Masterthesis

Simulations-Realitäts-Transfer – Lernen von gezieltem Zurückspielen auf einem Ballspielroboter

Johannes Hackbarth

2 7 8 8 6 4 5

joha@informatik.uni-bremen.de

29. Juni 2018

Erstgutachter	Prof. Dr. Udo Frese
Zweitgutachter	Prof. Dr. Rolf Drechsler

Fachbereich 03: Informatik/Mathematik

Eidesstattliche Erklärung

Hiermit versichere ich, die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der angegebenen Quellen und Hilfsmittel verfasst zu haben. Die aus fremden Quellen direkt oder indirekt übernommenen Stellen sind als solche kenntlich gemacht.

Ich bestätige außerdem, dass die vorliegende Arbeit noch nicht im Rahmen eines anderen Prüfungsverfahrens eingereicht wurde.

Ort, Datum

Johannes Hackbarth

ZUSAMMENFASSUNG

Ein Verhalten ausschließlich in einer Simulation zu trainieren, hat verschiedene Vorteile. So kann im Voraus für ein robotisches System gefährliches Verhalten identifiziert und vermieden werden. Auch die Verschleißerscheinungen bei realen Systemen spielen in einer Simulation keine Rolle. Daher erscheint es sinnvoll ein Verhalten vorab in einer Simulation zu trainieren und im Anschluss nahtlos auf das reale System zu übertragen. Aktuelle Fortschritte im Bereich des Deep Reinforcement Learnings haben hier vielversprechende Möglichkeiten aufgezeigt. Weiterhin wurden Ballspielverhalten bisher zumeist mit dem Vorwissen eines Experten im Kontext des Imitationslernens trainiert. In dieser Arbeit soll ein Ballspielverhalten ohne Vorwissen, welches gezielt Bälle auf ein Ziel spielen kann, auf dem Unterhaltungsroboter Doggy exklusiv in einer Simulation trainiert werden und im Anschluss auf das echte System. Hierfür werden die aktuellen Fortschritte im Bereich des Deep Reinforcement Learnings und im Bereich der Domain Randomisation untersucht und auf das gegebene Problem angewendet. Darauf aufbauend werden verschiedene Ballspielverhalten in der Simulation trainiert. Diese Verhalten werden dann auf dem realen System ausgerollt und auf ihre Übertragbarkeit hin untersucht.

Abstract

Training a behaviour exclusively in a simulation has several advantages. This way dangerous behaviour for a robotic system can be identified and avoided in advance. The signs of wear in real systems also play no role in a simulation. Hence it makes sense to train a behaviour in advance in a simulation and then seamlessly transfer it to the real system. Current advances in the field of deep reinforcement Learnings have shown promising possibilities here. Furthermore, up to now ball play behaviour has mostly been trained with the previous knowledge of an expert in the context of imitation learning. In this work a ball play behaviour without previous knowledge, which can play balls on a target, is to be trained exclusively in a simulation on the entertainment robot Doggy. This is then to be used on the real system. For this purpose, the current advances in the field of deep reinforcement learning in connection with the randomisation of simulation domains is examined and applied to the given problem. Building on this, various ball game behaviours are trained in the simulation. These behaviours are then rolled out on the real system and examined for their transferability.

Inhaltsverzeichnis

Listingverzeichnis **IX**

Abbildungsverzeichnis **XI**

Tabellenverzeichnis **XIII**

1 Einleitung **1**
1.1 Motivation 1
1.2 Zielsetzung 2
1.3 Aufbau der Arbeit 2

2 Grundlagen **3**
2.1 Reinforcement Learning 3
2.1.1 Markov Decision Process 4
2.1.2 Value-based Reinforcement Learning 5
2.1.3 Q-Learning 6
2.1.4 Policy Search 6
2.1.4.1 Policy Gradient 6
2.1.4.2 Actor-Critic 7
2.2 Neuronale Netze und Deep Learning 8
2.3 Deep Reinforcement Learning 10
2.3.1 Deep Deterministic Policy Gradient (DDPG) 10
2.3.2 Recurrent Deterministic Policy Gradient (RDPG) 12
2.4 Verwendetes System 14

3 Stand der Technik **15**
3.1 Domain Randomisation 15
3.1.1 Randomisierung der Dynamikparameter 15
3.1.2 Perzeptive Ansätze 17

4 Ansatz **21**
4.1 Verwendetes Verfahren 21
4.1.1 Identifikation der relevanten Dynamikparameter 21
4.1.2 Verwendeter Algorithmus 23
4.1.3 Rewardfunktion 24
4.1.4 Exploration des Aktionsraums 27

5 Umsetzung **31**
5.1 Stand der Software 31
5.2 Erstellen einer Simulationsumgebung 32
5.2.1 Interaktion zwischen dem Agenten und der Simulation 36
5.3 Gesamtsystem 38

6 Experimente **39**
6.1 Simulationsszenarien 39
6.1.1 Definition Trainingsszenario 39
6.1.2 Definition Testszenario 41
6.2 Ergebnisse der Simulationsexperimente 42
6.2.1 Szenario Zieldistanz 3,5 m 42
6.2.1.1 Training 42
6.2.1.2 Test - finale Policy 44
6.2.1.3 Test - beste Policy 47
6.2.2 Szenario Zieldistanz 4 m 49
6.2.2.1 Training 49
6.2.2.2 Test - finale Policy 51
6.2.2.3 Test - beste Policy 54
6.2.3 Untersuchung der gelernten Schlagbewegungen 56
6.3 Zwischenfazit 60
6.4 Realsystemszenario 61
6.5 Ergebnisse der Experimente auf dem realen System 62
6.5.1 Qualitative Analyse der Schlagbewegungen 62
6.5.2 Untersuchung der Abweichung zur Simulation im Schlag 70
6.5.2.1 Experimente mit der zwei Achsenpolicy 71
6.5.2.2 Experimente mit der drei Achsenpolicy 75
6.6 Fazit 79

7 Zusammenfassung **81**
7.1 Ausblick 82

Glossar **83**

Akronyme **84**

Literatur **85**

Anhang **89**
A CD 90

Listingverzeichnis

5.1 Gekürzter Auszug aus der Konfigurationsdatei der Simulationswelt mit den Modellen *Doggy* und *Ball*, ihren Plugins sowie deren Konfigurationsmöglichkeiten 36
5.2 Gekürzter Python-Pseudocode der DoggySim-v1 Umgebung 37

Abbildungsverzeichnis

1.1 Der Unterhaltungsroboter *Doggy* 1

2.1 Interaktion zwischen dem Agenten und der Umgebung zu einem Zeitpunkt t mit einer Aktion a_t und dem Feedback der Umgebung als Umgebungszustand s_{t+1} und Reward r_{t+1} [SB98] 3
2.2 Die *Actor-Critic* Architektur [Aru+17] 8
2.3 Ein einfaches neuronales Netz mit einem Hidden-Layer 9
2.4 Darstellung des Ballspielroboters Doggy. Links mit Kostüm, daneben ohne Kostüm, eine schematische Darstellung der Koordinatensysteme und Achsen und rechts eine schematische Darstellung der Achsen. [SWF16] 14

3.1 Eine Policy auf dem Fetch Arm [Pen+17] 16
3.2 Simulierte, mit dem Generatornetz adaptierte und reale Bilder [Bou+17] 17
3.3 Das GrapGAN-Netz [Bou+17] 18
3.4 Generierte, randomisierte Bilder in der Simulation und das reale Szenario [Tob+17] 19

4.1 Layer für den *Deep Deterministic Policy Gradient (DDPG)*- und *Recurrent Deterministic Policy Gradient (RDPG)*-Agenten, links der Actor, rechts der Critic 23
4.2 Negativer Exponent 25
4.3 Vergleich der Bewegungen des Schlägers mit Rewardfunktion ohne und mit Zeitkomponente 27
4.4 Vergleich der Wirkungsbereiche von reinem Aktionsrauschen und Parameterraumrauschen [Pla17] 28

5.1 Beispiel eines erkannten in kräftigem rot Balls mit Vorhersage der Ballflugbahn in grün 31
5.2 Zustand der Softwarekomponenten auf dem PC im Demobetrieb 32
5.3 Doggy in der Simulationsumgebung mit Ball in einer Startpose und dem Zielobjekt. 33
5.4 Übersicht über die Nodes und Topics in einer Simulationsumgebung mit einer Instanz und einer Lernumgebung 35
5.5 Übersicht über das Zusammenspiel vom Agenten mit der Simulation . . 38

6.1 Durchschnittliche über alle Episoden. Unterteilt in die verschiedenen Bewegungsmöglichkeiten: Balltreffer und Zieltreffer im Szenario 3,5 m . . . 43
6.2 Durchschnittlicher Reward in Zeitschritten einer Simulationsinstanz, leicht geglättet mit einem Fenster von 250 Zeitschritten 44
6.3 Trefferraten bei 3,5 m Zielentfernung 45
6.4 Position der Bälle am Episodenende 47
6.5 Trefferraten bei 3,5 m Zielentfernung mit der jeweils besten Policy 48
6.6 Durchschnittliche über alle Episoden. Unterteilt in die verschiedenen Bewegungsmöglichkeiten: Balltreffer und Zieltreffer im Szenario Szenario 4 m 50
6.7 Durchschnittlicher Reward in Zeitschritten einer Simulationsinstanz, leicht geglättet mit einem Fenster von 250 Zeitschritten 50
6.8 Trefferraten bei 4 m Zielentfernung 52
6.9 Position der Bälle am Episodenende 54
6.10 Trefferraten bei 4 m Zielentfernung mit der jeweils besten Policy 55
6.11 Vergleich dreier Schlagbewegungen mit ähnlicher, mittiger Ballflugbahn 58
6.12 Vergleich dreier Schlagbewegungen mit ähnlicher, außen rechts liegender Ballflugbahn 60
6.13 Beispiele von real ausgeführten Schlagbewegungen 62
6.14 Vergleich der realen und simulierten Schlagbewegung bei erfolgreicher Ausführung 64
6.15 Vergleich der realen und simulierten Schlagbewegung bei nicht erfolgreicher Ausführung 66
6.16 Vergleich der realen und simulierten Schlagbewegung bei erfolgreicher Ausführung 68
6.17 Vergleich der realen und simulierten Schlagbewegung bei nicht erfolgreicher Ausführung 70
6.18 Schlägerpositionen in Simulation und Realität sowie deren Abweichungen mit der zwei Achsenpolicy 72
6.19 Gelenkpositionen in Simulation und Realität sowie deren Abweichungen mit der zwei Achsenpolicy 73
6.20 Gelenkgeschwidigkeiten in Simulation und Realität sowie deren Abweichungen. 74
6.21 Schlägerpositionen in Simulation und Realität sowie deren Abweichungen mit der drei Achsenpolicy 76
6.22 Gelenkpositionen in Simulation und Realität sowie deren Abweichungen mit der drei Achsenpolicy 77
6.23 Gelenkgeschwindigkeiten in Simulation und Realität sowie deren Abweichungen mit der drei Achsenpolicy 78

Tabellenverzeichnis

4.1 Identifizierte relevante Dynamikparameter 22

5.1 Parameter der Links 33
5.2 Parameter der Gelenke 33

6.1 Bereiche der Randomisierung der Ballwurfparameter 39
6.2 Bereiche der Randomisierung der Dynamikparameter 40
6.3 Hyperparameter für die Experimente 41
6.4 Vergleich der Trefferraten von finaler zu bester Policy im 3,5 m Szenario 49
6.5 Vergleich der Trefferraten von finaler zu bester Policy im 4 m Szenario . 56

1 Einleitung

1.1 Motivation

Die Lücke zwischen Simulation und Realität zu überbrücken, ist eines der grundlegenden Probleme der Robotik. In einer simulierten Umgebung lassen sich vielfältige Ansätze und Probleme testen und evaluieren, ohne auf das echte System angewiesen zu sein. Dabei können insbesondere problematische Verhalten eines robotischen Systems im Voraus identifiziert und vermieden werden. Weiterhin können eventuelle Hardwareausfälle für den Moment umgangen werden, da ein Problem abhängig von der Abstraktionsebene auch ohne direkten Zugriff auf die Hardware weiter bearbeitet werden kann.
Für Lernalgorithmen ist die Generierung von ausreichend vielen Beispielen von äußerster Wichtigkeit. Diese lassen sich in großer Zahl ohne größeren Aufwand in einer Simulation generieren, wohingegen ein Training ausschließlich auf einem echten System einen großen Zeitaufwand, sowie insbesondere die Gefahr einer Beschädigung durch ein nicht optimales Verhalten in sich birgt.

Abbildung 1.1: Der Unterhaltungsroboter *Doggy*

Die größte Schwierigkeit bei einem ausschließlich in einer Simulation gelernten Verhalten ist die Übertragbarkeit in die Realität. Eine Simulation kann nur ansatzweise die reale Welt abbilden und birgt an diversen Stellen Unschärfen in der Abbildung der selbigen, beispielsweise im Verhalten eines abprallenden Balls. Eine gängige Physik-Engine kann hier nur eine begrenzte Menge Parameter bieten, um diesen Prozess zu modellieren und in akzeptabler Zeit zu simulieren. Durch diese Limitationen kann ein, in einer Simulationsumgebung gelerntes Verhalten, in der Realität scheitern, da sich durch die Grenzen der Simulation für den Roboter Probleme ergeben, die sein Verhalten nicht behandeln kann.

Dieses Problem soll auf dem Unterhaltungsroboter *Doggy* (siehe Abbildung 1.1) gelernt werden. *Doggy* ist ein in der Arbeitsgruppe Multisensorische-Interaktive Systeme von Prof. Dr. Udo Frese et al. entwickelter Ballspielroboter. Er kann ihm zugeworfene Bälle wahrnehmen, verfolgen und aus diesen Daten eine Schlagbewegung vollführen. Leider ist es ihm aktuell noch nicht möglich Bälle gezielt zu schlagen. Stattdessen versucht er nur so schnell wie möglich an eine vorhergesagte Schlagposition zu kommen und den Ball dann nach Möglichkeit zu treffen, was teils auch misslingt.

1.2 Zielsetzung

Ziel der Arbeit ist es ein möglichst realitätsnahes Szenario zu entwickeln, in dem ein Ballspielroboter lernt, Bälle gezielt zu einer Position zurückzuspielen. Dafür soll der verwendete Roboter *Doggy* dieses Verhalten zunächst in einer Simulationsumgebung erlernen und anschließend das Erlernte in die reale Welt übertragen.

1.3 Aufbau der Arbeit

Einleitend wird ein Einblick in die Grundlagen des *Reinforcement Learnings* und darauf aufbauend in die aktuellen Entwicklungen im *Deep Reinforcement Learning* gegeben. Im Anschluss daran werden bestehende Algorithmen und Verfahren untersucht, mit denen einerseits ein schneller Lernerfolg, andererseits ein Übertragen von Ergebnissen aus einer Simulation ermöglicht wird.
Aus diesen Erkenntnissen wird ein Ansatz entwickelt, mit dem das gegebene System sowie die Aufgabenstellung in einer Simulation abgebildet und gelernt werden kann. Darauf aufbauend wird erörtert, wie dieser Ansatz in die Realität übertragen werden kann. Mit dieser Methode werden Experimente in Simulation und auf dem realen System durchgeführt, um die genutzten Verfahren zu testen und zu bewerten.
Abschließend werden die Ergebnisse zusammengefasst und ein Ausblick auf künftige Arbeiten, die sich aus den Experimenten ergeben, gegeben.

2 Grundlagen

In diesem Kapitel wird ein Überblick über maschinelles Lernen (*Machine Learning*) und im Speziellen über *Reinforcement Learning* und dessen Unterkategorie *Deep Reinforcement Learning* gegeben. Dafür werden die grundlegenden Ideen hinter dem *Reinforcement Learning* erklärt. Darauf aufbauend wird die Erweiterung des *Deep Reinforcement Learning* erläutert. In diesem Abschnitt werden auch einige aktuelle Algorithmen vorgestellt. Im Folgenden werden jeweils die Fachbegriffe erläutert.

2.1 Reinforcement Learning

Reinforcement Learning behandelt eine Unterkategorie des maschinellen Lernens, in der es gilt einem Agenten eine definierte Aufgabe erlernen zu lassen. Dabei kann der Agent mit einer Menge von Aktionen mit der Welt interagieren. Aus der Welt wiederum erhält der Agent Feedback in Form des Weltzustands, der entweder vollständig beobachtbar (*fully-observable*) oder nur teilweise beobachtbar (*partially-observable*) sein kann. Damit der Agent aus seinen Aktionen lernen kann, erhält er für diese eine Belohnung (*Reward*), welche von einem entsprechenden Algorithmus verarbeitet werden kann, um Rückschlüsse auf den Erfolg der jeweiligen Aktion ziehen zu können.

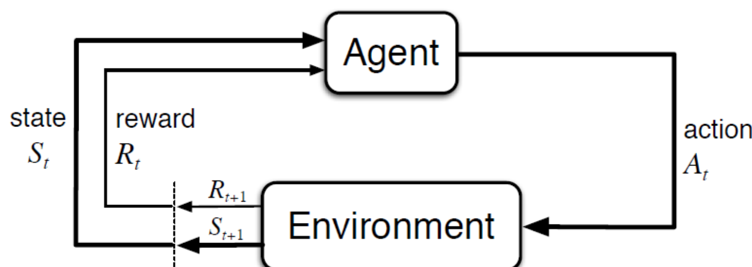


Abbildung 2.1: Interaktion zwischen dem Agenten und der Umgebung zu einem Zeitpunkt t mit einer Aktion a_t und dem Feedback der Umgebung als Umgebungszustand s_{t+1} und Reward r_{t+1} [SB98]

Dafür exploriert der Agent selbständig den Zustandsraum mit den ihm zur Verfügung stehenden Aktionen. Währenddessen wird ebenfalls der Reward gespeichert, um nach dem Erreichen oder nicht Erreichen des Ziels Rückschlüsse daraus zu ziehen.

2.1.1 Markov Decision Process

Formal liegt dem Lernproblem der *Markov Decision Process (MDP)* zu Grunde [SB98]. Ein *MDP* ist formal als $M = (\mathcal{S}, \mathcal{A}, \mathcal{P}, p_0, r)$ mit dem Zustandsraum $\mathcal{S}$ und dem Aktionsraum $\mathcal{A}$ definiert. Der Reward r wird dabei als Funktion aus $r = (s_t, a_t)$ definiert, der zu einem Zeitpunkt t erreicht wird. $\mathcal{P}$ ist die Zustandsübergangswahrscheinlichkeitsverteilung, welche beschreibt wie wahrscheinlich es ist, in einem Zustand s_t gegeben einer Aktion a_t einen Folgezustand s_{t+1} zu erreichen, formal $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \mapsto [0, 1]$.
Ein Agent kann wie eingangs erwähnt zu einem Zeitschritt t auf Basis der Wahrnehmung des Umgebungszustands $s_t \in S$ eine Aktion $a_t \in A(s)$ wählen. Nach Ausführung der Aktion a_t erhält der Agent einen Reward $r_{t+1} \in R$ und er befindet sich in einem neuen Zustand $s_{t+1} \in S$. Dieses Verhaltensprinzip wird in Grafik 2.1 illustriert. Aus dieser Abfolge von Aktionen ergibt sich eine Trajektorie τ der Form $\tau = (s_0, a_0, r_1, s_1, a_1, \ldots s_t, r_t)$, wobei s_t und r_t Terminalzustände bezeichnen. Eine solche abgeschlossene Trajektorie wird auch als Rollout bezeichnet. Das übergeordnete Ziel des Agenten ist es eine Strategie (*Policy*) π zu entwickeln, welche die Aufgabe löst und damit die Summe der Rewards J langfristig maximiert. Eine Policy wird definiert als die bedingte Wahrscheinlichkeit eine Aktion $a \in A(s)$ gegeben einen Zustand $s \in S$, also $\pi(a|s)$.

Das Lernproblem ist demnach das Finden einer optimalen Policy π^*, welche die Summe der Rewards maximiert.

$$\pi^* = \underset{\pi}{\operatorname{argmax}}\ J_\pi \tag{2.1}$$

Dabei ist J_π erwartete Summe der Rewards einer Trajektorie τ gegeben einer Policy π.

$$J_\pi = \mathbb{E}[R(\tau)|\pi] = \int R(\tau) p_\pi(\tau) d\tau \tag{2.2}$$

wobei $p_\pi(\tau)$ die Verteilung der Trajektorien gegeben einer Policy π ist.

Bei der Betrachtung der Erwartungswerte der langfristigen Rewards sind drei Modelle [SB98] zu erwähnen. Einmal das *Finite Horizon Model*, in dem der Agent die Summe der erwarteten Rewards über die nächsten N Zeitschritte maximiert

$$\mathbb{E}[R_t] = \mathbb{E}\left[\sum_{k=1}^{N} r_{t+k}\right] \tag{2.3}$$

Das *Finite Horizon Model* betrachtet nur eine begrenzte Anzahl zukünftiger Zeitschritte. Sind aber alle Zeitschritte relevant für den Lernprozess wird das *Infinite Horizon Discounted Model* angewendet. Dabei werden alle zukünftigen Zeitschritte mit einem Discountfaktor γ, wie in 2.3 betrachtet. Der Discountfaktor γ gibt dabei an, wie wichtig

die langfristige Reawrds sind. Ein Faktor von $\gamma = 0$ betrachtet nur den aktuellen Reward und $\gamma = 1$ misst den langfristigen Rewards die höchste Bedeutung zu.

$$\mathbb{E}[R_t] = \mathbb{E}\left[\sum_{k=1}^{\infty} \gamma^{k-1} r_{t+k}\right] \tag{2.4}$$

Zusätzlich gibt es noch das *Average Reward Model*, bei dem der durchschnittliche Langzeit-Reward maximiert wird.

$$\mathbb{E}[R_t] = \lim_{N \to \infty} \mathbb{E}\left[\frac{1}{N} \sum_{k=1}^{N} r_{t+k}\right] \tag{2.5}$$

2.1.2 Value-based Reinforcement Learning

Der Grundgedanke im *Reinforcement Learning* ist die Maximierung des Rewards. Da die echte Reward-Funktion dem Agenten meist nicht bekannt ist, gilt es diese zu approximieren. Ein Ansatz, den eine Vielzahl von Reinforcement-Learning-Algorithmen dafür verwenden, ist es den langfristigen Reward mit der Value-Funktion abzuschätzen. Dabei wird auf Basis eines Zustands s gegeben einer Policy π der zu erwartende Reward geschätzt.

$$V^{\pi}(s) = \mathbb{E}_{\pi}[R|s] \tag{2.6}$$

Analog zur Value-Funktion, die nur den aktuellen Zustand betrachtet, kann die Q-Funktion, auch als Action-Value-Funktion bezeichnet, definiert werden. Diese beachtet zusätzlich zum aktuellen Zustand s auch die auszuführende Aktion a gegeben der Policy π.

