die Einarbeitung in die relativ grossen und komplexen Frameworks wie ROS und Gazebo. Bis nur schon der Arm in der Simulation bewegt werden konnte vergingen mehrere Wochen. Die schlechte Umsetzung des Greifers kostete nochmals mehrere Tage. Deshalb empfiehlt es sich, für ein späteres, ähnliches Projekt wenn möglich einen Roboter zu verwenden, welcher (neben dem physischen Vorhandensein) bereits über eine Gazebo-Simulation verfügt, da dann mehr Zeit für das Reinforcement Learning aufgewendet werden kann.

Das eigentliche Reinforcement Learning schliesslich konnte dann nur während rund zwei Wochen mit dem Pincher-environment ausgetestet werden - inklusive Einarbeitung in die Policy Gradient Algorithmen und deren Implementierungen. Dadurch konnten leider nur wenige RL-Algorithmen überhaupt getestet werden, was zu keinen guten Resultaten für die Performance führte. Nichtsdestotrotz stellt das Projekt eine sehr gute Vorbereitung für die anschliessende Master-Thesis dar, in welcher das Reinforcement Learning wieder angewendet wird.

9 Ausblick

Nach dem Abschluss dieses Projektes gibt es noch verschiedene weitere mögliche Tätigkeiten, welche Vorteile für das Training des RL-Agenten bieten können. Einige Beispiele wären:

- DDPG-Algorithmus verbessern, z.B. durch:
 - Andere Architektur der neuronalen Netze verwenden (z.B. CNN)
 - Preprocessing des Tiefenbildes (State), damit der State-Space kleiner wird
 - Verschiedenste Hyper-Parameter (Lernrate, Action-Noise, Function Approximator etc.) anpassen (dies setzt aber eine immens kürzere Trainings-Zeit voraus)
 - Andere Reward-Berechnung
 - Separates Training eines CNNs, welches anhand des Tiefenbildes die Position des Quaders zurückgibt (und dann dieses trainierte CNN im Training des RL-Agenten verwenden) → ergibt viel kleineren State-Space
 - Evtl. Quader jeden Zeitschritt an eine neue Position setzen
- Performance-Verbesserungen, wie z.B.:
 - Ganzes System mit einer Parallelisierung auf einem High-Performance-Cluster laufen lassen
 - Ganzes System auf einem Rechner mit nativ installiertem Ubuntu, geeigneter Grafikkarte und GPU-Beschleunigung laufen lassen
 - Ein Roboter mit 6-DOF verwenden, da dieser mehrere Positionen erreichen kann als der 4-DOF-Pincher
 - Das Bewegen des Armes mit MoveIt noch weiter versuchen zu beschleunigen
 - Spezifischer IK-Solver für den Pincher einbinden für schnellere Pfadplanung (benötigt aber OpenRave)
 - Gazebo (wenn überhaupt) nur noch für die Tiefenbilder verwenden, das Bewegen des Armes und das Berechnen des Rewards nur in MoveIt selbst (ohne Gazebo-Simulation) ausführen
- Weitere Algorithmen austesten/implementieren, z.B.:
 - HER-Algorithmus [38]
 - ACER-Algorithmus [53]
 - PPO-Algorithmus [54]
 - Cont. Q-Learning [55]
 - ...
- Versuchsaufbau verbessern (Stabilität, Grundplatte, Kamera evtl. höher montieren etc.)
- Simulations-Setup dem realen Versuchsaufbau besser anpassen (z.B. durch das Einfügen der Halterung für die Kinect, und dies auch dem MoveIt mitteilen, damit der reale Pincher nicht in die Halterung fährt)

- Einfacheres Umstellen zwischen Training (mit dem Gazebo-Simulator) und dem Anwenden des Gelernten (mit dem realen Pincher-Arm)
- ...