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}[R|s, a] \tag{2.7}$$

Aus den vorherigen Ausführungen ergibt sich das übergeordnete Ziel des *Reinforcement Learnings* eine optimale Policy π^*. Um diese zu finden, wird im Value-basierten *Reinforcement Learning* die Maximierung der Value-Funktion $V(s)$ sowie der Action-Value-Funktion $Q(s, a)$ angestrebt.

$$V^*(s) = \max_{\pi} V^{\pi}(s) \tag{2.8}$$

$$Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a) \tag{2.9}$$

2.1.3 Q-Learning

Bei einem reinem Value-Funktionsbasiertem Lernen werden zumeist Ansätze aus dem Bereich des *Dynamic Programmings* genutzt. Da sich die optimale Value-Funktion meist nicht direkt schätzen lässt, weil dies umfassendes Wissen des Agenten über die Dynamik der Umgebung erfordern würde, wird zumeist die Q-Funktion gelernt, welche mithilfe der Bellman-Gleichung [Bel52] die *Markov Eigenschaft* des Problems in folgender Form ausnutzt.

$$Q^\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}(r_{t+1} + \gamma Q^\pi(s_{t+1}, \pi(s_{t+1}))) \tag{2.10}$$

Damit kann Q^π mittels Bootstrapping verbessert werden. Das ist dadurch möglich, da der Agent Wissen über seine Aktionen und den Zustand der Umgebung zum Zeitpunkt t hat. Der Agent kann also die aktuellen Werte nutzen, um seine Schätzung der von Q^π zu verbessern.

Der bekannteste Algorithmus, der diese Eigenschaft nutzt, ist das *Q-Learning*. Hierbei wird über die eingangs erwähnten Trajektorien τ bestehend aus Tupeln der Form (s_t, a_t, r_t, s_{t+1}) der *Temporal Difference (TD)* Fehler berechnet [SB98].

$$\delta_t = r_t + \gamma \max_a (Q(s_{t+1}, a) - Q(s_t, a_t)) \tag{2.11}$$

Mit der Updateregel kann iterativ die Schätzung von Q verbessert werden.

$$Q(s, a) = Q(s_t, a_t) + \alpha\delta_t \tag{2.12}$$

Wobei $0 \leq \alpha \leq 1$ die Lernrate ist, mit der die Schätzung von Q aktualisiert wird. Damit ist eine Konvergenz zur optimalen Q-Funktion gegeben.

2.1.4 Policy Search

Ein Ansatz, der ohne die Schätzung der Q-Funktion oder der Value-Funktion auskommt, ist die *Policy Search* Methode. Hier wird direkt eine parametrisierte Policy π_θ optimiert, mit dem Ziel den erwarteten Reward gegeben der Parameter θ zu maximieren $\mathbb{E}[R|\theta]$.

2.1.4.1 Policy Gradient

In Gradienten-basierten Methoden werden Policies nicht mittels Maximierung, beispielsweise der Q-Funktion (siehe Abschnitt 2.1.3), sondern mithilfe Funktionsapproximation über die Repräsentation einer Policy π und ihrer Parameter θ gelernt.
In einem Gradienten-basierten Ansatz wird eine Aktion, im deterministischen Fall, als

$a = \pi(s|\theta^\pi)$, wobei π eine differenzierbare Funktion mit ihren Parametern θ^π ist. Nach [Sil+14] definiert sich das Lernziel hierbei wie folgt

$$
\begin{aligned}
J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) r(s, \pi_\theta(s)) \mathrm{d}s \\
&= \mathbb{E}_{s\sim\rho^\pi}[r(s, \pi_\theta(s))]
\end{aligned}
\tag{2.13}
$$

Daraus lässt sich nach [Sil+14] der Gradient einer deterministischen Policy wie folgt bilden

$$
\begin{aligned}
\nabla_\theta J(\pi_\theta) &= \int_{\mathcal{S}} \rho^\pi(s) \nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)|_{a=\pi(s|\pi^\theta)} \mathrm{d}s \\
&= \mathbb{E}_{s\ \rho^\pi}[\nabla_\theta \pi_\theta(s) \nabla_a Q^\pi(s,a)|_{a=\pi(s|\pi^\theta)}]
\end{aligned}
\tag{2.14}
$$

Die Anpassung der Parameter θ^π kann beispielsweise durch den Hill-Climbing Ansatz [KBP13] mit einer definierten Schrittgröße α erfolgen.

$$
\theta^\pi_{i+1} = \theta^\pi_i + \alpha \nabla_\theta J \tag{2.15}
$$

2.1.4.2 Actor-Critic

Bei der *Actor-Critic* Architektur [SB98] werden die Vorzüge einer explizit repräsentierten Policy, die mit einem Gradienten trainiert wird, und einem Value-Funktionsansatz, wie in 2.1.3 beschrieben, kombiniert. Dabei ist der *Actor* die explizite Repräsentation der Policy. Dieser agiert über seine Aktionen mit der Umgebung und erhält Feedback in Form des Umgebungszustands. Der *Critic* wiederum bekommt ebenfalls die Zustände der Umgebung sowie zusätzlich den Reward, der sich aus der vorangegangen Aktion ergeben hat. Aus der Kombination dieser beiden Eingaben kann der *Critic* die Q-Funktion lernen und die Aktionen des *Actors* gewissermaßen kritisieren. Dies wird mit dem *TD*-Fehler realisiert. In Abbildung 2.2 ist diese Methode veranschaulicht.

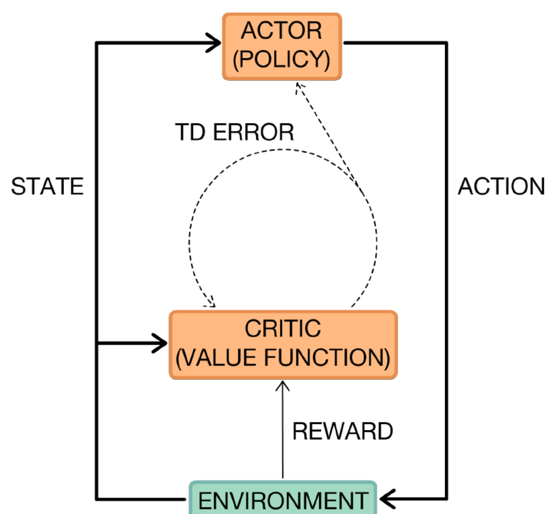


Abbildung 2.2: Die *Actor-Critic* Architektur [Aru+17]

Formal ist ein *Actor-Critic* Algorithmus für eine deterministische Zielpolicy $\mu_\theta(s)$ über eine beliebige stochastische Policy $\pi(s,a)$ wie folgt definiert [Sil+14]

$$\begin{aligned} J_\beta(\mu_\theta) &= \int_{\mathcal{S}} \rho^\mu(s) \nabla_a Q^\pi(s, \mu^\theta(s)) \mathrm{d}s \\ &= \mathbb{E}_{s \ \rho^\mu}[\nabla_a Q^\mu(s, \mu^\theta(s))] \end{aligned} \tag{2.16}$$

Dazu wird die Q-Funktion für den *Critic* als eine differenzierbare Funktion $Q(s,a|\theta^Q)$ mit den Parametern θ^Q definiert. Der Policy Gradient ist dabei

$$\begin{aligned} \nabla_\theta J(\beta_\theta) &= \int_{\mathcal{S}} \rho^\beta(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu(s|\mu^\theta)} \mathrm{d}s \\ &= \mathbb{E}_{s \ \rho^\beta}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu(s|\mu^\theta)}] \end{aligned} \tag{2.17}$$

2.2 Neuronale Netze und Deep Learning

Neuronale Netze bilden die Grundlage des Deep Learnings, mit dem schon in verschiedenen Bereichen in den letzten Jahren sehr gute Ergebnisse erzielt wurden, beispielsweise im Bereich der Bildklassifikation und -segmentierung (vgl. [KSH12] und [BKC17]). Künstliche neuronale Netze werden schon seit recht langer Zeit untersucht. So wurden in [MP43] einfache Netze gebildet, mit denen die Vorgänge von Neuronen im Gehirn modelliert wurden. Diese McCulloch-Pitts-Netze bestehen aus einfachen binären Signalen, die ein Neuron ab einem definierten Schwellwert aktivieren bzw. hemmen. Diese exzitatorischen

(erregenden) und inhibitorischen (hemmenden) Verbindungen sind dabei ungewichtet.
Eine Erweiterung zu den McCulloch-Pitts-Netzen stellt das Perzeptron dar, welches in [Ros58] beschrieben wird. Dieses erweitert den Gedanken der Neuronen um gewichtete Eingangssignale sowie anstelle eines einfachen Schwellwerts eine Aktivierungsfunktion. Damit können den Verbindungen zwischen den Neuronen sozusagen Prioritäten zugewiesen werden. Das Perzeptron bildet die Grundlage für alle heutigen künstlichen neuronalen Netze, die im Grunde nur mehrschichtige untereinander verbundenen Perzeptren sind (vgl. Grafik 2.3). Darüber hinaus gibt es noch diverse neuere Erweiterungen der Netzarchtikturen, wie *Convolutional Neural Networks (CNN)*, die *Long-Short-Term Memory (LSTM) Zelle* [HS97] und *Generative Adversial Networks (GAN)* [Goo+14].

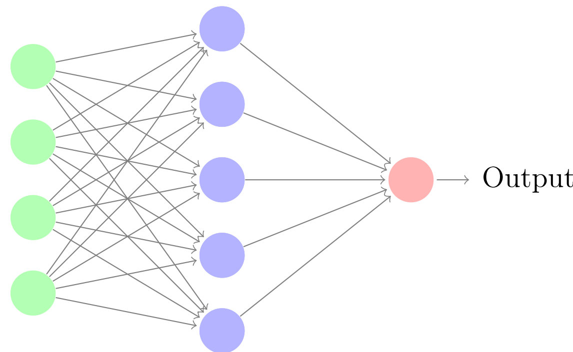


Abbildung 2.3: Ein einfaches neuronales Netz mit einem Hidden-Layer

Einfache vollständig-verbundene neuronale Netze mit N Eingabeeinheiten und M Ausgabeeinheiten können als lineare Funktion beschrieben werden

$$h = Wx + b \tag{2.18}$$

wobei $W \in \mathbb{R}^{M \times N}$ die Gewichtsmatrix, $b \in \mathbb{R}^M$ der Biasvektor, $x \in \mathbb{R}^N$ der Eingabevektor und $h \in \mathbb{R}^M$ ein Vektor von versteckten Einheiten oder der Ausgabevektor y sein kann. Lernbare Parameter sind hierbei die Gewichtsmatirx M und der Biasvektor b.
Um zusätzlich auch nicht-lineare Probleme lösen zu können, werden die Ausgaben eines Layers zusätzlich mit einer Aktivierungsfunktion umgeben.

$$h = f(Wx + b), \tag{2.19}$$

Diese nicht-lineare Funktion $f()$ sollte dabei stetig differenzierbar sein. Einige übliche Aktivierungsfunktionen, abgesehen von der linearen Aktivierung, sind beispielsweise der Tangens Hyperbolicus (tanh) oder die *Rectified Linear Unit (ReLU)*.

$$f(x) = \tanh(x), \ f(x) = \max(0, x) \tag{2.20}$$

Damit ein neuronales Netz trainiert werden kann, muss zunächst eine Fehlerfunktion, auch als *Loss* bezeichnet, definiert werden. Eine hier übliche Metrik ist die Summe der quadratischen Fehler zwischen den Vorhersagen $\hat{y}$ und den wahren Zielwerten y, dies wird auch als *Mean Squared Error (MSE)* bezeichnet.

$$\mathcal{L} = \frac{1}{2} \sum_{n=1}^{N} \parallel \hat{y}_n - y_n \parallel^2 \tag{2.21}$$

Damit das neuronale Netz trainiert werden kann, müssen die Gewichte sukzessive angepasst werden, um den Fehler zwischen Vorhersage und den Zieldaten zu minimieren. Dies wird im Falle von neuronalen Netzen heute üblicherweise durch Backpropagation des Fehlergradienten realisiert, dem so genannten Backpropagation-Algorithmus [RHW86]. Zur Beschleunigung des Lernens sind heute diverse optimierte Verfahren verfügbar wie z.B. Adagrad [DHS11] oder Adam [KB14]. Formal kann die Anpassung der lernbaren Parameter zu einem Schritt t wie folgt definiert werden:

$$\theta_{t+1} = \theta_t - \alpha \nabla_{\theta_k} \mathcal{L} \tag{2.22}$$

Dabei repräsentiert θ alle trainierbaren Parameter des Netzes (W, b) und α eine anpassbare Lernrate, die bestimmt wie sehr sich der Fehler in einem Schritt t auf die Parameter auswirkt.

2.3 Deep Reinforcement Learning

Nachdem in den vorigen Abschnitten die Grundlagen des Reinforcement Learnings und Neuronaler Netze erläutert wurden, werden in diesem Abschnitt Algorithmen aus der Kategorie des *Deep Reinforcement Learnings* vorgestellt, die sich für die Problemstellung eines Agenten mit kontinuierlichem Aktionsraum eignen.

2.3.1 Deep Deterministic Policy Gradient (DDPG)

Einfaches Q-Learning und die darauf basierenden *Deep Reinforcement Learning* Algorithmen wie *Deep Q-Networks (DQN)* [Mni+13; Mni+15] erzielen gute Ergebnisse in einem hochdimensionalen Zustandsraum, lassen sich dabei aber nur in einem in niedrigdimensionalen Aktionsraum verwenden. Dies stellt insbesondere in der Robotik ein Problem dar, da hier ein Agent zumeist über Reglerkommandos in einem kontinuierlichen Aktionsraum $\mathcal{A}$ agiert. Ein Ansatz dieses Problem zu lösen, wäre den Aktionsraum in einem gewissen Rahmen zu diskretisieren. Dies birgt einige Probleme in sich, da ein gut abgestimmter Regler nötig ist, der die fehlenden Zwischenschritte interpolieren kann.

Weiterhin besteht bei einer Diskretisierung die Gefahr, dass diese zu grob gewählt wird, und somit kaum mehr sinnvolle Aktionen vom Agenten ausgeführt werden können.

Um Probleme im kontinuierlichen Aktionsraum zu lösen, schlagen [Lil+15] den *Deep Deterministic Policy Gradient (DDPG)* vor, welcher Q-Learning mit der zuvor in 2.1.4.2 beschriebenen Actor-Critic Architektur ausführt. Anstatt wie bei *DQN* ein Netzwerk für die Q-Funktion zu nutzen, werden hier zwei separate Netzwerke aufgebaut, eines für den Actor, der die Policy π repräsentiert und eines für den Critic, welcher die Q-Funktion repräsentiert. Da es nicht praktikabel ist, die Q-Funktion für alle Aktionen im kontinuierlichen Aktionsraum $\mathcal{A}$ zu maximieren, wird das Actor-Netzwerk μ_θ mit dem Gradienten des Critic-Netzwerks Q_θ als Zielwert trainiert. Konkret speichert der Algorithmus Erfahrungen in jedem Zeitschritt der Form $e = (s_t, a_t, r_{t+1}, s_{t_1})$ in einem Replay-Buffer $\mathcal{R}$. Aus diesem Replay-Buffer werden im Anschluss zufällig Batches der Größe N gezogen. Mit diesen wird auf Basis der aus dem Ziel-Actor-Netz generierten Aktionen, der discounted Reward berechnet. Mit diesen Zielwerten wird das Critic-Netz trainiert und mit dem daraus entstandenen Gradient das Actor-Netz aktualisiert. Die Exploration des Aktionsraums $\mathcal{A}$ wird hier durch Addition eines Wertes aus einem Zufallsprozess auf die aktuelle Aktion realisiert. Als Zufallsprozess wird zumeist der Ornstein-Uhlenbeck-Prozess [UO30] genutzt, der zeitlich korreliertes Rauschen liefert.

Algorithmus 1 : Deep Deterministic Policy Gradient [Lil+15]

Randomly initialise critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights θ^Q and θ^μ
Initialise target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay buffer $\mathcal{R}$
for *episode=1, M* **do**
  Initialise a random process $\mathcal{N}$ for action exploration
  Receive inital observation state s_0
  for *t=1, T* **do**
    Select action $a_t = \mu_\theta(s_t) + \mathcal{N}_t$ according to current policy and exploration noise
    Execute action a_t and observe reward r_t and next state s_{t+1}
    Store tranisition (s_t, a_t, r_t, s_{t+1}) in $\mathcal{R}$
    Sample a random minibatch of N transitions (s_t, a_t, r_t, s_{t+1}) from $\mathcal{R}$
    Set $y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^\mu)|\theta^{Q'})$
    Update critic by minimising 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:

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

    Update the target networks:

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

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

2.3.2 Recurrent Deterministic Policy Gradient (RDPG)

Eine Erweiterung des *DDPG* ist der *Recurrent Deterministic Policy Gradient (RDPG)*. Dieser verwendet im Gegensatz zum *DDPG* keine einfachen Feed-Forward-Netze, sondern rekurrente Netze wie beispielsweise *LSTM*-Zellen. Ein weiterer Unterschied zum nicht-rekurrenten *DDPG* besteht in der Trajektoriengenerierung für den Replay-Buffer und im darauf aufbauenden Training. Da rekurrente Netze mit dem *Backpropagation Through Time (BPTT)*-Verfahren trainiert werden, welches immer abhängige Sequenzen als Eingabe betrachtet, wird der Replay-Buffer mit vollständigen Episoden der Form $e = (o_1, a_1, r_1, o_2, \ldots, o_t, a_t, r_t)$ gefüllt. Dabei bildet die Beobachtung o nicht den zwingend den vollständigen Umgebungszustand ab, da hier auch *Partially Observable Markov Decision Processes (POMDPs)* [Hee+15] gelernt werden können. Das Training verläuft relativ ähnlich zum *DDPG*, wobei hier immer nach Abschluss einer Episode auf Batches von N Episoden trainiert wird.

Algorithmus 2 : Recurrent Deterministic Policy Gradient [Hee+15]

Randomly initialise critic network $Q(a, h|\theta^Q)$ and actor $\mu(h|\theta^\mu)$ with weights θ^Q and θ^μ
Initialise target networks Q' and μ' with weights $\theta^{Q'} \leftarrow \theta^Q,\ \theta^{\mu'} \leftarrow \theta^\mu$
Initialise replay buffer $\mathcal{R}$
for *episode = 1, M* **do**
  Initialise a random process $\mathcal{N}$ for action exploration
  Initialise empty history h_0
  for t *= 1,* T **do**
    Receive observation o_t
    $h_t \leftarrow h_{t+1}, a_{t+1}, o_t$ (append observation and previous action to history)
    Select action $a_t = \mu_\theta(h_t) + \mathcal{N}_t$ according to current policy and exploration noise
  end
  Store the sequence $(o_1, a_1, r_1, \ldots, o_T, a_T, r_T)$ in $\mathcal{R}$
  Sample a random minibatch of N episodes $(o_1^i, a_1^i, r_1^i, \ldots, o_T^i, a_T^i, r_T^i)_{i=1,\ldots,N}$ from $\mathcal{R}$
  Construct histories $h_t^i = (o_1^i, a_1^i, r_1^i, \ldots, o_T^i, a_T^i, r_T^i)$
  Compute target values for each sample episode $(y_t^i, \ldots, y_T^i)$ using the recurrent target networks

$$y_t^i = r_t^i + \gamma Q'(h_{t+1}^i, \mu'(h_{t+1}^i|\theta^{\mu'})|\theta^{Q'})$$

  Compute critic update (using $BPTT$)

$$\Delta\theta^\mu = \frac{1}{\mathrm{NT}} \sum_i \sum_i (y_t^i - Q(h_{t+1}^i, a_t^i)) \frac{\delta Q(h_{t+1}^i, a_t^i|\theta^Q)}{\delta\theta^Q}$$

  Compute actor update (using $BPTT$)

$$\Delta\theta^\mu = \frac{1}{\mathrm{NT}} \sum_i \sum_i \frac{\delta Q(h_{t+1}^i, \mu(h_t^i|\theta^\mu))}{\delta a} \frac{\delta\mu(h_t^i|\theta^\mu)}{\delta\theta^\mu}$$

  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

2.4 Verwendetes System

In dieser Arbeit wurde der Ballspielroboter Doggy als Plattform verwendet. Doggy hat zur Interaktion mit dem Ball drei Freiheitsgrade, wobei der Arbeitsraum eine Halbkugel darstellt. Daraus folgt, dass die Yaw-Achse am Hüftgelenk redundant im Sinne der Abdeckung des Arbeitsraums ist. Der Endeffektor ist eine Styrofoam Kugel mit einem Durchmesser von 40 cm, welche an einer 1 m langen Karbonröhre befestigt ist. Der Roboter ist mit Basis und Antriebseinheit insgesamt 2 m hoch. Im Kopf ist eine *Inertial Measurement Unit (IMU)* verbaut, die zusammen mit der am Mikrocontroller angebrachten *IMU* zur Schätzung der Gelenkpositionen q und Gelenkgeschwindigkeiten $\dot{q}$ genutzt wird [Sch17]. Angetrieben werden die Gelenke von Brushed DC Motoren über ein Zahnriemengetriebe. Die Regelung der Gelenke erfolgt auf einem Mikrocontroller, der gleichzeitig auch die Schätzung des Systemzustands auf Basis der *IMU*-Daten und der Motorencoder vornimmt. Zur Erkennung der Bälle sind zwei monochrome Kameras verbaut, die zueinander stereo-kalibriert sind, um die räumliche Wahrnehmung der Bälle zu ermöglichen.

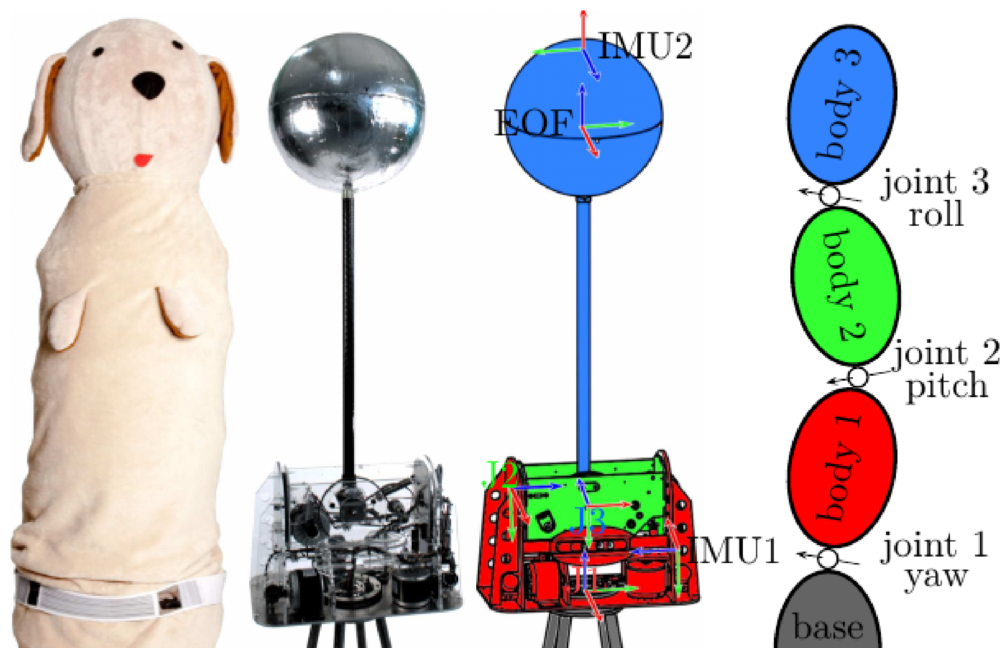


Abbildung 2.4: Darstellung des Ballspielroboters Doggy. Links mit Kostüm, daneben ohne Kostüm, eine schematische Darstellung der Koordinatensysteme und Achsen und rechts eine schematische Darstellung der Achsen. [SWF16]

3 Stand der Technik

In diesem Kapitel werden verschiedene Lösungsansätze vorgestellt, die in einer Simulation gelerntes Verhalten in die Realität übertragen können.

3.1 Domain Randomisation

Eine Simulation kann die reale Welt nur begrenzt gut abbilden. So sind beispielsweise nicht alle Parameter, die ein robotisches System ausmachen klar identifizierbar und können nur näherungsweise in einer Simulation abgebildet werden. Auch limitieren die Simulationen an sich, da sich nicht alle Parameter abbilden lassen, weil sich dies negativ auf die Geschwindigkeit der Simulation (wie die Echtzeitfähigkeit) auswirken würde oder es schlicht nicht möglich ist, da die verwendete Physikengine nur eine bestimmte Menge von Dynamikparametern unterstützt. Des Weiteren können die in einer Physikengine verwendeten Verfahren zur Simulation Limitationen aufweisen, was bestimmte Dynamiken, wie Softbodysimulation oder Kontaktdynamiken zwischen zwei Körpern, angeht.
Auch die simulierten Sensordaten von Kameras etc. und das Verhalten unterscheidet sich meist zwischen Simulation und Realität. So werden in einer Simulation häufig unverrauschte Ground Truth Daten von Sensoren geliefert, die erst nachträglich mit Rauschen versehen werden können. Dies kann aber nur näherungsweise die realen Daten von Sensoren abbilden. Weiterhin unterscheiden sich zum Beispiel Kamerabilder oft stark von ihren realen Gegenstücken, da die verwendeten Modelle nicht genau den realen Robotern und Objekten entsprechen. Dies hat zumeist Performancegründe, da die detaillierte Darstellung eines 3D-Modells mit vielen Polygonen und detaillierten Texturen in Kombination mit einer akkuraten Physikengine viel Leistung in Anspruch nehmen würde.

Nachfolgend werden einige Verfahren vorgestellt, welche die zuvor genannten Probleme mit unterschiedlichen Lösungsansätzen angehen und zu guten Ergebnisse im Transfer von Simulation zu Realität geführt haben.

3.1.1 Randomisierung der Dynamikparameter

Die Dynamikparameter einer Simulation sind die offensichtliche Quelle für eine unscharfe Abbildung des realen Systems. Einerseits können diese teilweise nur näherungsweise bestimmt werden. Andererseits können sich die Parameter zur Laufzeit verändern, z.B. durch

thermische Effekte in Motoren und Gelenken eines Roboters oder sie haben unterschiedliche Eigenschaften auf verschiedenen Oberflächen wie z.B. den Restitutionskoeffizient, welcher bestimmt wie stark ein Objekt bei einer Kollision abprallt.

In [Pen+17] wird ein Ansatz beschrieben, wie ein Agent, in diesem Fall ein Fetch Robotics Arm, lernt in einer randomisierten Simulationsdomäne einen Puck zu einer zufälligen Zielposition zu schieben. Dafür haben Peng et al. die relevanten Parameter der Umgebung sowie des Agenten identifiziert und setzen diese in einem definierten Rahmen in jeder Episode zufällig neu. Der Agent lernt daraus indirekt die Dynamik des Systems.

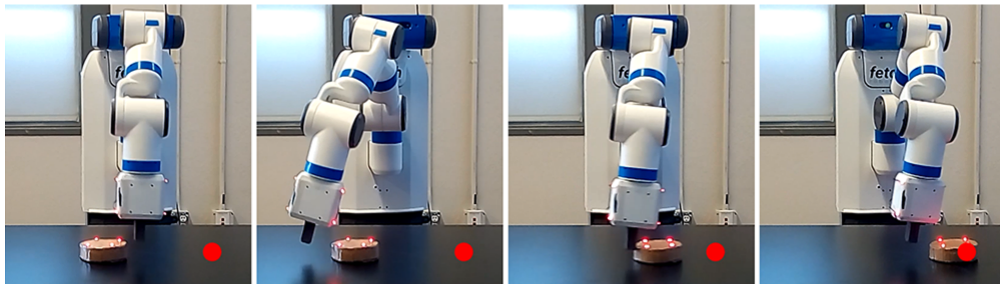


Abbildung 3.1: Eine Policy auf dem Fetch Arm [Pen+17]

Dafür nutzen Peng et al. eine rekurrente Policy auf Basis von *LSTM* Einheiten in Kombination mit einem nicht-rekurrenten Feed-Forward-Zweig. Dabei soll der rekurrente Teil des Netzes die Dynamik des Systems in einer Episode auf Basis des Zustands s_t und der vorangegangen Aktion a_{t-1} erlernen bzw. sich daran anpassen. Im Feed-Forward-Branch erhält der Agent noch Informationen über das Ziel und den Zustand s_t. Der Critic erhält dabei zusätzlich zu den Eingaben, die der Actor bekommt, noch den Satz der aktuellen Dynamikparameter μ und die aktuelle Aktion a_t. Als Algorithmus wird bei Peng et al. *RDPG* verwendet. Die Umgebung liefert nur binäre Sparse-Rewards, also nur wenn der Agent das Ziel erreicht, wird auch ein Reward-Signal gegeben.Um die Schwierigkeiten beim Lernen mit Sparse-Rewards zu umgehen, nutzen Peng et al. ein zusätzliches Replay-Verfahren. Beim *Hindsight Experience Replay (HER)* [And+17] werden, bevor der eigentliche Lernalgorithmus ausgeführt wird, die Rewards einer Episode augmentiert, indem mit einer definierten Wahrscheinlichkeit neue virtuelle Ziele gesetzt werden. Anhand dieser Ziele wird der Reward neu berechnet. Mit diesem Verfahren kann auch aus Fehlschlägen noch gelernt werden, da ein für das ursprüngliche Ziel schlechter Zustand in einem anderen Kontext durchaus einen positiven Reward liefern kann. Dieser neu berechnete Reward wird wieder im Replaypuffer gespeichert.
Mit diesem Verfahren erreichen Peng et al. sehr gute Ergebnisse in Hinblick auf die Übertragung einer rein in Simulation gelernten Policy. Die hier erlernte Policy kann in einem Testszenario in 89 % aller Fälle, bei 28 realen Tests, ein zufällig am Anfang der Episode gesetztes Ziel erreichen. Zusätzlich wurde noch untersucht, ob sich die Policy an eine Änderung der Dynamikparameter anpassen kann. Dies wurde durch die Veränderung der Reibeigenschaften durch Anbringen einer Chipstüte [Pen+17] unter dem Puck erreicht.

Auch in diesem Fall kann die Policy das Ziel in 91 % der Fälle erreichen.

3.1.2 Perzeptive Ansätze

In [Bou+17] wird ein Ansatz beschreiben, in dem ein Roboterarm mittels randomisierter Bilddaten lernt, Objekte verschiedener Formen zu greifen. Hierfür werden mehrere aktuelle Entwicklungen im Bereich der neuronalen Netze miteinander kombiniert. Die Grundidee hier ist ein System auf generierten Simulationsdaten und echten Daten zu trainieren.

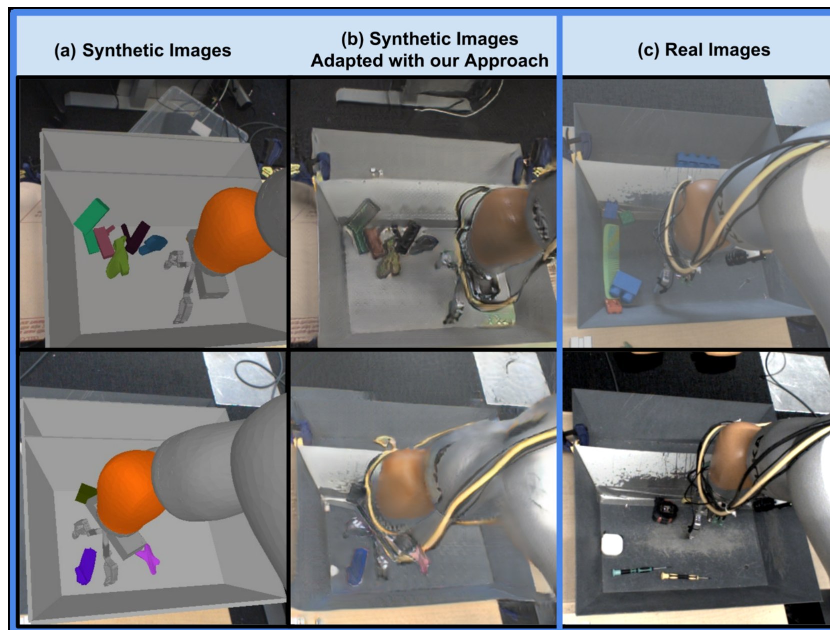


Abbildung 3.2: Simulierte, mit dem Generatornetz adaptierte und reale Bilder [Bou+17]

Der Ansatz besteht dabei aus mehreren Komponenten, einem Vorhersagenetz C, welches die Wahrscheinlichkeit eines erfolgreichen Griffs anhand der Eingabedaten, die aus den Bilddaten und einem Bewegungskommando zu einem Zeitpunkt t bestehen, bewertet. Das Vorhersagenetz ist dabei ein tiefes, mehrschichtiges *CNN*. Die Eingabemenge für das Klassifikatornetz C besteht dabei sowohl aus realen Beispielen als auch simulierten Bildern. Die zu greifenden Objekte wurden für die simulierten Bilder in zwei Ansätzen generiert, zum einen prozedural auf Basis zufälliger geometrischer Formen und zum anderen mit dem ShapeNetv2 [Cha+15], wobei die prozedural generierten Objekte eine insgesamt etwas bessere Performance gezeigt haben. Damit die generierten Simulationsbilder möglichst nah an einem realen Bild sind bzw. schwer davon unterscheidbar sind, haben Bousmalis et al. ein Generatornetz zwischen die Ausgabe der simulierten Bilder und die Eingabe in das Klassifikatornetz gelegt. Bei diesem Netz handelt es sich um ein *GAN*, welches die simulierten Bilder an reale Aufnahmen angleicht (vgl. Abbildung 3.2). Diese Generatornetz

wird mit dem Feedback von C und dem eines Diskriminators D trainiert. Der Diskriminator klassifiziert dabei, ob es sich um ein reales oder ein generiertes Bild handelt.

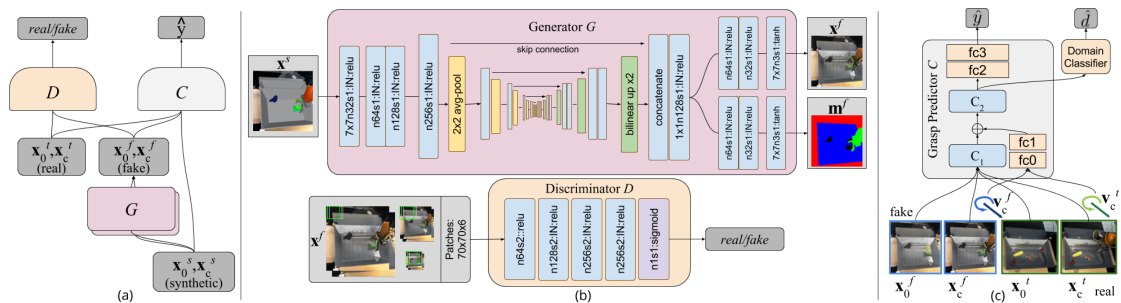


Abbildung 3.3: Das GrapGAN-Netz [Bou+17]

Das gesamte GraspGAN wurde mit einer Kombination aus realen und simulierten Bildern trainiert. Hierbei wurde insbesondere untersucht wie stark sich eine reduzierte Menge realer Beispiele auf die Performance des Netzes auswirkt sowie die Beschränkung auf eine Teilmenge der simulierten Bilder. Das reale Datenset besteht dabei aus Aufnahmen von sechs Kuka-Roboterarmen, die jeweils 102 individuelle Griffe durchgeführt haben, was zu einer Gesamtmenge von 612 Griffen resultiert.
Das mit prozedural generierten Objekten und dem vollen realen Datenset trainierte Netz hat insgesamt eine Erfolgsquote von 76.67%, was eine gute Performance in Anbetracht der Komplexität der gestellten Aufgabe ist. Wird die Menge der realen Beispiele auf 20 % des Datensatzes reduziert, liegt die Performance immer noch bei 74.04%, was immer noch ein sehr gutes Ergebnis ist. Selbst bei nur 1 % des realen Datensatzes erreicht das Netz noch eine Performance von 59.95% erfolgreicher Griffe.

In [Tob+17] wird ein Ansatz beschrieben, um die Unzulänglichkeiten von simulierten Kameras auszugleichen. Simulierte Kamerabilder haben wie zuvor schon genannt das Problem, dass sie aufgrund der Limitation einer Simulation meist keine realitätsgetreuen Bilder liefern können. Dies liegt meist daran, dass keine hochqualitativen Modelle der Systeme vorhanden sind sowie an den Limitierungen in der grafischen Darstellung, da ein fotorealistisches Rendering auch heute noch zu viel Rechenleistung in Anspruch nehmen würde. Daher schlagen Tobin et al. vor einen Objektklassifkator auf Basis von stark randomisierten Simulationsbildern zu trainieren. Dabei werden unter anderem die Anzahl und Art der Objekte variiert, die Beleuchtung sowie die Parameter der Kamera, wie zum Beispiel die Position im Raum (siehe Abbildung 3.4). Der Klassifikator ist eine abgewandelte VGG-16 Architektur, welche aus 13 Convolutional Layern und zwei Fully-Connected Layern besteht. Ausgabe ist dabei eine Position (x, y, z) der zu klassifizierenden Objekte. Das Verfahren hat bei Tests gute Ergebnisse erzielt. So konnte ein Fetch-Robotics Arm 38 von 40 Objekte in einer Szene mit vielen Objekten erfolgreich greifen.

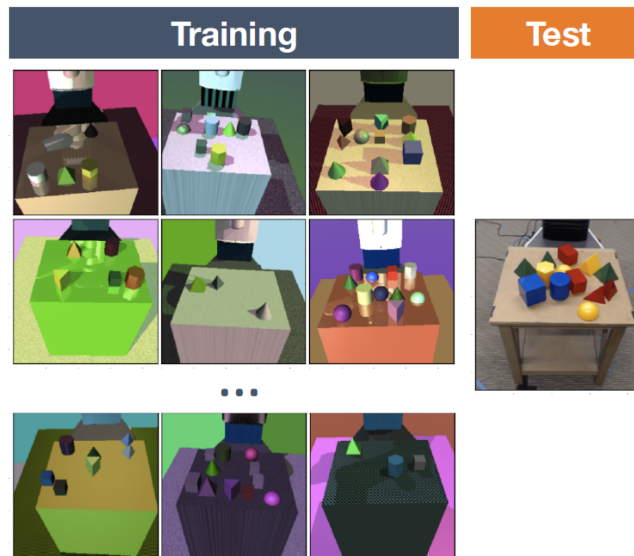


Abbildung 3.4: Generierte, randomisierte Bilder in der Simulation und das reale Szenario [Tob+17]

Auf Basis dieses Domain-Randomisierungsverfahrens haben [Pin+17] einen Multi-Ziel Agenten trainiert. Dafür haben Pinto et al. eine asymmetrische Actor-Critic-Architektur entwickelt. In diesem Fall bekommt der Actor eine vom Umgebungszustand abgeleitete Beobachtung o_t in Form eines RGB-Bildes. Der Critic hingegen hat als Eingabe den Umgebungszustand s_t. Zusätzlich bekommen beide Netzwerke noch das zu erreichende Ziel als Eingabe, da hier eine universelle Policy gelernt werden soll, die auch mit wechselnden Zielen umgehen können soll. Trainiert wurden hier sowohl reine *DDPG*-Agenten als auch mit *HER* augmentierte Agenten.
Bei Tests auf dem realen System hat sich gezeigt, dass die hier trainierten Policies sehr gut übertragbar auf die Realität sind. In den drei Szenarien (Pick, Push, Block Move) konnte der *HER*-Agent in jeweils fünf Testläufen jeden erfolgreich absolvieren. Dabei haben Vergleichstests auch gezeigt, dass die Domain-Randomisierung hier eine große Rolle gespielt hat. Ein Agent, der ohne Randomisierung gelernt hat, konnte in keinem Szenario das Ziel erreichen. Wenn nur der Blickpunkt der Kamera nicht variiert wurde, konnte nur im *Block Move* Szenario in vier von fünf Fällen das Ziel erreicht werden.

4 Ansatz

Im diesem Kapitel wird das verwendete Verfahren sowie dessen Integration in das System beschrieben.

4.1 Verwendetes Verfahren

Da das Ziel einer möglichst systemnahe Modellierung angestrebt wird, die nach Möglichkeit viele der vorhandenen Komponenten auf dem realen System nutzen kann, soll der in [Pen+17] beschriebene Ansatz der Dynamikrandomisierung der System- und Umgebungsparameter zum Einsatz kommen. Ein Ansatz, der auf externen Kamerabildern lernt, wie er in [Pin+17] oder [Bou+17] beschrieben wird, liefert zwar gute Ergebnisse, ist aber im Kontext eines Ballspielroboters, der auch in einem Eventkontext eingesetzt werden soll, nicht realisierbar, da dies den Aufbau des Systems deutlich verkomplizieren würde. Weiterhin ist dabei nicht unbedingt sichergestellt, dass ein solcher Ansatz auch mit großen Störungen durch unvorhergesehene Ereignisse umgehen kann. Insbesondere ist auch in Betracht zu ziehen, dass der vorhandene im System *Multiple Hypothesis Tracker (MHT)* bereits sehr gute Vorhersagen ermöglicht, auch von mehreren Bällen gleichzeitig, und effizient implementiert ist. Zudem würde es das Lernproblem im gegebenen Kontext deutlich verkomplizieren, wenn das System zusätzlich zum Schlagen der Bälle auch noch die Vorhersage und den Schnittpunkt mit dem Arbeitsraum selbst lernen sollte. Daher wird im Folgenden angenommen, dass ein Trackingsystem für Bälle existiert, welches gute Vorhersagen liefert.

4.1.1 Identifikation der relevanten Dynamikparameter

Zunächst müssen die für das Ballspielproblem relevanten Dynamikparameter festgestellt werden, da diese in Experimenten randomisiert werden sollen. Betrachten wir als Erstes das System *Doggy*. Der Roboter kann über drei Achsen seinen Endeffektor bewegen. Demnach sind drei Gelenke vorhanden, deren Parameter betrachtet werden müssen. Dabei sind von besonderem Interesse der Reibbeiwert μ und die Dämpfung der Gelenke d_s. Diese sind in Bezug auf die Beweglichkeit der Achsen wichtig, da besonders die Roll-Achse eine hohe Reibung und Dämpfung durch ihre Konstruktion aufweist. Da für die Simulation ein vereinfachtes Modell des Roboters genutzt wird, sind die auf dem realen System vorhanden Zahnriemengetriebe nicht modelliert. Stattdessen werden zwischen den Links einfache Revolute-Joints verwendet, die sich in der entsprechenden Achse bewegen können. Also müssen die zuvor genannten Parameter die gesamte Dynamik des Antriebsstrangs

modellieren, besonders auch Effekte die in der Simulation nicht modelliert werden können, wie der Abrieb in den Zahnriemen. Zudem lassen sich im verwendeten Simulator *Gazebo* keine Motoren abbilden. Daher müssen diese Parameter außer Acht gelassen werden und können nur indirekt durch die Limitierung des möglichen Gelenkmoments Anwendung finden. Im Antriebsstrang sind zusätzlich auf Softwareseite noch die Gains des auf dem Mikrocontroller laufendes Reglers relevant, da diese Werte großen Einfluss auf die Bewegung der Gelenke haben.
Weitere für die Bewegung des Roboters interessante Parameter sind die Massen der Links und damit einhergehend der Trägheitsmomente. Aufgrund des vereinfachten Simulationsmodells wird das Kostüm nicht mitmodelliert. Zudem ist die verwendete Physik-Engine *Open Dynamics Engine (ODE)* in dieser Hinsicht limitiert, da hier keine Softbody-Simulation des Fells möglich ist. Daher werden die Massen der Links in einem gewissen Rahmen randomisiert, um zum einen eventuelle Ungenauigkeiten im CAD-Modell auszugleichen, zum anderen den Verzicht auf das Kostüm in der Simulation zu kompensieren.

Abgesehen vom Roboter selbst müssen auch die Dynamikeigenschaften des Balls betrachtet werden. Hierbei sind von besonderem Augenmerk die Kontaktdynamik des Balls im Moment des Schlages sowie eventuelles Abprallen auf dem Boden nach dem Schlag. Der hier relevante Parameter ist der Restitutionskoeffizient k, welcher bestimmt, wie sehr ein Objekt nach einer Kollision wieder zurückspringt. Dieser variiert je nach Materialbeschaffenheit des Objekts sowie des Balldrucks. So verhält sich ein Ball mit einem hohen Druck im Moment des Schlages deutlich anders als ein Ball mit einem niedrigeren Druck. Weiterhin lässt sich hiermit die Unschärfe durch den Verzicht auf das Kostüm miteinbeziehen. Darüber hinaus ist der Reibungskoeffizient μ des Balls für die Kontaktdynamik relevant. Zusätzlich soll auch die Masse des Balls in die Menge der relevanten Parameter aufgenommen werden.

Aus der vorangegangenen Analyse ergibt sich die folgende Menge von relevanten Dynamikparametern.

Objekt	Relevante Dynamikparameter
Doggy Links	m
Doggy Gelenke	μ, d_s, PID-Gains
Ball	m, μ, k

Tabelle 4.1: Identifizierte relevante Dynamikparameter

Die genauen für die Simulation genutzten Parameter sind im folgenden Kapitel in Tabelle 5.2 aufgeführt.