Abbildungsverzeichnis

1.1	Der autonome Roboter SPOT von Boston Dynamics [1]	1
2.1	Konzept des Reinforcement Learnings [6]	4
2.2	Q-Learning Algorithmus [6]	7
2.3	Windy Gridworld mit Start- und Goal-State	8
2.4	State Value Function der Windy Gridworld	9
2.5	Illustration des DQN-Algorithmus [14]	11
2.6	Pseudocode des DQN-Algorithmus [13]	11
2.7	Illustration der Actor-Critic-Methode [5]	13
2.8	Pseudocode eines Actor-Critic-Algorithmus [6]	14
2.9	Pseudocode des DDPG-Algorithmus [15]	15
3.1	Gazebo-Simulator	18
3.2	PhantomX Pincher Roboter-Arm [19]	19
3.3	Simulations-Setup in Gazebo für den Greifen-Task	21
3.4	Schematische Darstellung der Forward & Inverse Kinematics [22]	22
3.5	Roll-, Pitch- & Yaw-Winkel [23]	22
3.6	Überblick der Software-Komponenten	23
4.1	Von dem Roboter losgelöste Greifer-Backe in Gazebo	29
4.2	Probleme mit dem Greifen in Gazebo	30
4.3	Falsch platzierte Links des Pincher-Greifers	31
4.4	Links des Pincher-Greifers nach der Überarbeitung	32
4.5	Screenshot des MoveIt Setup Assistant	33
5.1	Reward-Berechnung im Pincher-Environment	39
5.2	arm_wrist_flex_link des Pincher-Armes	42
5.3	Tiefenbild der simulierten Kinect-Kamera mit Gausschem Rauschen	44
6.1	realer Versuchsaufbau mit Pincher-Arm und Kinect-Kamera	46
6.2	Verkabelung des Pincher-Armes [41]	48
6.3	Auswahl ROS-Sketchbook in Arduino IDE	49
6.4	USB-Bridge Pincher	49
6.5	USB-Settings in VirtualBox für reale Kinect	50
6.6	USB-Bridge Kinect	51
6.7	In Rviz visualisierte Punktewolke einer realen Kinect-Kamera	51
6.8	Tiefenbild der realen Kinect-Kamera	52
7.1	Action-Noise (links) vs. Parameter-Noise (rechts) [48]	57
7.2	Laufzeitanalyse	60
7.3	Laufzeitanalyse nach Beschleunigung von MoveIt	60

Code Listings

4.1	Ausschnitt aus dem Launch-File des Gazebo-Packages	26
4.2	Ausschnitt aus dem World-File des Gazebo-Packages	27
4.3	Ausschnitt aus dem Haupt-URDF-File des Pinchers	28
4.4	Ausschnitt aus dem Launch-File des Control-Packages	31
4.5	Beispiel-Programm um über die MoveIt Python API den Pincher-Arm zu einer Position zu bewegen	33
5.1	Beispiel-Programm in OpenAI Gym	35
5.2	init-Methode des Pincher-environments	36
5.3	reset-Methode des Pincher-environments	37
5.4	step-Methode des Pincher-environments	37
5.5	Auszug aus der Reward-Berechnung im Pincher-environment	40
5.6	Code für das Berechnen der Greifer-Backen-Positionen & der Pose des Quaders im Pincher-environment (Auszug)	42
5.7	Code für das Einlesen der simulierten Tiefenbilder im Pincher-environment (Auszug)	44
5.8	Definition des State- & Action-Spaces in der init-Methode des Pincher-environments	45
6.1	Ausschnitt aus dem Launch-File des MoveIt-Packages für den realen Pincher-Arm	47
7.1	Ausschnitt aus der Logging des DQN-Algorithmus	55
7.2	Ausschnitt aus dem DDPG-Algorithmus mit der Skalierung der Action-Dimensionen	56