Diese Parameter werden dann im Prozess des Lernverfahrens zu Beginn jeder Episode in einem definierten Rahmen zufällig neu gesetzt.

4.1.2 Verwendeter Algorithmus

Als Lernalgorithmus werden *DDPG* und *RDPG* genutzt werden. Aus der Literaturarbeit zu [Pen+17] hat sich ergeben, dass diese beiden Algorithmen gute Ergebnisse in Bezug auf die Übertragbarkeit von in Simulation gelernten Policys in der Realität geliefert haben. Beim Aufbau der Layer der neuronalen Netze wird sich an der Literatur orientiert.

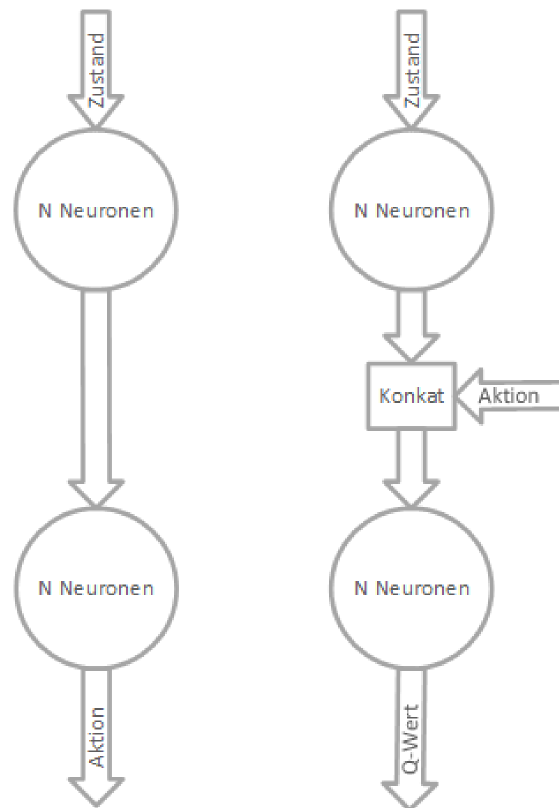


Abbildung 4.1: Layer für den *DDPG*- und *RDPG*-Agenten, links der Actor, rechts der Critic

Der Aktionsraum $\mathcal{A} \in \mathbb{R}^3$ umfasst dabei die angetriebenen Achsen Yaw, Pitch und Roll (vgl. Abbildung 2.4). Zusätzlich soll noch der Aktionsraum $\mathcal{A} \in \mathbb{R}^2$ mit den Achsen Pitch und Roll betrachtet werden, um festzustellen, ob der Agent im Stande ist, die erweiterten Bewegungsmöglichkeiten der Hüftachse (Yaw) zu nutzen. Die beobachtbaren Zustände umfassen dabei

- die Position des Schlägers $x_{ee} \in \mathbb{R}^3$ im kartesischen Raum
- die Gelenkpositionen $q \in \mathbb{R}^3$
- die Gelenkgeschwindigkeiten $\dot{q} \in \mathbb{R}^3$
- die Ballposition $x_{ball} \in \mathbb{R}^3$

- die Ballgeschwindigkeit $\dot{x}_{ball} \in \mathbb{R}^3$
- der vorhergesagte Schnittpunkt des Balls mit dem Arbeitsraum mit relativer Zeit zum Zeitpunkt des Schnitts mit dem Arbeitsraum $\hat{x}_{hit} \in \mathbb{R}^4$
- die relative Position des Schlägers $x_{rel} = x_{hit} - x_{ee}, \in \mathbb{R}^3$ zum vorhergesagten Schnittpunkt x_{hit}
- die Position des Zielobjekts $x_{goal} \in \mathbb{R}^3$

Aus diesen Zuständen ergibt sich der Zustandsraum $\mathcal{S} \in \mathbb{R}^{25}$. Diese Zustände wurden so gewählt, dass sie alle auch auf dem realen System ohne größeren Aufwand zu akquirieren sind. Die Zustände der Gelenke lassen sich aus dem Schätzer auf dem Mikrocontroller abfragen und die Informationen über den Ball liefert der Tracker. Die Position des Schlägers lässt sich einfach aus der Vorwärtskinematik mit den Gelenkwinkel q berechnen.

4.1.3 Rewardfunktion

Große Relevanz kommt dem Entwurf der Rewardfunktion zu. Probleme, die aus einer nicht gut entworfenen Rewardfunktion entspringen können sind unter anderem: Lernen von potenziell unerwünschtem oder schädlichem Verhalten [KBP13]. Weiterhin muss die Art der Rewardfunktion mit großer Sorgfalt gewählt werden. In unserem Ballspielszenario erscheint es offensichtlich nur dann einen Reward zu geben, wenn der Ball erfolgreich zum Ziel gespielt wurde. Diese Art der Rewardfunktion von sparse, binary Rewards hat den Vorteil, dass sie leicht umzusetzen ist. Dennoch hat diese Methode einige Nachteile in Hinblick auf die Maximierung des Rewards in einem Lernverfahren. So hat der Agent bis zum Treffen des Ziels kein Feedback bekommen, welches ihm ermöglicht einen Rückschluss auf den aktuellen Spielzustand zu ziehen. Eine weitere Möglichkeit wäre ein weiteres Rewardsignal, wenn der Ball getroffen wurde. Dies ist eine übliche Vorgehensweise zum Beispiel in mehrstufigen Manipulationsszenarien, in denen ein Agent erst ein Objekt greifen soll und im Anschluss auf einem anderen Objekt platzieren soll. Hierbei besteht die Gefahr, dass der Agent nur lernt den Ball zu schlagen bzw. den Ball abprallen zu lassen, da er in diesem Szenario quasi nur eine Möglichkeit hat, den Ball aktiv in Richtung des Ziels zu schlagen.
Daher wird eine graduelle Funktion gewählt, die sich einem Maximum annähert, je besser das Ziel erreicht wurde. Eine geeignete Wahl erscheint dabei der negative Exponent der e-Funktion zu sein. Insbesondere ist es von Vorteil, wenn sich die Funktion im Falle eines positiven Rewards 1 annähert.

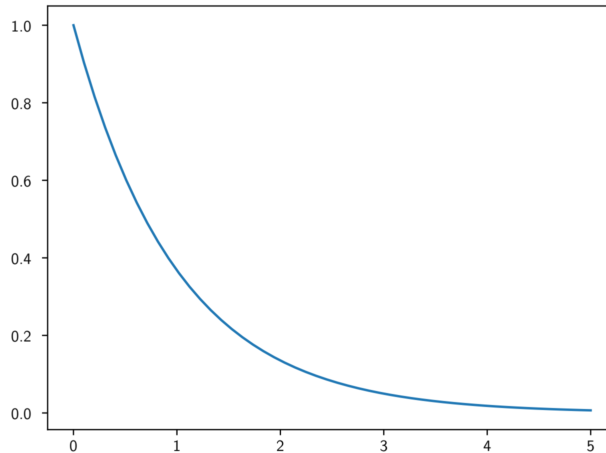


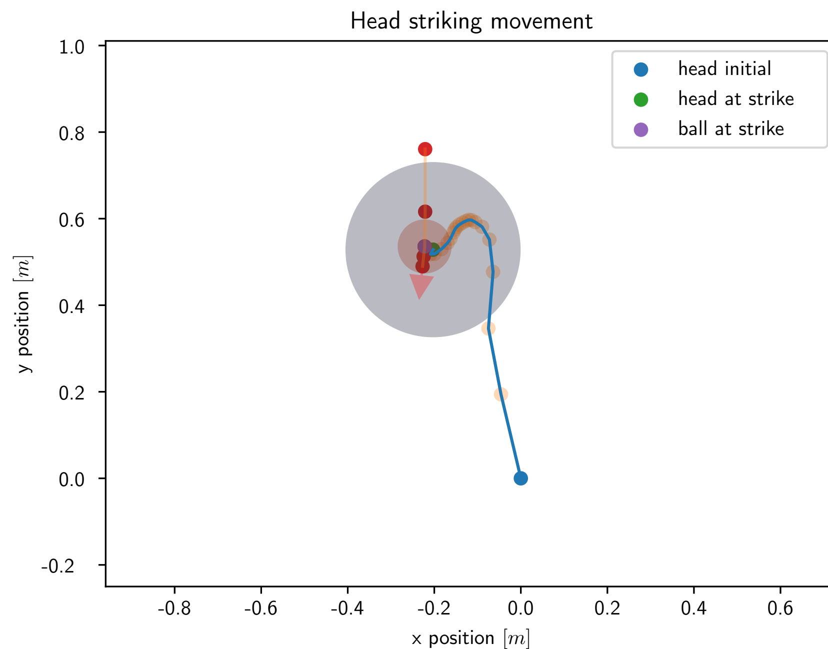
Abbildung 4.2: Negativer Exponent

Zusätzlich soll jedoch die Idee der mehrteiligen Rewards miteingflochten werden. So kann es dem Agenten durchaus helfen, wenn er vor dem Schlagen des Balls eine Vorstellung über das Rewardsignal bekommt, wie weit er von der berechneten Schlagposition entfernt ist. Dazu wird, sobald der Ball geschlagen wurde, die Entfernung zum Ziel als zusätzliches Rewardsignal genutzt. Wenn der Ball das Zielobjekt trifft, wird ein großes Rewardsignal gegeben und in diesem Fall ist auch ein Terminalzustand erreicht. Der Reward für diesen Fall ist so angelegt, dass der Agent diesen nicht durch zufälliges Abschließen einer Episode erreichen kann. Das ist insofern notwendig, als der Agent sonst nicht zwischen einem positiv abgeschlossenen Terminalzustand und einem beliebigen Zustand in der Interaktion mit der Umgebung unterscheiden kann. Aus diesen Vorüberlegungen ergibt sich die nachfolgende Rewardfunktion $r(s_t)$

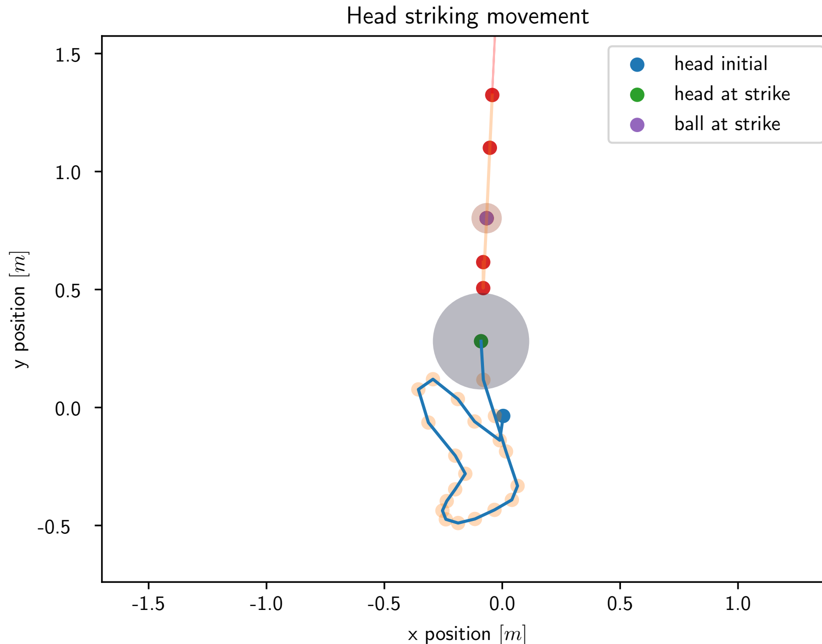
$$r(s_t) = \begin{cases} 50 + \frac{1}{2}\mathrm{e}^{-\|x_{ee} - x_{hit}\|^2} & \text{if target hit} \\ \frac{1}{2}\mathrm{e}^{-\|x_{ball} - x_{goal}\|^2} + \frac{1}{2}\mathrm{e}^{-\|x_{ee} - x_{hit}\|^2} & \text{if ball hit} \\ \frac{1}{2}\mathrm{e}^{-\|\hat{x}_{ee} - \hat{x}_{hit}\|^2} & \text{else} \end{cases} \tag{4.1}$$

Dabei soll für den Zeitraum vor dem Schlag die Distanz zwischen Endeffektorpose x_{ee} und erwarteter Schlagposition x_{hit} als Maß genutzt werden. In einer Voruntersuchung hat sich gezeigt, dass nur die Distanz zwischen Schläger und erwarteter Schlagposition kein ausreichendes Maß für eine sinnvolle Schlagbewegung ist. Wird zusätzlich die zeitliche Distanz zum erwarteten Schlagzeitpunkt t_{hit} in die Rewardberechnung mit aufgenommen, wird der Agent ermutigt erst zum vorhergesagten Zeitpunkt an der Schlagposition zu sein. Hieraus ergeben sich echte Schlagbewegungen, wohingegen ein Reward nur mit der Distanz zur vorhergesagten Schlagposition vornehmlich Abpraller produziert.

Das beschriebene Verhalten der Abpraller ist insbesondere in Abbildung 4.3a gut zu erkennen. Dargestellt ist hier der Pfad des Schlägers von der Startposition zur Zielposition sowie die Bewegung des Balls drei Zeitschritte vor und nach dem Schlag mit der Richtung, in die der Ball sich nach dem Schlag bewegt. Die graue Sphäre stellt den Schläger, die rote Sphäre den Ball jeweils zum Zeitpunkt des Schlages in der Draufsicht dar.
Hier ist der Schläger klar zu früh an der vorhergesagten Ballposition und verweilt an selbiger, was zur Folge hat, dass der Ball in diesem Fall nach links abprallt. In Abbildung 4.3b ist nach Änderung der Rewardfunktion eine echte Schlagbewegung zu erkennen, die den Ball mit großer Geschwindigkeit in Richtung des Ziels schlägt. Beide Simulationen wurden mit dem gleichen Randomseed zur Wahrung einer möglichst guten Vergleichbarkeit durchgeführt. Da es sich hier um eine Voruntersuchung handelt, sind die abgebildeten Bewegungen nicht repräsentativ für die in den finalen Experimenten gelernten Bewegungen.



(a) Bewegung des Schlägers bei der Rewardfunktion ohne Zeitkomponente



(b) Bewegung des Schlägers bei der Rewardfunktion mit Zeitkomponente

Abbildung 4.3: Vergleich der Bewegungen des Schlägers mit Rewardfunktion ohne und mit Zeitkomponente

Bei einem erfolgreichen Schlag wird als zusätzliches Singal noch die Distanz zwischen Ballposition x_{ball} und Zielobjekt x_{goal} genutzt. Im Falle eines erfolgreichen Treffers wird ein großer Reward anstelle des Distanzmaßes gegeben, was einen positiven Terminalzustand bezeichnet und demnach das erfolgreiche Ende einer Episode markiert.

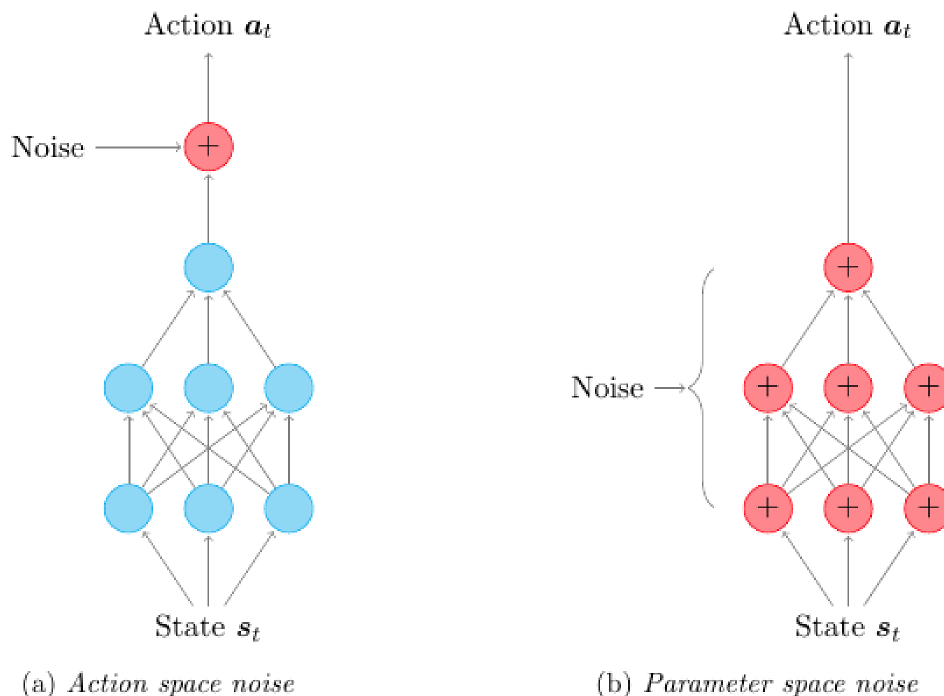
4.1.4 Exploration des Aktionsraums

Ein weiterer wichtiger Aspekt beim Lernen eines Verhaltens ist eine ausreichende Exploration des zur Verfügung stehenden Aktionsraums $\mathcal{A}$. Dies wird bei *DDPG* und verwandten Algorithmen üblicherweise über das Verrauschen des Aktionsoutputs des Actor-Netzes erreicht. Dafür wird der Ornstein-Uhlenbeck-Prozess [UO30] genutzt. Hierbei handelt es sich um zeitlich korreliertes Rauschen, welches im Falle vom *DDPG* die Aktion additiv mit Rauschen versieht.

$$\hat{\pi}(a|s) = \pi_\theta(a|s) + \mathcal{N}(0, \sigma^2 I) \tag{4.2}$$

Diese Art der Exploration hat den Nachteil, dass hier nur die Ausgabe des Actor-Netzes

betrachtet wird.



(a) *Action space noise*

(b) *Parameter space noise*

Abbildung 4.4: Vergleich der Wirkungsbereiche von reinem Aktionsrauschen und Parameterraumrauschen [Pla17]

In [Pla17] wird perturbiertes Parameterrauschen beschrieben. Dieser Ansatz verrauscht den gesamten Parameterraum θ des Actor-Netzes (vgl. Abbildung 4.4). Im Falle von neuronalen Netzen im Kontext des *Deep Reinforcement Learning* sind dies die Gewichtsvektoren der Neuronen.

$$\tilde{\theta} = \theta + \mathcal{N}(0, \sigma^2 I) \tag{4.3}$$

Bei diesem Verfahren sind zwei Dinge zu beachten, einerseits ist das Verhalten von verschiedenen Layern unterschiedlich sensitiv auf das Rauschen, was es nötig macht eine Möglichkeit der Vereinheitlichung hierfür zu finden. Andererseits ist das Maß der Verrauschung der Parameter nicht intuitiv, da die Layer eines Netzes zumeist als Black-Box angesehen werden. Daher kann die Bestimmung einer passenden Standardabweichung σ nicht offensichtlich sein. Plappert schlägt zur Lösung des ersten Problems die Anwendung von *Layer Normalization* [BKH16] nach jedem Fully-Connected-Layer vor. Dieses Verfahren arbeitet unabhängig von Batches, da hierfür die Normalisierung die Layer und nicht die Eingabe betrachtet werden. Für das zweite Problem schlägt Plappert eine adaptive Skalierung des Rauschens anhand der Ausgabe des Actor-Netzes vor, weil eine

Betrachtung der Parameter als Maß für σ ungeeignet erscheint, aufgrund des eingangs erwähnten Black-Box-Charakters eines neuronalen Netzes. Formal beschrieben wird die adaptive Skalierung von σ wie folgt

$$\sigma_{k+1} = \begin{cases} \alpha\sigma_k & \text{if } d(\pi(a|s), \tilde{\pi}(a|s)) \leq \delta, \\ \frac{1}{\alpha}\sigma_k & \text{otherwise} \end{cases} \tag{4.4}$$

Dabei ist $d(.,.)$ ein Distanzmaß zwischen der perturbierten Policy $\tilde{\pi}$ und der nicht perturbierten Policy π, was im Falle von *DDPG* dem Ziel-Actor-Netz entspricht, da dieses immer unverrauschte Ausgaben liefert, $\delta \in \mathbb{R}_{>0}$ ist ein Schwellwert und $\alpha \in \mathbb{R}_{>0}$ der Wert mit dem der aktuelle Wert von σ_k angepasst wird.

Dieses Verfahren hat zu sehr guten Ergebnissen geführt und war im Mittel dem reinen Aktionsrauschen überlegen. Insbesondere hat sich gezeigt, dass Parameterrauschen zu deutlich anderen und oftmals auch bessere Policys im Kontext eines sinnvollen Verhaltens geführt hat. [1] Da dies für das Finden von geeigneten Schlagbewegungen insbesondere bei einem Achsen-redundanten System geeignet erscheint, soll in unserem Fall Parameterrauschen anstatt reinen Aktionsrauschen zum Einsatz kommen.

[1] Hier ist insbesondere das Verhalten in der Umgebung *HalfCheetah-v1* exemplarisch für die besseren Policies `https://blog.openai.com/better-exploration-with-parameter-noise/`, (*abgerufen am 10.06.2016*)

5 Umsetzung

In diesem Kapitel wird die Erstellung der Simulationsumgebung und der Lernumgebung beschrieben, wobei zunächst festgestellt wird, wie der Stand der aktuellen Software ist.

5.1 Stand der Software

Zu Beginn dieser Arbeit wird auf dem Roboter eine einfache Routine zur Bewegung zu einer vorhergesagten Schnittposition des Balls mit dem Arbeitsraum genutzt. Die Software ist in einem *Robot Operating System (ROS)*-Stack integriert. Dafür werden mit einem Kreiserkenner und einem *Multiple Hypothesis Tracker (MHT)* [BF09] Balltrajektorien generiert. Der Tracker erstellt auf Basis der zwei Kamerabilder eine 3D-Repräsentation der erkannten Kreise, wobei diese auf Basis des im Tracker vorhandenen Ballmodells nach Wahrscheinlichkeit eines fliegenden Balls vorgefiltert werden.

Abbildung 5.1: Beispiel eines erkannten in kräftigem rot Balls mit Vorhersage der Ballflugbahn in grün

Auf Basis der Differenzbilder zweier erkannter Bälle und des Ballmodells, kann der *MHT* Balltracks generieren, die von der High-Level-Steuerung verarbeitet werden. Sobald

genügend Tracks vorhanden sind, testet die Steuerungskomponente, ob die Vorhersagen erreichbar sind und plant daraufhin mittels inverser Kinematik eine Zielpose für den Endeffektor. Diese wird im aktuellen Zustand so schnell wie möglich angefahren, ohne den vorhergesagten Zeitpunkt, in dem der Ball den Arbeitsraum schneidet zu beachten. Das hat zur Folge, dass der Roboter häufig Bälle verfehlt, weil er durch die mechanischen Limitierungen diese nicht rechtzeitig erreichen kann oder Bälle in eine beliebige Richtung schlägt, da es sich hier nicht um wirkliche Schlagbewegungen, sondern nur um eine möglichst schnelle Bewegung zur Zielposition handelt. Das hat auch zur Folge, dass der Endeffektor unter Umständen zu früh an der Zielposition sein kann und der Ball in diesem Fall nur abprallt.

Auf Hardwareebene wird zur Regelung der Gelenkwinkel ein Mikrocontroller eingesetzt. Die Regelung selbst ist ein einfacher P-Regler für alle Gelenke. Zusätzlich ist auf dem Mikrocontroller der Zustandsschätzer aus [Sch17] implementiert. Dieser liefert eine Schätzung unter anderem der Gelenkpositionen $q \in \mathbb{R}^3$ und -geschwindigkeiten $\dot{q} \in \mathbb{R}^3$, welche von besonderem Interesse für das Ziel des Ballspielens ist. Ferner liefert der Mikrocontroller die Werte der *IMU* für den Tracker sowie die Werte der Motorencoder $\theta, \dot{\theta} \in \mathbb{R}^3$.

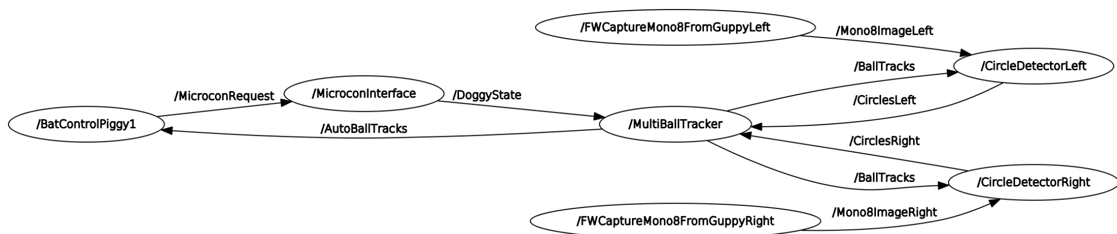


Abbildung 5.2: Zustand der Softwarekomponenten auf dem PC im Demobetrieb

Eine Simulationsumgebung, die in den *ROS*-Stack integriert ist, war zu Beginn der Arbeit noch nicht vorhanden. Bisher ist lediglich ein CAD-Modell, das zur Konstruktion genutzt wurde, vorhanden.

5.2 Erstellen einer Simulationsumgebung

Ein erstes Ziel ist die Erstellung einer Simulationsumgebung, die leicht in den vorhandenen Softwarestack integriert werden kann. Dafür müssen sowohl ein Modell des Roboters für den Simulator als auch ein Ballmodell erstellt werden.
Die Wahl des Simulators fällt dabei auf *Gazebo* [KH04]. Dieser ist mit entsprechenden Bibliotheken schon in *ROS* integriert und nutzt das gleiche Protokoll zum Nachrichtenaustausch. Da zum Beginn der Arbeit wie eingangs erwähnt noch kein Modell des Roboters *Doggy* für eine Simulation verfügbar war, wurde dieses auf Basis der Parameter (Masse, Abmessungen, etc.) aus dem CAD-Modell sowie der in [SWF16] identifizierten Dynamikparameter für die Gelenke erstellt. Visuell wurde auf eine vereinfachte Darstellung des

Roboters zurückgegriffen (siehe Abbildung 5.3). Die relevanten Daten des Modells werden hier kurz tabellarisch aufgeführt.

Link	Masse [kg]	Maße [m]
base	11,043	0,52 × 0,754 × 0,52
electronic_box	11,328	0,547 × 0,371 × 0,307
pitch_box	5,929	0,3 × 0,3 × 0,3
stick	0,179	$h = 1{,}163$, $r = 0{,}05$
head	0,377	$r = 0{,}2$

Tabelle 5.1: Parameter der Links

Gelenk	μ [N m]	d_s [N m s rad/s]	Limits [°]
Yaw	1,6	5,335	$[-80, 80]$
Pitch	2,56	4,143	$[-55, 90]$
Roll	4,86	8,043	$[-42, 45]$

Tabelle 5.2: Parameter der Gelenke

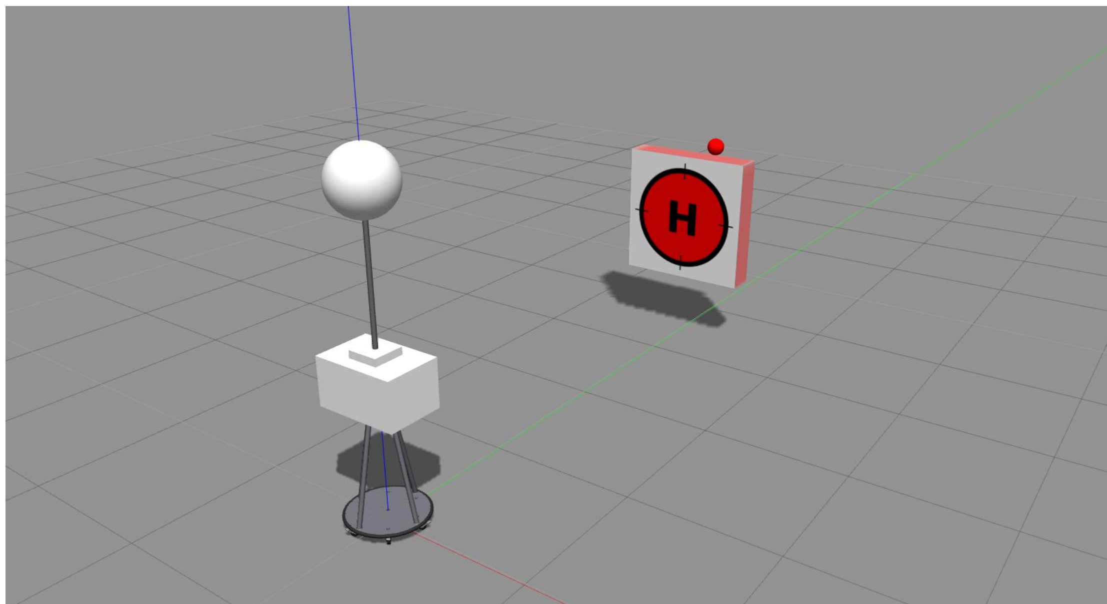


Abbildung 5.3: Doggy in der Simulationsumgebung mit Ball in einer Startpose und dem Zielobjekt.

Weiterhin müssen einige Komponenten exklusiv für die Simulation erstellt werden, da diese nicht ohne großen Aufwand integrierbar sind bzw. direkt auf dem Mikrocontroller implementiert sind. Für die Ballvorhersagen wird hierbei auf eine einfache Vorhersage

für Projektiltrajektorien auf Basis der Ground Truth Daten aus der Simulation gesetzt. Diese wird wie folgt realisiert

$$x_{t+1} = x_t + (\dot{x}_t \Delta t + g)\alpha \tag{5.1}$$

wobei $\alpha = 1 - \Delta t c_w$ der Luftwiderstand ist und g die Gravitationsbeschleunigung. Darauf aufbauend können aus generierten Ballvorhersagen Schnittposition mit dem Arbeitsraum von Doggy erstellt werden. Diese Generierung folgt einer ähnlichen Logik wie die bereits bestehende Komponente auf dem Realsystem. Die Funktionalität wird über einen Service in der Komponente *doggy_simulation* angeboten.

Weiterhin muss für eine Lernumgebung ein Feedback vorhanden sein, mit dem bestimmt werden kann, ob der Ball und das Ziel getroffen wurden. Dies wird mittels simulierter Kontaktsensoren realisiert, von denen je einer im Endeffektor des Roboters und einer im Zielobjekt platziert sind. Diese Kontaktsensoren geben bei Kollision mit einem Objekt an, um welches es sich handelt. Diese Daten werden mit einem *ROS*-Service in der Komponente *doggy_simulation* bereitgestellt. Zusätzlich werden auch die Ballposition und -geschwindigkeit sowie die Gelenkposition und -geschwindigkeit zum Schlagzeitpunkt ausgegeben, damit untersucht werden kann, wie der Roboter den Ball schlägt.
Außerdem müssen Beobachtungen über den Systemzustand und den Zustand des Balls, die auf dem realen System aus dem Tracker und dem Mikrocontroller kommen, vorhanden sein. Diese Informationen sind in der Simulation relativ einfach zu beschaffen, da diese entweder aus schon vorhandenen Topics der Modellzustände im Simulator abgegriffen werden können oder als Modellplugin realisiert werden können.

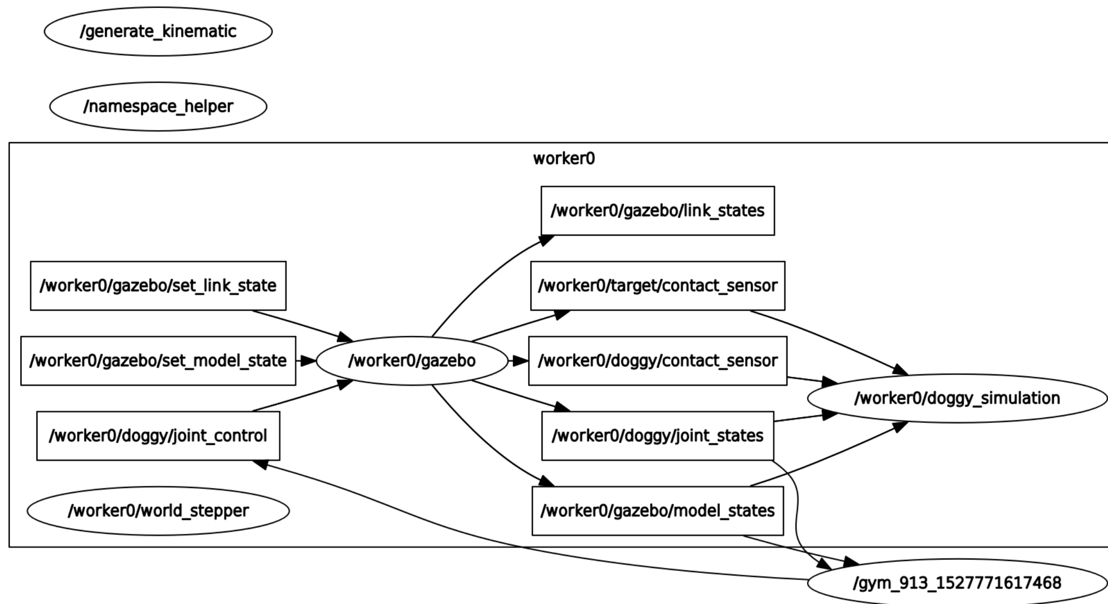


Abbildung 5.4: Übersicht über die Nodes und Topics in einer Simulationsumgebung mit einer Instanz und einer Lernumgebung

Darüber hinaus werden noch eine Reihe von Plugins für *Gazebo* verwendet und erstellt, welche entweder Daten über den Zustand des Modells oder für die Domain Randomisierung genutzt werden. Vorhandene Modellplugins sind z.B. das Controller-Plugin für die Gelenke und der Publisher der Gelenkzustände. Da eine möglichst systemnahe Modellierung angestrebt wird, ist der für die Simulation verwendete Regler wie auf dem echten System ein einfacher P-Regler.

```
<world>
    <model name="doggy">
        <plugin name="doggy_joint_states" filename="libdoggy_joint_states_plugin.so">
            <jointName> Doggy::yaw, Doggy::pitch, Doggy::roll</jointName>
        </plugin>
        <plugin name="gzplugin_joint_control" filename="libgazebo_joint_control.so"/>
        <plugin name="gzplugin_joint_state_client"
 ↪  filename="libgazebo_joint_state_client.so"/>
        <plugin name="dynamics_randomiser" filename="libdynamics_randomiser.so">
            <enable>false</enable>
            <link name="Doggy::head">
                <inertial>0.8 1.2</inertial>
            </link>
            <joint name="Doggy::pitch">
                <damping>0.75 1.25</damping>
                <friction>0.75 1.25</friction>
                <pid>0.8 1.2</pid>
            </joint>
        </plugin>
    </model>
    <model name="ball">
        <plugin name="ball_push" filename="libball_push.so">
            <velocity>6</velocity>
            <angle>-40</angle>
            <randomise>false</randomise>
            <distance_offset>-0.6 0.4</distance_offset>
            <height_offset>-0.4 0.4</height_offset>
            <angle_offset>60 120</angle_offset>
            <velocity_offset>-1 2.5</velocity_offset>
            <throw_angle_offset>-25 5</throw_angle_offset>
        </plugin>
        <plugin name="dynamics_randomiser" filename="libdynamics_randomiser.so">
            <enable>false</enable>
            <link name="ball::link">
                <inertial>0.9 1.1</inertial>
                <collision>
                    <mu>0.1 5</mu>
                    <bounce>0.5 0.7</bounce>
                </collision>
            </link>
        </plugin>
    </model>
    <plugin name="dynamics_proivder" filename="libdynamics_proivder.so"></plugin>
</world>
```

Listing 5.1: Gekürzter Auszug aus der Konfigurationsdatei der Simulationswelt mit den Modellen *Doggy* und *Ball*, ihren Plugins sowie deren Konfigurationsmöglichkeiten

5.2.1 Interaktion zwischen dem Agenten und der Simulation

Die Schnittstelle für die *Reinforcement Learning*-Algorithmen stellt eine *OpenAI Gym* [Bro+16] Umgebung dar. *Gym* liefert für einen Agenten eine Menge definierte Schnittstellen zur Interaktion mit einer beliebigen Umgebung. Die zur Interaktion genutzten

Methoden bestehen im Wesentlichen aus einer *Reset*-Methode, die bei jeder neuen Episode die Umgebung in den Grundzustand zurückversetzt und den ersten Umgebungszustand s_0 an den Agenten gibt sowie der *Step*-Methode, welche die vom Agenten auf Basis des Umgebungszustands gewählte Aktion ausführt. Hier werden zusätzlich noch der Reward r_{t+1} nach dem Ausführen der Aktion a_t und der neue Umgebungszustand s_{t+1} berechnet. Zusätzlich können hier auch noch Informationen aus der Umgebung zu Diagnosezwecken ausgegeben werden. Dafür werden folgende für den Agenten relevanten Funktionen implementiert:

```
1  class DoggySim:
2      def __init__:
3          initialise all relevant services and subscriber for interaction with the
   ↪  Gazebo simulation
4          initialise all variables
5          seed environment
6
7      def step(action):
8          take action
9          step world for n timesteps
10         observe enviroment state
11         calculate reward
12         decide if done
13         return observation, reward, done, additional information
14
15     def reset:
16         reset simulation via ROS service
17         reset all variables
18         take initial observation
19         return initial observation
```

Listing 5.2: Gekürzter Python-Pseudocode der DoggySim-v1 Umgebung

Der Übersicht halber werden Hilfsfunktionen wie die *render* oder *close* Funktion ausgelassen. Diese bieten die Möglichkeit die Visualisierung der Simulation durch Starten des Clients für *Gazebo* zu veranschaulichen sowie die Option, alle Instanzen der Simulation nach Beendigung eines Experiments herunterzufahren.
Für einen Agenten ist es von Vorteil, wenn Zeitschritte diskret ablaufen. Dazu wird das Modul *world_ stepper* implementiert, mit dem die Simulation eine definierte Menge an Simulationsschritten berechnet. Anschließend wird die Simulation wieder pausiert. Dies dient vor allem der Reproduzierbarkeit von Aktionen mit der Welt sowie der genaueren Festlegung der Zeitschritte zwischen Aktion und Observation.
Zur Beschleunigung der Trajektoriengenerierung bei Off-Policy Lernalgorithmen werden mehrere Simulationen in unterschiedlichen Namespaces gestartet. Diese laufen in jeweils eigenen Welten und *ROS*-Nodes. Jede Instanz der *DoggySim-v1* Umgebung läuft dabei in einem eigenen Simulationsnamespace.

5.3 Gesamtsystem

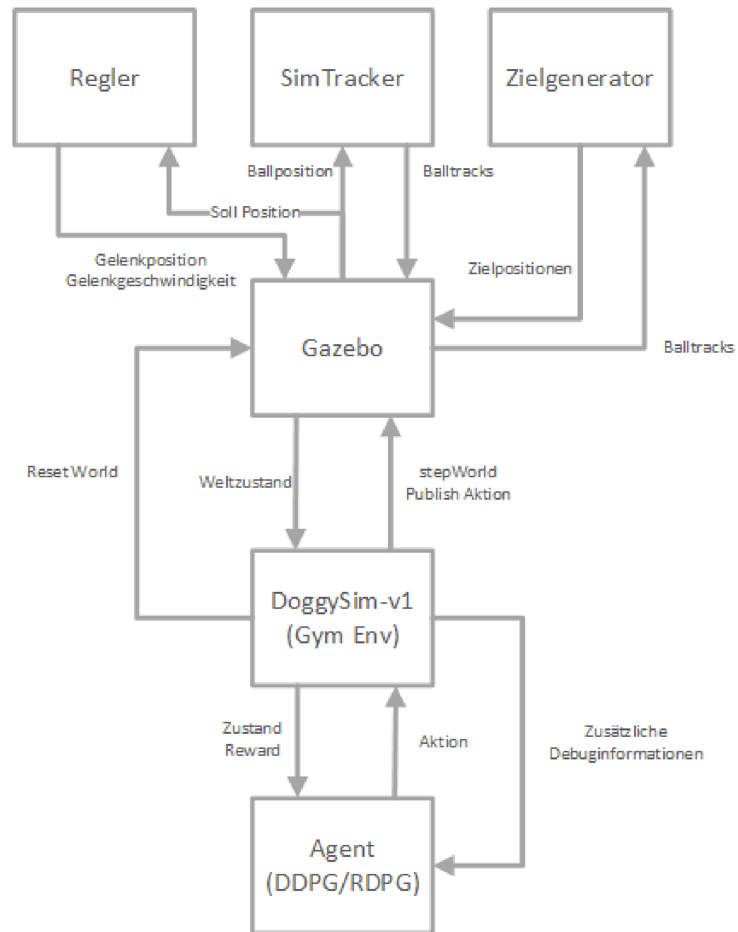


Abbildung 5.5: Übersicht über das Zusammenspiel vom Agenten mit der Simulation

6 Experimente

In diesem Kapitel wird der in Kapitel 4 besprochene Ansatz in verschiedenen Experimenten evaluiert. Die Ergebnisse werden in Hinsicht auf die Performance im definierten Szenario besprochen und untersucht.

6.1 Simulationsszenarien

Zunächst müssen für die Simulationsszenarien zwischen dem Trainings- und dem Testszenario unterschieden werden. Das Trainingsszenario sollte idealerweise einen möglichst großen Raum von Beispielen abdecken, damit es dem Agenten möglich ist, über das Problem hinweg zu generalisieren. In den Evaluationsszenarien wird darauf aufbauend das Verhalten unter verschiedenen Gesichtspunkten wie der Variation der Wurfparameter untersucht.

6.1.1 Definition Trainingsszenario

Im Trainingsszenario soll der Agent lernen, wie er einen Ball schlagen muss, um ihn gezielt zu einer definierten Zielposition zurückzuspielen. Damit eine möglichst gute Generalisierung über das Problem gewährleistet ist, wird zum einen die Art des Wurfs stark variiert. In einem echten Ballspiel kann kaum vorhergesagt werden, wie ein Mitspieler den Ball wirft. Daher soll hier ein möglichst großer Raum von möglichen Würfen abgedeckt werden. Als mögliche Parameter wurden die folgenden identifiziert: Entfernung zu Doggy in y-Richtung, die Wurfhöhe h, der Wurfwinkel α, die Wurfgeschwindigkeit v sowie ein Winkeloffset β zur der definierten Wurfposition. Der Bereich der Variation wird in Tabelle 6.1 detailliert aufgeschlüsselt. Die Referenzposition liegt dabei 50 cm vor dem Zielobjekt in einer Höhe von 1,5 m.

Parameter	Bereich
Entfernung in y-Richtung [m]	$[-0.6, 0.4]$ + Vorgabe
Wurfhöhe h [m]	$[-0.4, 0.4]$ + Vorgabe
Wurfwinkel α [°]	$[-25, 5]$ + Vorgabe
Wurfgeschwindigkeit v [m/s]	$[-1, 2.5]$ + Vorgabe
Winkeloffset β [°]	$[60, 120]$

Tabelle 6.1: Bereiche der Randomisierung der Ballwurfparameter

In Kapitel 4.1.1 wurden die Dynamikparameter beschrieben, die am wichtigsten für die Lernumgebung sind. Bei der Wahl des Bereichs, in dem diese Werte variiert werden, muss darauf geachtet werden, dass diese nicht das Lernproblem an sich beeinträchtigen. So hat Doggy schon in seiner Grundkonfiguration eine vergleichsweise hohe Reibung und Dämpfung in den Gelenken. Wird nun der Faktor, mit dem diese Werte randomisiert werden, zu groß gewählt, kann es im schlechtesten Fall zu dem Problem kommen, dass sich der Endeffektor nicht mehr bewegt, da die Regler die Gelenkreibung nicht mehr kompensieren können. Ebenso ist bei den Controller Gains ein maßvolles Intervall zu wählen, da diese eng mit der Gelenkreibung und -dämpfung verzahnt sind. Besonders zu kleine Werte können hier problematisch sein. Diese können ebenfalls dazu führen, dass sich der Roboter nicht bewegen kann, da die Gelenkreibung nicht überwunden werden kann.
Diese Problematik wurde im Vorfeld untersucht und auf Basis der daraus gewonnenen Erkenntnisse ist der in Tabelle 6.2 der Bereich aufgeschlüsselt, der für die jeweiligen Parameter gewählt wurde.