Tabellenverzeichnis

Literatur

- [1] B. Dynamics. (2018). Spot, Adresse: <https://www.bostondynamics.com/spot> (besucht am 08.08.2018).
- [2] M. Gualtieri, A. ten Pas und R. P. Jr., "Category Level Pick and Place Using Deep Reinforcement Learning", *CoRR*, Jg. abs/1707.05615, 2017. arXiv: 1707.05615. Adresse: <http://arxiv.org/abs/1707.05615>.
- [3] (2018). Spiel-Komplexität, Adresse: <https://de.wikipedia.org/wiki/Spiel-Komplexit%C3%A4t> (besucht am 08.08.2018).
- [4] M. Reynolds, "DeepMind's AI beats world's best Go player in latest face-off", *NewScientist*, 23. Mai 2017. Adresse: <https://www.newscientist.com/article/2132086-deepminds-ai-beats-worlds-best-go-player-in-latest-face-off/> (besucht am 09.08.2018).
- [5] R. S. Sutton und A. G. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998. Adresse: <http://www.cs.ualberta.ca/~sutton/book/the-book.html>.
- [6] D. Silver. (2015). UCL Course on RL, Adresse: <http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html> (besucht am 08.08.2018).
- [7] C. Nicholls. (2016). Chris explains machine learning to himself, Adresse: <https://cgnicholls.github.io/> (besucht am 08.08.2018).
- [8] S. Huang. (2018). Introduction to Various Reinforcement Learning Algorithms, Adresse: <https://towardsdatascience.com/introduction-to-various-reinforcement-learning-algorithms-i-q-learning-sarsa-dqn-ddpg-72a5e0cb6287> (besucht am 08.08.2018).
- [9] V. M. Vilches. (2018). Basic Reinforcement Learning, Adresse: https://github.com/vmayoral/basic_reinforcement_learning (besucht am 08.08.2018).
- [10] H. Kim und J. Kim. (2018). Awesome Reinforcement Learning, Adresse: <https://github.com/aikorea/awesome-rl> (besucht am 08.08.2018).
- [11] V. M. Vilches. (2016). Reinforcement learning in robotics, Adresse: https://vmayoral.github.io/robots/_ai/_deep_learning/_rl/_reinforcement_learning/2016/07/06/rl-intro/ (besucht am 08.08.2018).
- [12] J. Weatherwax. (2015). Code For Various Figures and Problems, Adresse: https://waxworksmath.com/Authors/N_Z/Sutton/sutton.html (besucht am 14.08.2018).
- [13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra und M. A. Riedmiller, "Playing Atari with Deep Reinforcement Learning", *CoRR*, Jg. abs/1312.5602, 2013. arXiv: 1312.5602. Adresse: <http://arxiv.org/abs/1312.5602>.
- [14] H. Mao, M. Alizadeh, I. Menache und S. Kandula, "Resource Management with Deep Reinforcement Learning", in *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, Ser. HotNets '16, Atlanta, GA, USA: ACM, 2016, S. 50–56, ISBN: 978-1-4503-4661-0. DOI: 10.1145/3005745.3005750. Adresse: <http://doi.acm.org/10.1145/3005745.3005750>.
- [15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver und D. Wierstra, "Continuous control with deep reinforcement learning", *CoRR*, Jg. abs/1509.02971, 2015. arXiv: 1509.02971. Adresse: <http://arxiv.org/abs/1509.02971>.
- [16] P. Emami. (2016). Deep Deterministic Policy Gradients in TensorFlow, Adresse: <https://pemami4911.github.io/blog/2016/08/21/ddpg-rl.html> (besucht am 09.08.2018).
- [17] F.-F. Li, J. Johnson und S. Yeung. (2017), Adresse: <https://www.youtube.com/watch?v=lvoHnicueoE&feature=youtu.be&t=1676> (besucht am 09.08.2018).
- [18] J. S. Simon und M. Ferguson. (2017). TurtleBot Arm Github Repository, Adresse: https://github.com/turtlebot/turtlebot_arm (besucht am 08.08.2018).