Parameter	Bereich
Linkmasse m [kg]	$[0.8, 1.2] \cdot$ Vorgabe
Gelenkreibung μ [N m]	$[0.75, 1.25] \cdot$ Vorgabe
Gelenkdämpfung d_s [N m s rad/s]	$[0.75, 1.25] \cdot$ Vorgabe
Controller Gains	$[0.8, 1.2] \cdot$ Vorgabe
Ballmasse m [kg]	$[0.1, 1.1] \cdot$ Vorgabe
Haftreibung Ball μ	$[0.1, 5]$
Restitutionskoeffizient k	$[0.5, 0.7]$

Tabelle 6.2: Bereiche der Randomisierung der Dynamikparameter

Aufgrund der vergleichsweise hohen Trainingszeit von acht bis zehn Stunden auf einem aktuellen Computersystem muss die Menge der Simulationsszenarien zur Vergleichbarkeit eingeschränkt werden. Für die Evaluation des Systems erscheint sinnvoll den zweiachsigen mit dem dreiachsigen Betrieb vergleichend zu untersuchen. Aus diesem Vergleich kann abgeleitet werden, wie gut der Agent die Redundanz der Yaw-Achse ausnutzen kann. In einigen Voruntersuchungen hat sich herausgestellt, dass der Agent aufgrund der hohen Reibung in den Gelenken teils nicht schnell genug ist, um den Ball mit ausreichender Geschwindigkeit in das Ziel zu schlagen. Daher soll zusätzlich die Auswirkung einer veränderten der Zieldistanz sowie die der Einfluss einer veränderten Startpose des Schlägers untersucht werden. Daraus ergeben sich die folgenden Testszenarien

- Zwei-Achsen bei $d_{goal} = 3{,}5\,\mathrm{m}$ und Startpose des Schlägers $0°$
- Zwei-Achsen bei $d_{goal} = 4\,\mathrm{m}$ und Startpose des Schlägers $0°$
- Drei-Achsen bei $d_{goal} = 3{,}5\,\mathrm{m}$ und Startpose des Schlägers $0°$
- Drei-Achsen bei $d_{goal} = 4\,\mathrm{m}$ und Startpose des Schlägers $0°$
- Drei-Achsen bei $d_{goal} = 3{,}5\,\mathrm{m}$ und Startpose des Schlägers $-10°$ auf der Pitch-Achse
- Drei-Achsen bei $d_{goal} = 4\,\mathrm{m}$ und Startpose des Schlägers $-10°$ auf der Pitch-Achse

Zusätzlich müssen auch noch die Hyperparameter für den *DDPG*-Agenten festgelegt werden. Die Grundlage für den Agenten bildet die Implementierung aus [Dha+17]. Als Orientierung dienen dabei die in [Pla17] beschriebenen Hyperparameter. Es kommt die in 4.1.2 beschriebene Architektur zum Einsatz. Als Optimierungsalgorithmus wird Adam [KB14] mit einer Batchgröße von 128 sowohl für das Actor- als auch für das Critic-Netz genutzt. Die Lernrate für den Actor beträgt 10^{-4} und für den Critic 10^{-3}, was den Werten aus [Lil+15] folgt. Der Soft-Update Wert τ beträgt 10^{-3}. Zur Generierung eines Trainingslaufs werden acht parallele Simulationen gestartet. Ein Durchlauf besteht hierbei aus 500000 Zeitschritten mit einer Epochenlänge von 5000 Zeitschritten und einer maximalen Episodenlänge von 100 Zeitschritten. Ein Zeitschritt entspricht 0,04 s in Simulationszeit. In Tabelle 6.3 sind die genutzten Werte noch einmal übersichtlich dargestellt.

Hyperparameter	Wert
Optimierungsalgorithmus	Adam [KB14]
Batchgröße	128
Actor Lernrate	10^{-4}
Critic Lernrate	10^{-3}
Soft-Updates der Zielnetze τ	10^{-3}
Replaypuffergröße	10^{5}
Discountfaktor γ	0.98
Trainingsschritte	500000
Epochenlänge	5000 Schritte
Maximale Episodenlänge T	100
Simulationszeitschritte t	0,04 s
Solverschrittgröße Δt	0,002 s

Tabelle 6.3: Hyperparameter für die Experimente

Der *RDPG*-Agent konnte in Voruntersuchungen keine konsistenten Ergebnisse produzieren, ob dies an ungünstigen Hyperparametern oder durch Fehler in der Implementierung bedingt ist, konnte nicht abschließend festgestellt werden.

6.1.2 Definition Testszenario

Das Testszenario unterscheidet sich vom Trainingsszenario nur in der Hinsicht, als dass der Ball in einem Raum von $[-0{,}5\,\mathrm{m}, 0{,}5\,\mathrm{m}]$ auf der x-Achse vor dem Ziel verschoben wird im Gegensatz zu der Bewegung auf einem Halbkreis vor dem Roboter im Training. Dies soll möglichst nahe an einem echten Wurf von der definierten Zielposition g liegen. Ansonsten werden die in Tabelle 6.1 definierten Werte für die Würfe genutzt. Evaluiert werden die drei Policies in jedem Szenario für 100 Episoden.
Im Testszenario soll zum einen untersucht werden, wie sich die Performance der Policies

zueinander in Bezug auf Robustheit bei verschiedenen Würfen und in Bezug auf eine Umgebung mit und ohne Dynamikrandomisierung verhält. Zusätzlich zur finalen Policy wird noch die „beste“ Policy untersucht und mit ersterer verglichen. Die beste Policy ist diejenige, die im Training den höchsten Referenz-Q-Wert erreichte.

6.2 Ergebnisse der Simulationsexperimente

Nachfolgend werden die Ergebnisse des Trainings in der Simulation ausgewertet und diskutiert. Dabei werden jeweils die Szenarien mit gleichem Zielabstand zusammengefasst. Zusätzlich wird untersucht, inwiefern der Agent sinnvolle Schlagbewegungen in den verschiedenen Szenarien erlernt hat.

6.2.1 Szenario Zieldistanz 3,5 m

6.2.1.1 Training

Bei der Betrachtung der Ergebnisse im Training zeigt sich, dass die zwei Achsen Policy am Ende des Trainings ein leicht besseres Ergebnis bei den Zieltreffern erreicht, als beide drei Achsenpolicies.

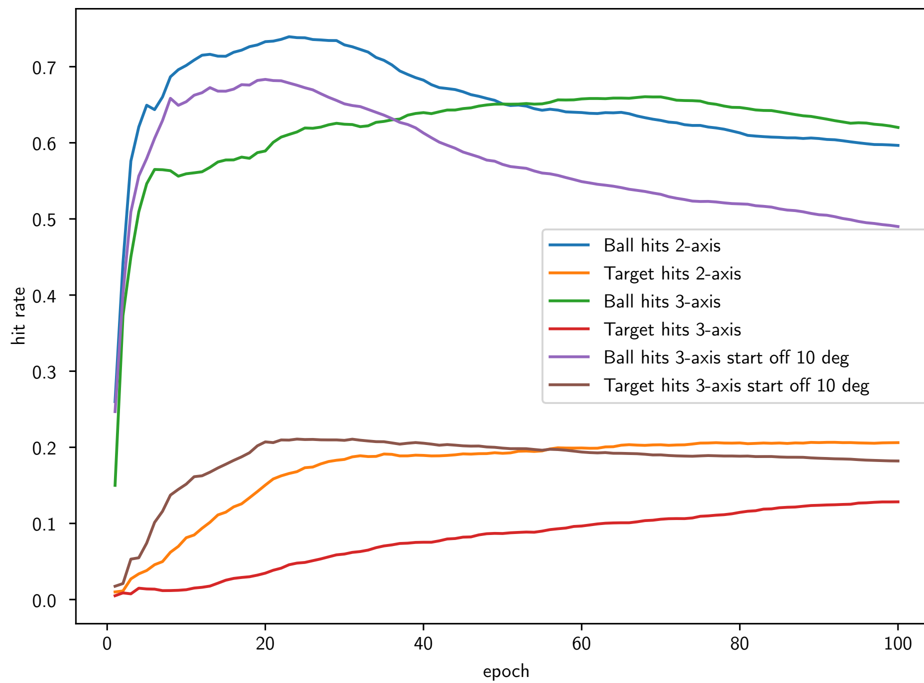


Abbildung 6.1: Durchschnittliche über alle Episoden. Unterteilt in die verschiedenen Bewegungsmöglichkeiten: Balltreffer und Zieltreffer im Szenario 3,5 m

Zusätzlich ist im Training zu beobachten, dass es ab der Hälfte des definierten Trainingszeitraums bei den Balltreffern zu einem leichten Underfitting kommt. Erkennbar ist dies an der absinkenden Balltrefferrate ab etwa Epoche 40, bei gleichzeitig konstanter Zieltrefferrate.

Dieses Underfitting Phänomen zeigt sich besonders deutlich im durchschnittlichen Reward nach Zeitschritten. Hier ist ein Abfallen des Rewards nach etwa der Hälfte der Zeitschritte zu beobachten.

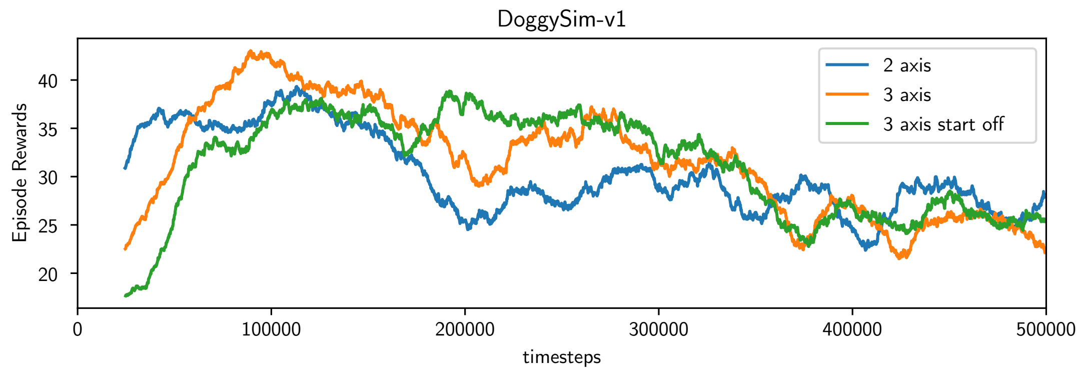
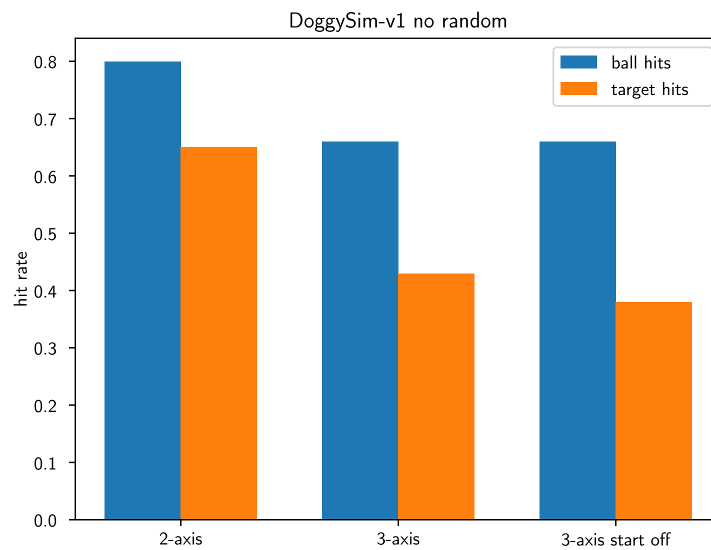
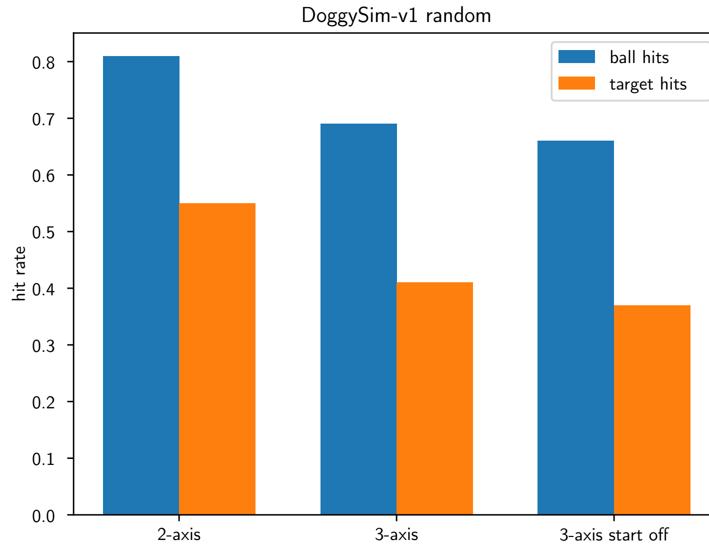


Abbildung 6.2: Durchschnittlicher Reward in Zeitschritten einer Simulationsinstanz, leicht geglättet mit einem Fenster von 250 Zeitschritten

6.2.1.2 Test - finale Policy



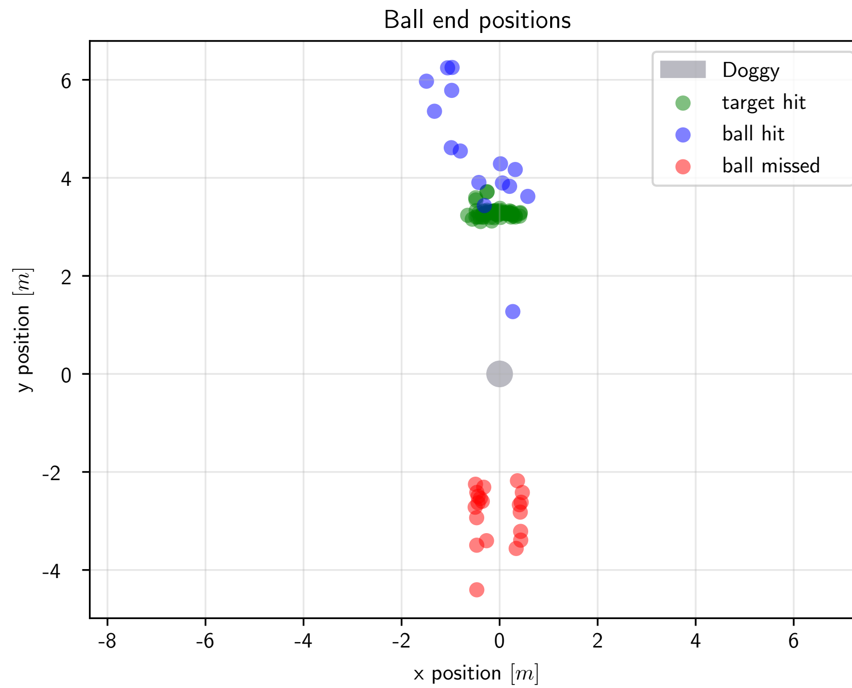
(a) Trefferrate ohne Randomisierung im Testszenario



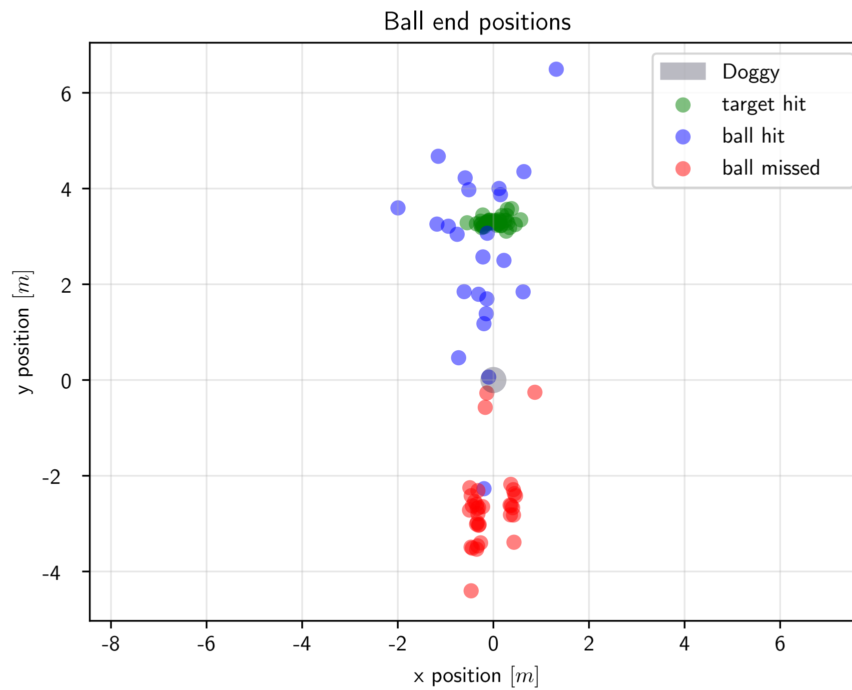
(b) Trefferrate mit Randomisierung im Testszenario

Abbildung 6.3: Trefferraten bei 3,5 m Zielentfernung

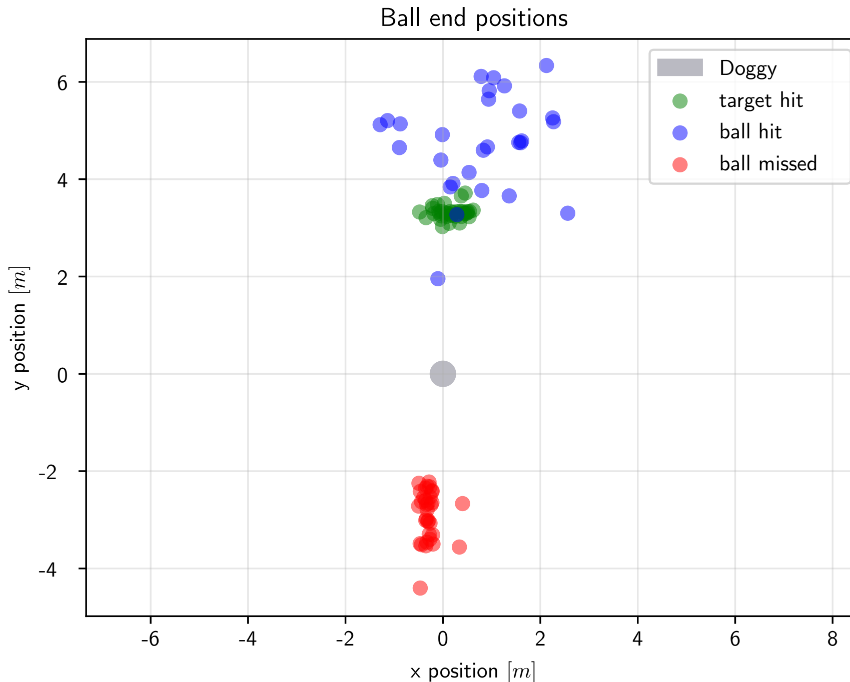
Bei der Auswertung der finalen Policies zeigt sich ein überraschendes Ergebnis. Die auf zwei Achsen beschränkte Policy erzielt eine deutlich bessere Performance als die drei Achsen Policies. Die Trefferrate der zwei Achsen Policy auf das Ziel liegt im Szenario mit fester Dynamik mit 65% Treffern auf das Ziel etwas mehr als 20 Prozentpunkte über den drei Achsen Polcies und im Szenario mit Randomisierung immer noch rund 15 Prozentpunkte über den drei Achsen Policies.
Ebenfalls ist zu beobachten, dass die veränderte Startposition von $-10°$ auf der Pitch-Achse vom Agenten nicht ausgenutzt wird. Hier wäre eine verbesserte Schlagbewegung erwartbar gewesen, da der Agent in diesem Fall mehr Zeit hat Geschwindigkeit aufzubauen und im Anschluss den Ball zu schlagen. Auch für Bälle die sehr nah an der Nullpose des Schlägers den Arbeitsraum schneiden sollten mit diesem Verhalten besser spielbar sein. Dennoch zeigen die Ergebnisse, dass damit kaum eine Verbesserung erreicht wurde, im Gegenteil verhält sich diese Policy im gegebenen Szenario sogar etwas schlechter.
In der Analyse der Trefferraten zeigt sich ebenfalls, dass die Dynamikrandomisierung es für das System insgesamt etwas schwerer macht, den Ball beim Schlag zu kontrollieren. Die Balltrefferraten unterscheiden sich zwischen den beiden Szenarien kaum, wohingegen die Trefferrate auf das Ziel deutlich abfällt.



(a) Auftreffpunkte zwei Achsenschlagbewegung



(b) Auftreffpunkte drei Achsenschlagbewegung



(c) Auftreffpunkte drei Achsenschlagbewegung mit Startpose $\theta = (0°, -10°, 0°)$

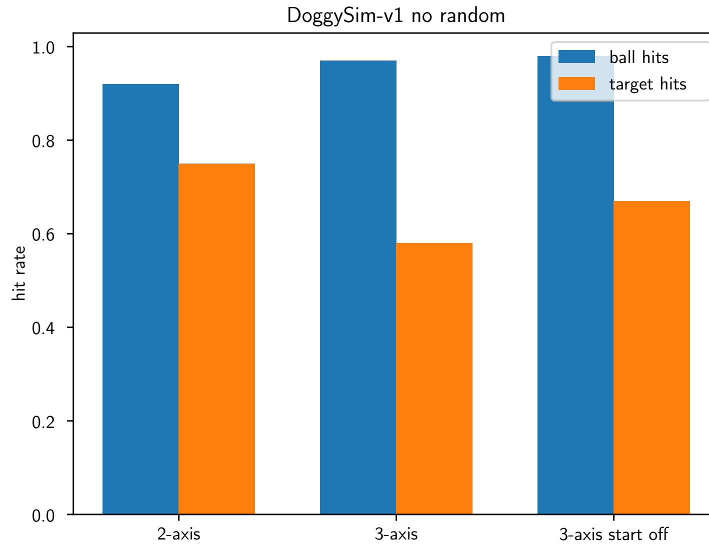
Abbildung 6.4: Position der Bälle am Episodenende

In der Analyse der gespielten Bälle zeigt sich, dass die zwei Achsenpolicy eine hohe Präzision der gespielten Bälle um und auf das Ziel hat. In der Tendenz spielt diese Policy die Bälle sogar etwas zu stark und somit leicht hinter das Ziel. Die drei Achsenpolicy zeigt ebenfalls eine hohe Präzision der gespielten Bälle um das Ziel herum. Dabei ist auffällig, dass einige Bälle deutlich vor dem Ziel liegen ($d_{goal} > 2\,\text{m}$). Das deutet darauf hin, dass hier zu wenig Geschwindigkeit in der Schlagbewegung aufgebaut wurde, um die Bälle ins Ziel zu spielen. Bei der drei Achsenpolicy mit veränderter Startpose zeigt sich, dass diese die Bälle tendenziell etwas zu weit spielt, was auf die veränderten Startbedingungen zurückzuführen ist.

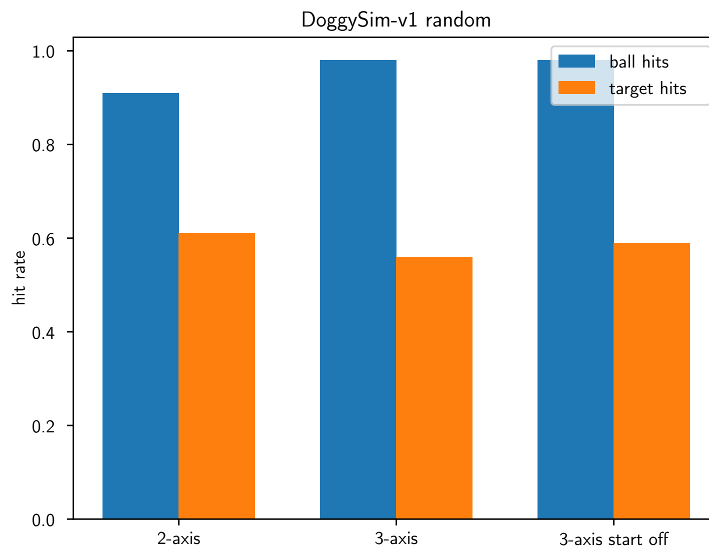
6.2.1.3 Test - beste Policy

Da im Training ein leichtes Underfitting beobachtet wurde, wird zusätzlich zu den finalen Policies die nach Trainingsdaten beste Policy der jeweiligen Bewegungsszenarien untersucht. Als Maß hierfür wurde der maximale durchschnittliche Q-Wert des Zielnetzes nach Abschluss einer Epoche genutzt. Dieser Wert gibt an, wie gut die Aktionen des Agenten in Bezug auf den zu erwartenden Reward sind. Im Falle der zwei Achsenpolicy

ist dies die Policy aus Epoche 33, bei der drei Achsenpolicy die Policy aus Epoche 32 und bei der drei Achsenpolicy die Policy aus Epoche 31.



(a) Trefferrate ohne Randomisierung im Testszenario



(b) Trefferrate mit Randomisierung im Testszenario

Abbildung 6.5: Trefferraten bei 3,5 m Zielentfernung mit der jeweils besten Policy

Bei Betrachtung der erreichten Trefferraten fällt auf, dass diese insgesamt etwas höher liegen, als die der finalen Policies. Insbesondere die drei Achsenpolicy mit veränderter Startposition verhält sich hier besser als die finale Variante.
In Tabelle 6.4 sind die Trefferaten noch einmal übersichtlich dargestellt.

Policy	Balltreffer (keine Randomisierung)	Zieltreffer (keine Randomisierung)	Balltreffer (mit Randomisierung)	Zieltreffer (mit Randomisierung)
2-Achsen final	80 %	65 %	81 %	55 %
2-Achsen beste	92 %	75 %	91 %	61 %
3-Achsen final	66 %	43 %	66 %	41 %
3-Achsen beste	97 %	57,9 %	98 %	56 %
3-Achsen final off	66 %	38 %	66 %	37 %
3-Achsen beste off	98 %	67 %	98 %	59 %

Tabelle 6.4: Vergleich der Trefferraten von finaler zu bester Policy im 3,5 m Szenario

6.2.2 Szenario Zieldistanz 4 m

6.2.2.1 Training

Im Szenario mit einer Zieldistanz von 4 m zeigt sich ein ähnliches Bild wie im vorigen Szenario. Auch hier ist die Performance der zwei Achsenpolicy insgesamt leicht besser als, die der drei Achsenpolicy.

Auch im 4 m Szenario zeigt sich bei der Betrachtung der Rewards ein Underfitting gegen Ende des Trainings. Dieses Verhalten ist im Vergleich zum vorigen Szenario noch ausgeprägter. Der Agent scheint hier ab der Hälfte des Trainings immer seltener den Ball und das Ziel zu treffen. Dies zeigt sich besonders im Fall der drei Achsenpolicy mit veränderter Startposition. Hier ist bei etwa 270000 Zeitschritten ein deutlicher Abfall des Rewards pro Episode zu beobachten.

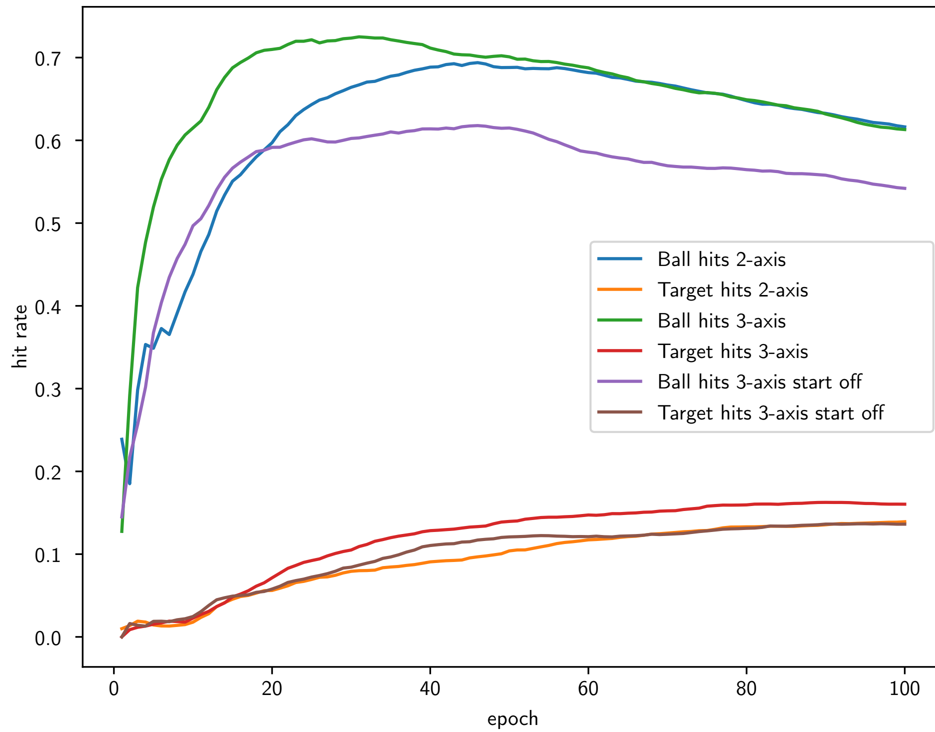


Abbildung 6.6: Durchschnittliche über alle Episoden. Unterteilt in die verschiedenen Bewegungsmöglichkeiten: Balltreffer und Zieltreffer im Szenario Szenario 4 m

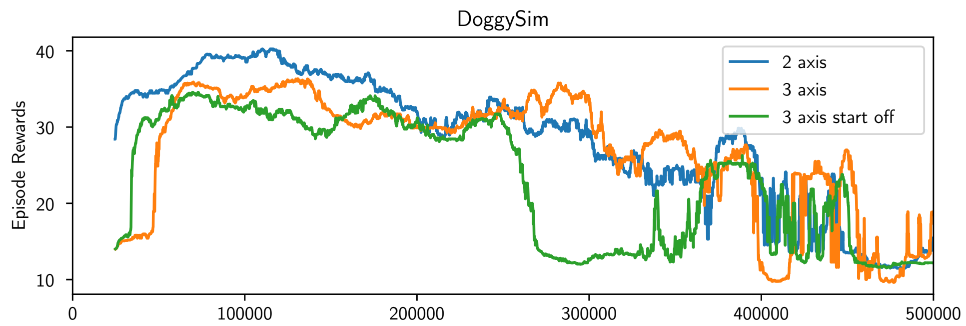
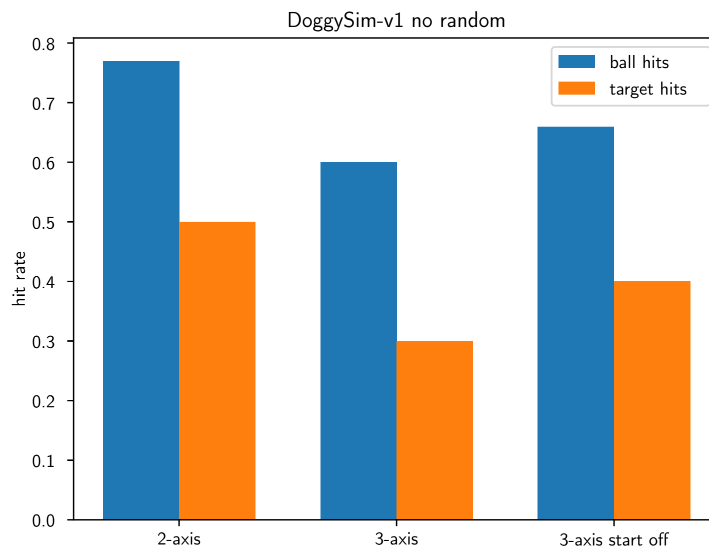


Abbildung 6.7: Durchschnittlicher Reward in Zeitschritten einer Simulationsinstanz, leicht geglättet mit einem Fenster von 250 Zeitschritten

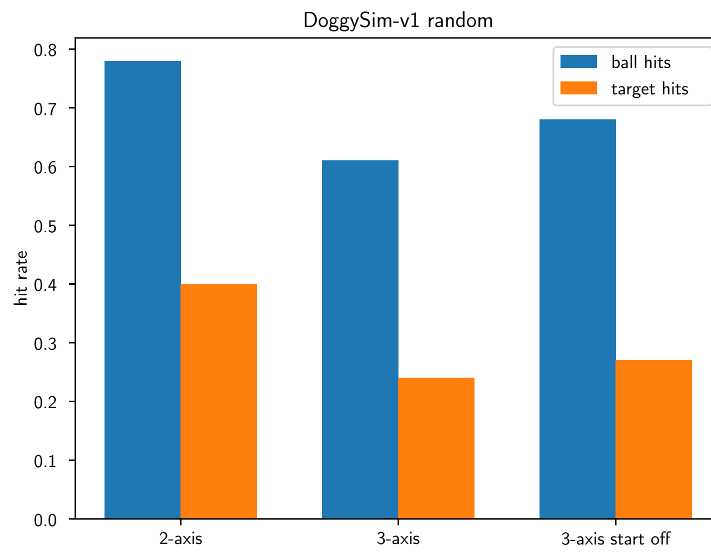
6.2.2.2 Test - finale Policy

Im Folgenden werden die Endpositionen der Ballwürfe für die drei gelernten Policies im vorliegenden Szenario untersucht.

Hier zeigt sich ein leicht anderes Bild im Vergleich zum vorigen Szenario. Die Trefferraten auf das Ziel liegen alle unter 50 %. In diesem Szenario hat die veränderte Startpose von $-10°$ positive Auswirkungen auf die Zieltreffer. Das Verhalten ist hier etwas besser als bei der normalen Drei-Achsenpolicy. Außerdem Generalisierung ist bei beiden drei Achsenpolicies schlechter als bei der zwei Achsenpolicy, wenn man das randomisierte Testszenario betrachtet wird.

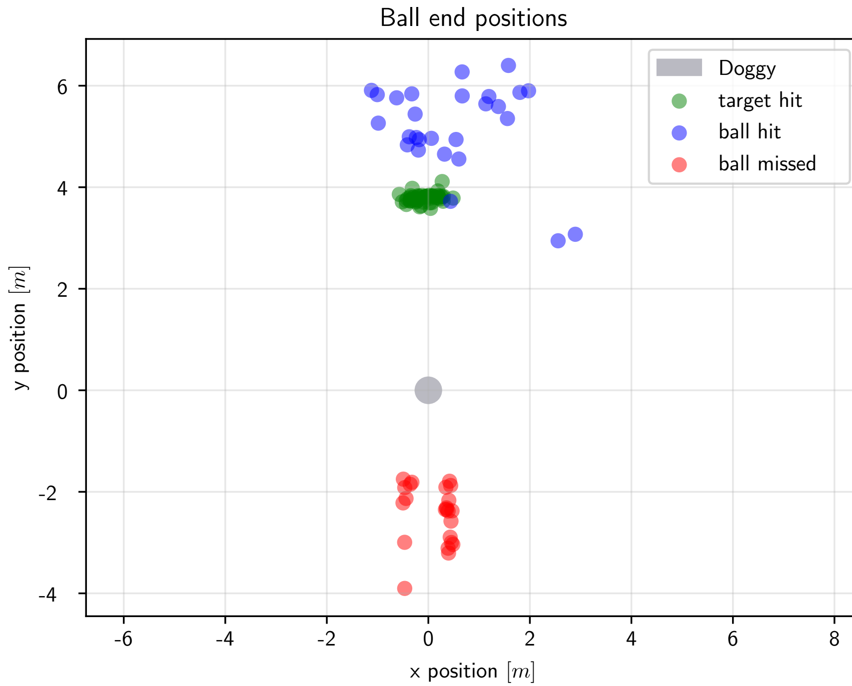


(a) Trefferrate ohne Randomisierung im Testszenario

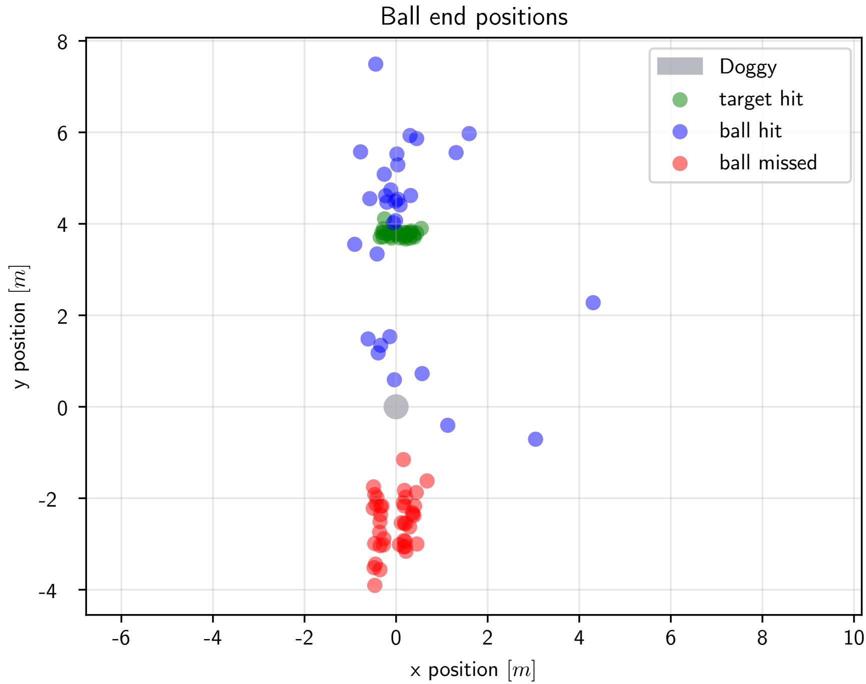


(b) Trefferrate mit Randomisierung im Testszenario

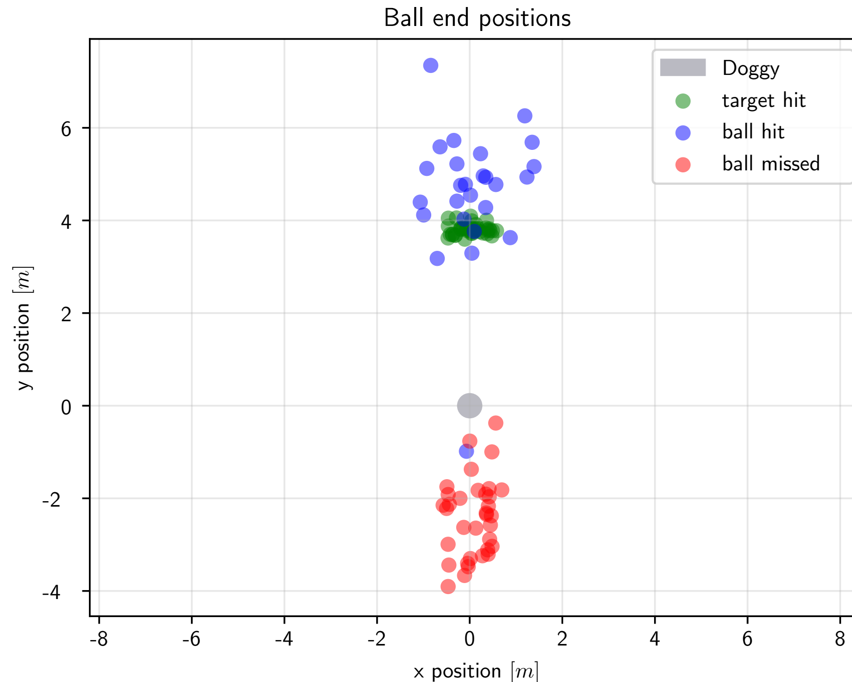
Abbildung 6.8: Trefferraten bei 4 m Zielentfernung



(a) Auftreffpunkte zwei Achsenschlagbewegung



(b) Auftreffpunkte drei Achsenschlagbewegung



(c) Auftreffpunkte drei Achsenschlagbewegung mit Startpose $\theta = (0°, -10°, 0°)$

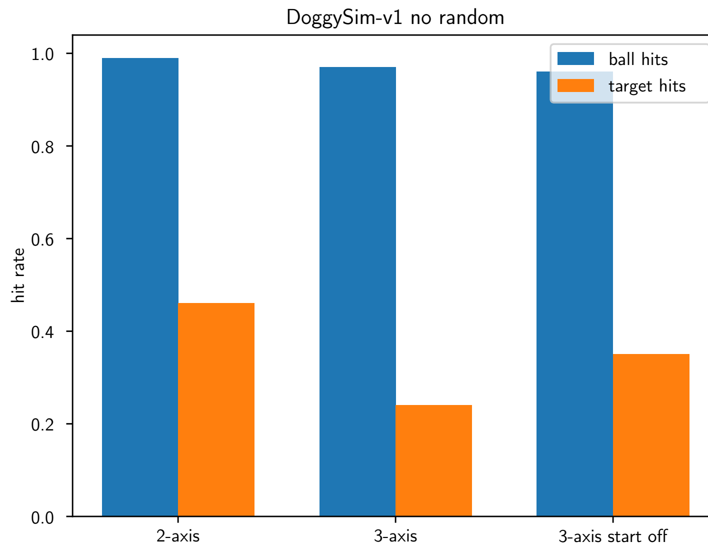
Abbildung 6.9: Position der Bälle am Episodenende

Insgesamt ist im 4 m Szenario zu beobachten, dass die Präzision der gespielten Bälle deutlich abgenommen hat. Dies ist wahrscheinlich auf die Limitierung in der Beweglichkeit der Gelenke zurückzuführen. Hierbei ist es wahrscheinlich, dass das System nicht weit genug ausholen kann, um den Ball mit einer ausreichenden Geschwindigkeit zu schlagen. Dies ist insbesondere bei der dreiachsigen Policy zu beobachten. Diese spielt die Bälle häufiger vor das Ziel als die zweiachsige Policy und dreiachsige Policy mit veränderter Startposition. Insgesamt kann man auch eine größere Streuung der gespielten Bälle um das Zielobjekt beobachten. Eine genauere Untersuchung hat hier gezeigt, dass das System zusätzlich zu den Limitierungen in der Beweglichkeit die Bälle nicht präzise genug schlägt, was auch zu einem zu frühen Aufkommen auf dem Boden führt. Zusammen mit der zu geringen Schlaggeschwindigkeit scheinen dies die Ursachen für die verringerte Spielperformance zu sein.

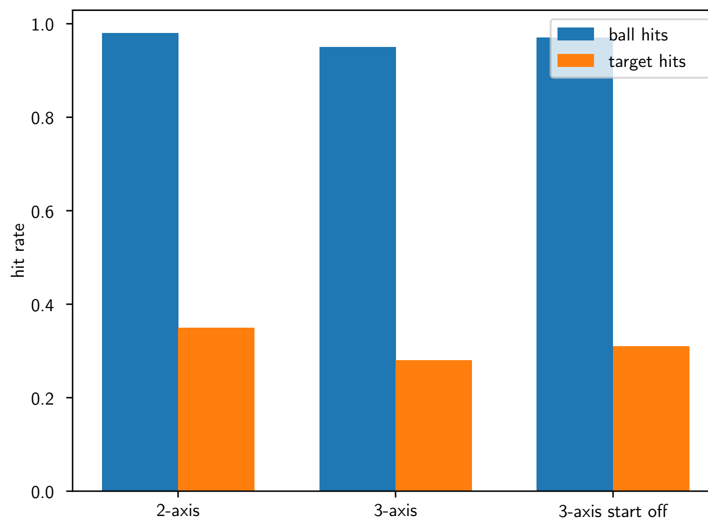
6.2.2.3 Test - beste Policy

Die besten Policies wurden nach dem gleichen Verfahren, welches in 6.2.1.3 besprochen wird, ausgewählt. Im Falle der zwei Achsenpolicy ist dies die Policy aus Epoche 26, bei

der drei Achsenpolicy die Policy aus Epoche 24 und bei der drei Achsenpolicy die Policy aus Epoche 28.



(a) Trefferrate ohne Randomisierung im Testszenario



(b) Trefferrate mit Randomisierung im Testszenario

Abbildung 6.10: Trefferraten bei 4 m Zielentfernung mit der jeweils besten Policy

Die besten Policies im 4 m Szenario unterscheiden sich nur in der insgesamt höheren Balltrefferrate. Die Treffer auf das Ziel sind nahezu gleich zur finalen Policy. Hier hatte das Underfitting also nur Auswirkungen auf das Verhalten beim Treffen von schwierig zu spielenden Bällen und nicht auf das Gesamtergebnis wie in 6.2.1.3. Die Zieltrefferraten sind in der finalen Policy sogar insgesamt etwas besser (vgl. Tabelle 6.5).