- [19] T. Robotics. (2018). PhantomX Pincher Robot Arm Kit Mark II - Turtlebot Arm, Adresse: <https://www.trossenrobotics.com/p/PhantomX-Pincher-Robot-Arm.aspx> (besucht am 09.08.2018).
- [20] P. N.V. (2017). Pickt, Adresse: <https://www.pickit3d.com/> (besucht am 09.08.2018).
- [21] I. A. Sucan und S. Chitta. (2018). MoveIt!, Adresse: <https://moveit.ros.org/> (besucht am 09.08.2018).
- [22] S. Kucuk und Z. Bingul. (2018), Adresse: https://www.researchgate.net/figure/The-schematic-representation-of-forward-and-inverse-kinematics_fig1_4106039 (besucht am 08.08.2018).
- [23] K. Ellis. (2018), Adresse: https://www.researchgate.net/figure/Average-roll-pitch-and-yaw-angles_fig2_262055313 (besucht am 08.08.2018).
- [24] I. Zamora, N. G. Lopez, V. M. Vilches und A. H. Cordero, "Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo", *arXiv preprint arXiv:1608.05742*, 2016.
- [25] M. Ilg, "Einführung in ROS", NTB Interstaatliche Hochschule für Technik Buchs, 2017.
- [26] A. Kalberer, "MME Simulationsprojekt: ROS und Gazebo", NTB Interstaatliche Hochschule für Technik Buchs, 2016.
- [27] O. S. R. Foundation. (2018). ROS Documentation, Adresse: <http://wiki.ros.org/> (besucht am 09.08.2018).
- [28] ——, (2018). ROS Tutorials, Adresse: <http://wiki.ros.org/ROS/Tutorials> (besucht am 09.08.2018).
- [29] ——, (2018). Gazebo, Adresse: <http://gazebosim.org/> (besucht am 09.08.2018).
- [30] ——, (2018). Gazebo, Adresse: <http://gazebosim.org/tutorials> (besucht am 09.08.2018).
- [31] Gazebo. (2014). Gazebo Tutorials: Connect to ROS, Adresse: http://gazebosim.org/tutorials?cat=connect_ros (besucht am 08.08.2018).
- [32] M. Ferguson. (2015), Adresse: <https://answers.ros.org/question/201625/turtlebot-arm-on-gazebo-lets-fall-all-grasped-objects/?answer=201692#post-id-201692> (besucht am 08.08.2018).
- [33] Z. Anderson. (2017). Cleaned up pincher gripper description, Adresse: https://github.com/turtlebot/turtlebot_arm/pull/24 (besucht am 08.08.2018).
- [34] Gazebo. (2014). ROS Depth Camera Integration, Adresse: http://gazebosim.org/tutorials?tut=ros_depth_camera&cat=connect_ros (besucht am 08.08.2018).
- [35] ROS. (2018). ROS Image Message, Adresse: http://docs.ros.org/kinetic/api/sensor_msgs/html/msg/Image.html (besucht am 08.08.2018).
- [36] erlerobot. (2018). Github Repository Gym-Gazebo, Adresse: <https://github.com/erlerobot/gym-gazebo> (besucht am 08.08.2018).
- [37] S.-U. Empy2. (2015). Check if a point is inside a rectangular shaped area (3D), Adresse: <https://math.stackexchange.com/questions/1472049/check-if-a-point-is-inside-a-rectangular-shaped-area-3d> (besucht am 08.08.2018).
- [38] M. Andrychowicz, F. Wolski, A. Ray, J. Schneider, R. Fong, P. Welinder, B. McGrew, J. Tobin, P. Abbeel und W. Zaremba, "Hindsight Experience Replay", *CoRR*, Jg. abs/1707.01495, 2017. arXiv: 1707.01495. Adresse: <http://arxiv.org/abs/1707.01495>.
- [39] F. Saccilotto, "Innenraumerfassung mit dem Turtlebot", NTB Interstaatliche Hochschule für Technik Buchs, 2013.

- [40] S.-U. corb555. (2014). Arbotix Drivers, Adresse: https://github.com/corb555/arbotix_ros (besucht am 08.08.2018).
- [41] Arbotix. (2018). ArbotiX-M Robocontroller Getting Started Guide, Adresse: <http://learn.trossenrobotics.com/arbotix/7-arbotix-quick-start-guide> (besucht am 08.08.2018).
- [42] W. Meeussen. (2012). ROS-Package openni, Adresse: <http://wiki.ros.org/openni> (besucht am 08.08.2018).
- [43] H. Martin, J. Blake und K. Machulis. (2012). ROS-Package freenect, Adresse: <http://wiki.ros.org/libfreenect> (besucht am 08.08.2018).
- [44] Oracle. (2018). VirtualBox 5.2.16 Oracle VM VirtualBox Extension Pack, Adresse: <https://www.virtualbox.org/wiki/Downloads> (besucht am 08.08.2018).
- [45] V. M. Vilches. (2016). DQN Code, Adresse: https://github.com/vmayoral/basic_reinforcement_learning/blob/master/tutorial5/dqn-cartpole.py (besucht am 09.08.2018).
- [46] Tensorflow. (2018). Tensorflow, Adresse: <https://www.tensorflow.org/> (besucht am 09.08.2018).
- [47] OpenAI. (2018). Baselines, Adresse: <https://github.com/openai/baselines> (besucht am 09.08.2018).
- [48] ——, (2017). Better exploration with parameter noise, Adresse: <https://blog.openai.com/better-exploration-with-parameter-noise/> (besucht am 09.08.2018).
- [49] P. Emami. (2018). deep-rl, Adresse: <https://github.com/pemami4911/deep-rl/tree/master/ddpg> (besucht am 09.08.2018).
- [50] M. Plappert, *keras-rl*, <https://github.com/keras-rl/keras-rl>, 2016.
- [51] J. Lutz. (2018). Saving and Restoring DDPG model, Adresse: <https://github.com/openai/baselines/issues/162#issuecomment-408349221> (besucht am 09.08.2018).
- [52] J. Nordmoen. (2017). Possible race condition in Event.hh causing Segmentation Fault, Adresse: <https://bitbucket.org/osrf/gazebo/issues/2332/possible-race-condition-in-eventhh-causing> (besucht am 16.08.2018).
- [53] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu und N. de Freitas, "Sample Efficient Actor-Critic with Experience Replay", *CoRR*, Jg. abs/1611.01224, 2016. arXiv: 1611.01224. Adresse: <http://arxiv.org/abs/1611.01224>.
- [54] J. Schulman, F. Wolski, P. Dhariwal, A. Radford und O. Klimov, "Proximal Policy Optimization Algorithms", *CoRR*, Jg. abs/1707.06347, 2017. arXiv: 1707.06347. Adresse: <http://arxiv.org/abs/1707.06347>.
- [55] S. Gu, T. P. Lillicrap, I. Sutskever und S. Levine, "Continuous Deep Q-Learning with Model-based Acceleration", *CoRR*, Jg. abs/1603.00748, 2016. arXiv: 1603.00748. Adresse: <http://arxiv.org/abs/1603.00748>.

Eigenständigkeitserklärung

Hiermit erkläre ich, die vorliegende Arbeit vollumfänglich eigenständig verfasst zu haben und dabei keine nicht im Literaturverzeichnis aufgeführten Quellen benutzt zu haben.

Buchs, den 18.08.2018



Joël Lutz

Digitaler Anhang

Der digitale Anhang wird als Daten-CD oder Zip-Archiv mitgeliefert. Er ist aber auch unter <https://github.com/joellutz/MSE-VT2-SmartBot-Anhang> zu finden. Die virtuelle Ubuntu-Maschine, welche für diese Arbeit hauptsächlich verwendet wurde, kann auf Anfrage zur Verfügung gestellt werden.

- Dokumentation
 - Dokumentation
 - Projektplan
 - Zwischenpräsentation
 - Abschlusspräsentation
- Einführung in ROS & Gazebo
 - MSE-Vertiefungsprojekt 1 „Einführung in ROS“ von Manuel Ilg
 - „MME-Simulationsprojekt: ROS und Gazebo“ von Andreas Kalberer
- MATLAB-Code des Windy Gridworld-Beispiels
- MSE-Vertiefungsprojekt „Innenraumerfassung mit dem Turtlebot“ von Fabian Saccilotto
- Setup-Skript für das ganze System mit ROS, Gazebo, OpenAI Gym, Code der RL-Algorithmen etc.