Policy	Balltreffer (keine Randomisierung)	Zieltreffer (keine Randomisierung)	Balltreffer (mit Randomisierung)	Zieltreffer (mit Randomisierung)
2-Achsen final	77 %	55 %	81 %	55 %
2-Achsen beste	99 %	41 %	98 %	35 %
3-Achsen final	60 %	30 %	61 %	24 %
3-Achsen beste	97 %	24 %	95 %	28 %
3-Achsen final off	66 %	40 %	68 %	27 %
3-Achsen beste off	96 %	35 %	97 %	31 %

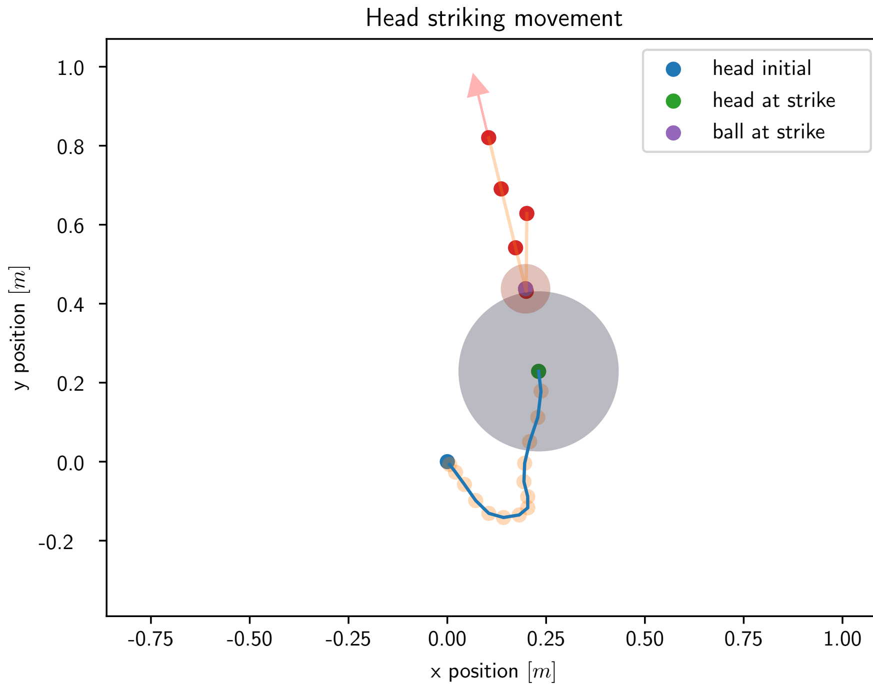
Tabelle 6.5: Vergleich der Trefferraten von finaler zu bester Policy im 4 m Szenario

6.2.3 Untersuchung der gelernten Schlagbewegungen

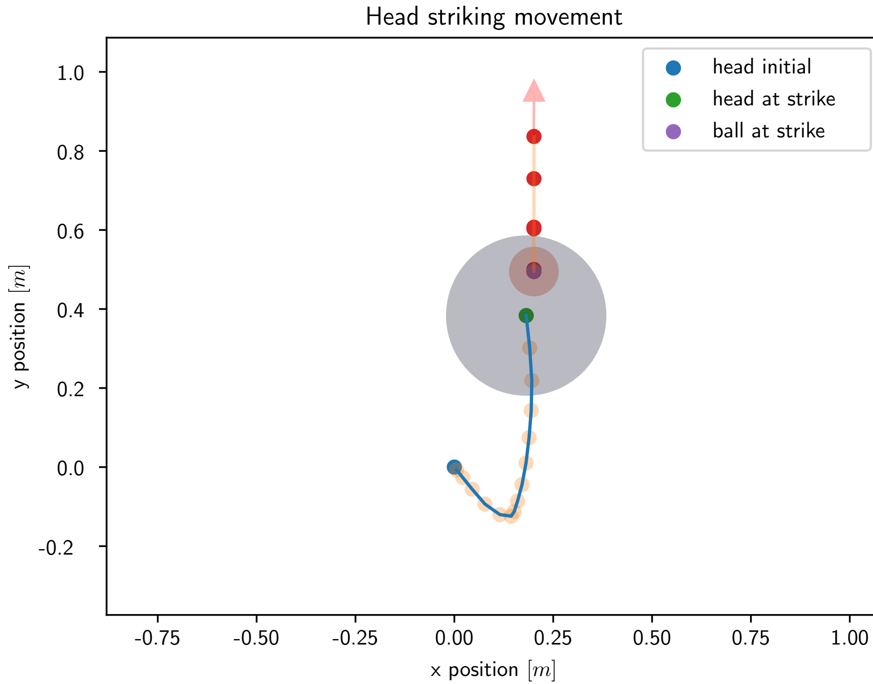
Nachfolgend soll untersucht werden, ob die Anpassung der Rewardfunktion in Kapitel 4.1.3 es dem Agenten ermöglicht hat, sinnvolle Schlagbewegungen zu erlernen. Dabei wird vergleichend untersucht, wie sich die Schlagbewegungen auf zwei Achsen beschränkt und mit allen drei Achsen verhalten, sowie mit einer veränderten Startposition des Schlägers von $-10°$ auf der Pitch-Achse.

In der qualitativen Analyse der Schlagbewegungen hat sich gezeigt, dass der Agent lernt Bälle, die sehr flach unterhalb des Schlägers den Arbeitsraum schneiden, zu ignorieren. Diese Bälle sind auch physisch aufgrund der Bauweise des Roboters quasi kaum zu schlagen und zurückzuspielen. In der Tendenz ignoriert der Agent auch Bälle die weit am linken und rechten äußeren Rand des Arbeitsraums liegen. In diesen Fällen hat der Roboter aufgrund der hohen Gelenkreibung nicht genug Zeit, um auszuholen und genug Geschwindigkeit aufzubauen, um den Ball gezielt zurückzuspielen.

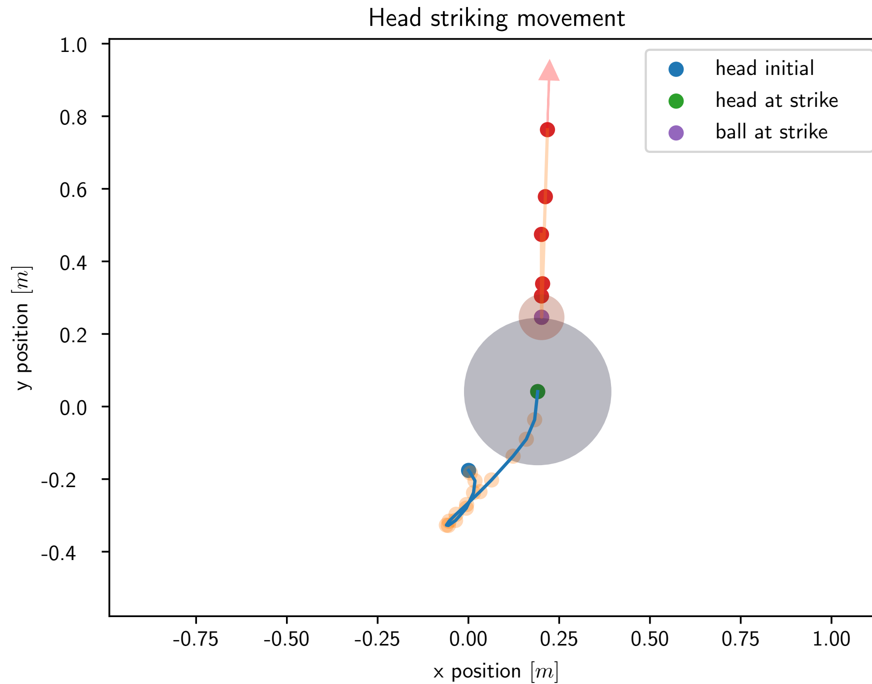
In den nachfolgenden Abbildungen werden jeweils Schlagbewegungen in verschiedenen Situationen dargestellt. In der Darstellung entspricht ein Punkt je einer Position des Schlägers bzw. des Balls zu einem Zeitpunkt t mit $\Delta t = 0{,}04\,\text{s}$. Schneidet der Ball den Endeffektor, so ist dies der Top-Down-Ansicht geschuldet. In diesen Fällen wurde der Ball entweder von unten oder von oben relativ zur Kugel geschlagen.



(a) Zwei Achsenschlagbewegung



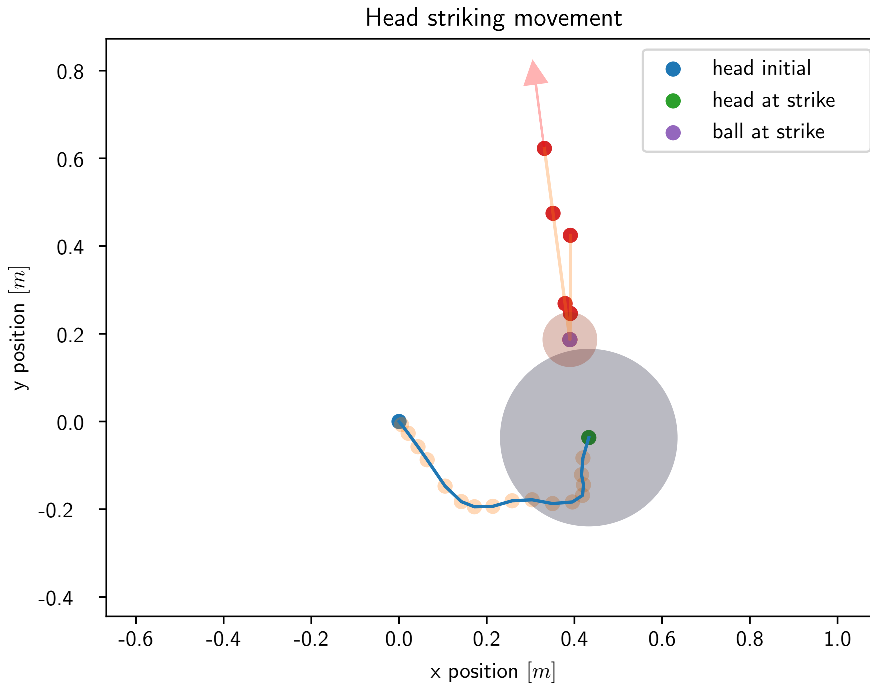
(b) Drei Achsenschlagbewegung



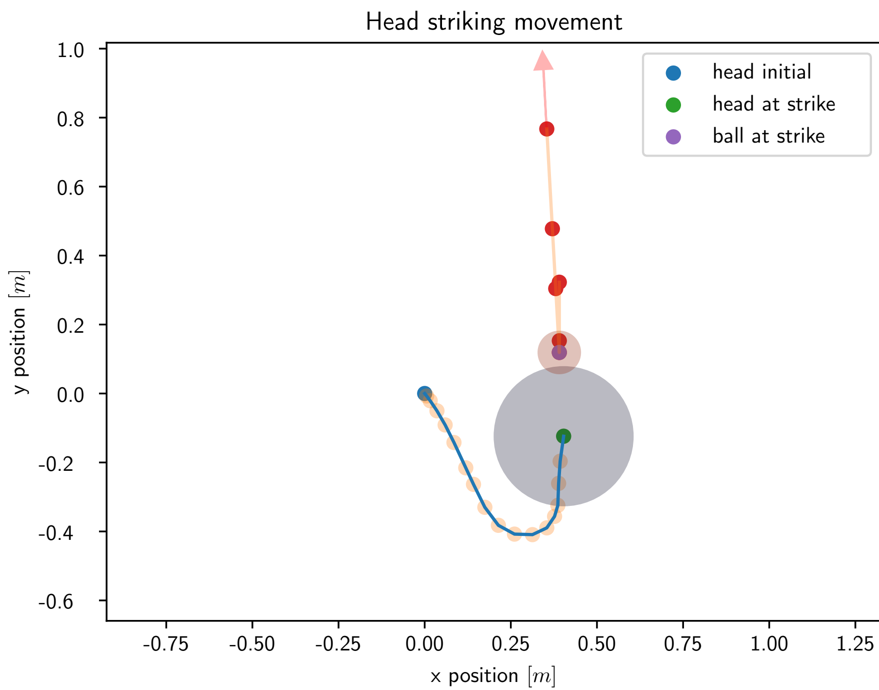
(c) Drei Achsenschlagbewegung mit Startpose $\theta = (0°, -10°, 0°)$

Abbildung 6.11: Vergleich dreier Schlagbewegungen mit ähnlicher, mittiger Ballflugbahn

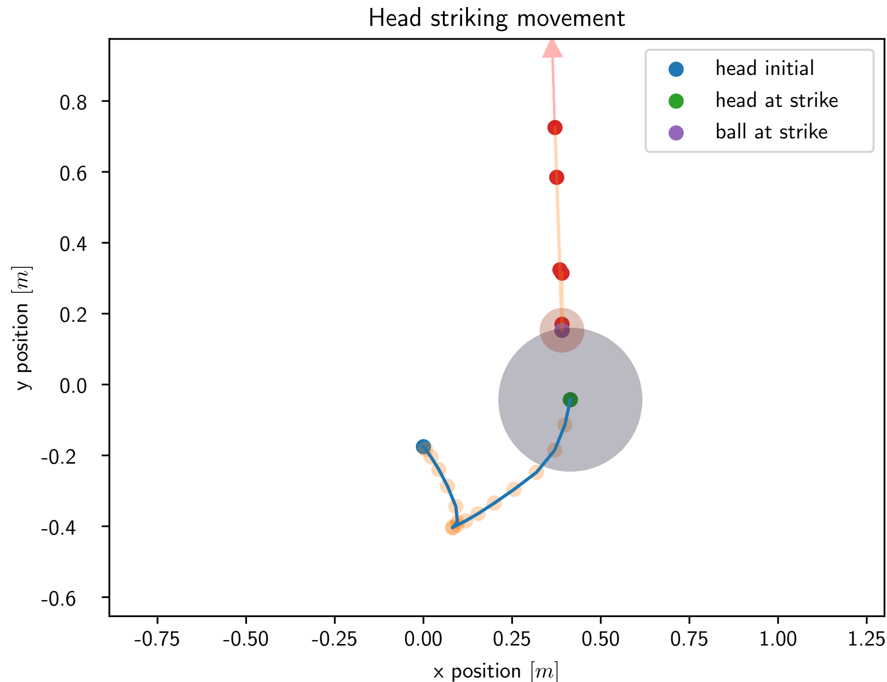
Die zwei in Abbildung 6.11 abgebildeten Bewegungen sind zur besseren Vergleichbarkeit der Schlagbewegungen sehr ähnliche Ballwürfe, was durch Setzen eines Randomseed in der Simulationsumgebung erreicht wurde. Bei der Betrachtung der Schlagbewegungen fällt auf, dass die Schlagbewegung mit drei Achsen eine insgesamt sauberere Bewegung des Schlägers produziert. Der Roboter holt erst aus und verweilt dann an dieser Position, um im Anschluss mit einer hohen Geschwindigkeit den Ball zu schlagen. Hier ist besonders in diesem Fall auffällig, dass der Eintrittsvektor des Balls in zwei Dimensionen von oben betrachtet fast dem Austrittsvektor entspricht.
Bei der Schlagbewegung mit zwei Achsen zeigt sich eine insgesamt etwas unsauberere Bewegung im Vergleich zur Bewegung mit drei Achsen. Man kann hier deutlich erkennen, dass der Agent nachregelt und nicht so lange an der Ausholposition verweilt. Aus dieser Bewegung resultiert auch der größere Drall des Balls zur linken Seite.



(a) Zwei Achsenschlagbewegung



(b) Drei Achsenschlagbewegung



(c) Drei Achsenschlagbewegung mit Startpose $q = (0°, -10°, 0°)$

Abbildung 6.12: Vergleich dreier Schlagbewegungen mit ähnlicher, außen rechts liegender Ballflugbahn

Im in Abbildung 6.12 dargestellten Beispiel kommt der Ball von deutlich weiter rechts. Im Fall der zwei Achsenschlagbewegung ist eine unsaubere Bewegung zu erkennen. Der Roboter holt hier nicht weit aus, sondern bewegt den Schläger in einer zackigen Bewegung in Richtung des erwarteten Schlagpunktes. Daraus resultiert auch eine im Vergleich zu den Schlagbewegungen mit drei Achsen geringere Austrittsgeschwindigkeit des Balls mit einem leichten Drall nach links. Bei der Bewegung mit drei Achsen aus der Nullposition ähnelt die Bewegung sehr der in 6.11b dargestellten Bewegung, auch hier holt der Roboter in einer weichen Bewegung aus und schlägt den Ball mit einer höheren Geschwindigkeit als in der zwei Achsenschlagbewegung.

6.3 Zwischenfazit

Zusammenfassend betrachtet ist festzustellen, dass der Agent in der Lage war nur im zweiachsigen Fall ein gutes Verhalten zu erlernen. Sowohl die dreiachsige Policy mit Start in der Nullposition als auch die versetzte Startposition haben entgegen der Erwartung

durchweg deutlich schlechtere Ergebnisse erzielt. Dies kann mehrere Gründe haben. So verändert sich durch starke Nutzung der Yaw-Achse der erreichbare Arbeitsraum deutlich. In einer qualitativen Beobachtung des dreiachsigen Verhaltens in der Simulation hat sich gezeigt, dass der Agent bei frontalen Bällen, die je weiter links oder rechts den Arbeitsraum schneiden, sehr stark die Yaw-Achse zu nutzen scheint. Dieses Verhalten führt dabei häufig dazu, dass der Agent die Bälle nicht mehr mit ausreichender Geschwidigkeit schlagen kann oder sie verfehlt, da er hier häufig die Roll- und Pitch-Achse nicht weit genug oder zu weit bewegt.
Betrachtet man dabei die Policies, bevor das Underfitting auftritt, so ergibt sich ein insgesamt etwas besseres Bild, wobei auch hier die zwei Achsenbewegung leicht überlegen ist. Die drei Achsenbewegung ist hier insofern ähnlich gut, als dass sie in der Lage ist mehr Bälle zu spielen, was in einem Spielkontext ein deutlich besseres Bild abliefert, als wenn der Agent Würfe einfach ignoriert.
Bei der Betrachtung der besten Policies hat sich gezeigt, dass diese insgesamt deutlich bessere Ergebnisse erzielen, was die Beobachtung des Underfittings untermauert. Versuche mit mehr Neuronen und veränderten Hyperparamtern haben hier keine signifikante Verbesserung bewirkt.

6.4 Realsystemszenario

Aufgrund von Limitierungen im aktuellen Softwarestand auf dem realen Robotersystem konnte nur eine Reihe von kleineren Experimenten durchgeführt werden, welche die prinzipielle Anwendbarkeit des gelernten Verhaltens auf dem Realsystem betrachten. So ist im aktuellen Stand der Software der *MHT* nicht auf die Nutzung der Yaw-Achse während des Verfolgens von Bällen kalibriert, was hier das Ergebnis deutlich verfälschen würde. Daher soll für das dreiachsige Verhalten ein simulierter Track genutzt werden. Dabei soll insbesondere untersucht werden, wie gut sich die gelernten Bewegungen auf das echte System übertragen lassen. Dafür wird eine drei Achsenpolicy, die in Kapitel 6.2.1 betrachtet wurde, auf dem Realsystem ausgerollt und mit einem in der Simulation aufgezeichneten Balltrack bespielt. Die reale Bewegung wird mit der simulierten Bewegung verglichen. Sind hier große Abweichungen zu erkennen, ist davon auszugehen, dass das in der Simulation gelernte Verhalten nicht gut in die Realität übertragbar ist.
Dasselbe Verfahren wird zur Vergleichbarkeit auch für die in 6.2.1 gelernte zwei Achsenpolicy angewendet. Die drei Achsenpolicy mit veränderter Startposition wird hier nicht weiter untersucht, da sie schon in der Simulation kaum eine Verbesserung der Trefferrate bewirkt hat.

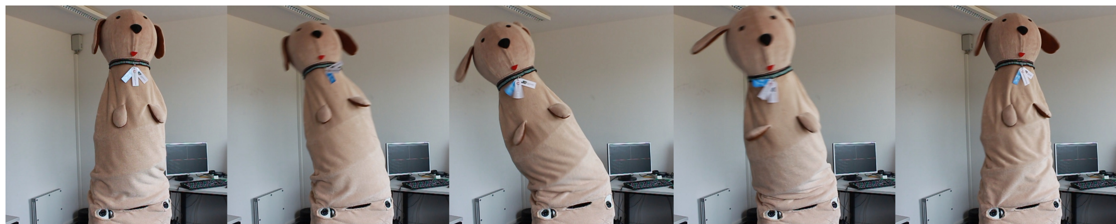
6.5 Ergebnisse der Experimente auf dem realen System

Für die Nutzung der gelernten Policies auf dem Roboter mussten kaum Anpassungen durchgeführt werden. Einzig die Gelenklimits, mit denen die Aktionsausgabe des *Actor*-Netzes skaliert wird, mussten um durchschnittlich 10° verringert werden, da sonst das Überschwingen des P-Reglers die Sicherheitslimits in der Mikrocontroller Firmware verletzt, was zum Blockieren aller folgenden Aktionen und dem Anfahren der Nullposition führt.

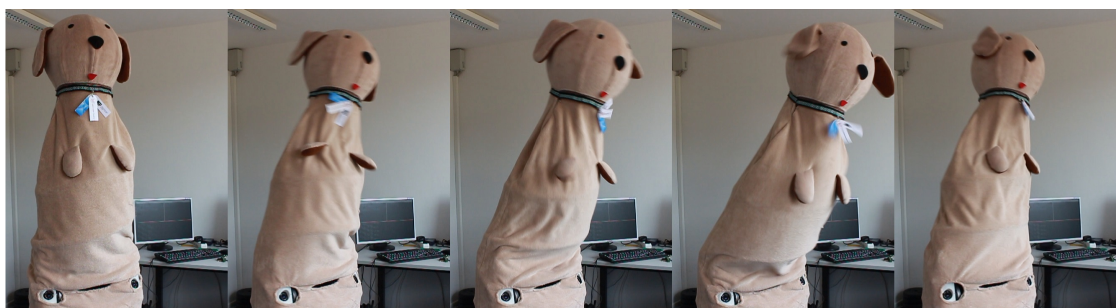
6.5.1 Qualitative Analyse der Schlagbewegungen

Im Folgenden werden die Versuchsergebnisse für die Übertragbarkeit der in Simulation gelernten Schlagbewegungen zunächst qualitativ betrachtet. Dabei werden jeweils ein positives und ein negatives Beispiel untersucht. Die Versuche wurden jeweils mit der finalen zwei Achsen- und der drei Achsenpolicy für das Szenario aus Kapitel 6.2.1 durchgeführt. Insgesamt wurden für die zwei Achsenpolicy 12 und für die drei Achsenpolicy 10 Experimente durchgeführt.

In Abbildung 6.13 sind zwei Beispiele für die Ausführung einer simulierten Schlagbewegung auf dem realen System illustriert.

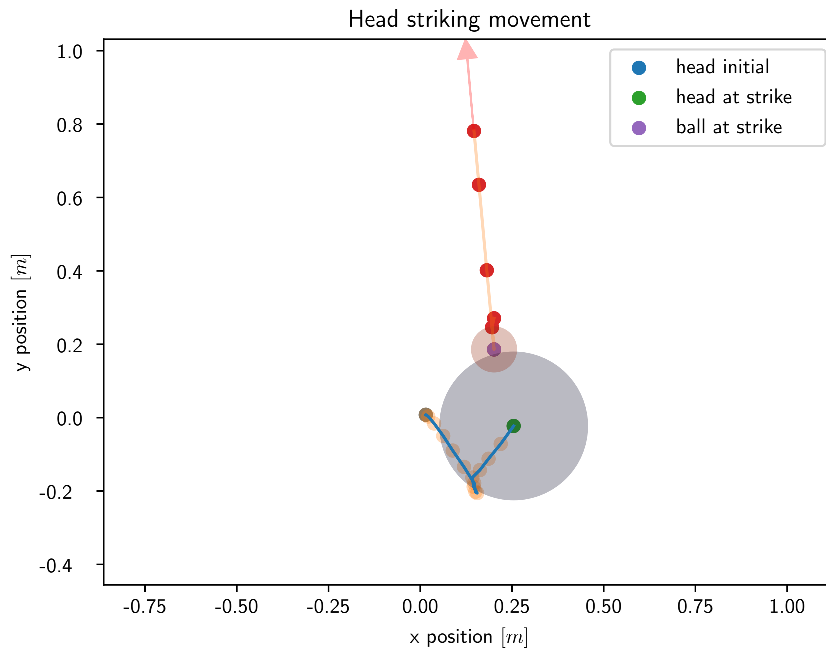


(a) Ausführung einer zwei Achsenschlagbewegung auf dem Roboter

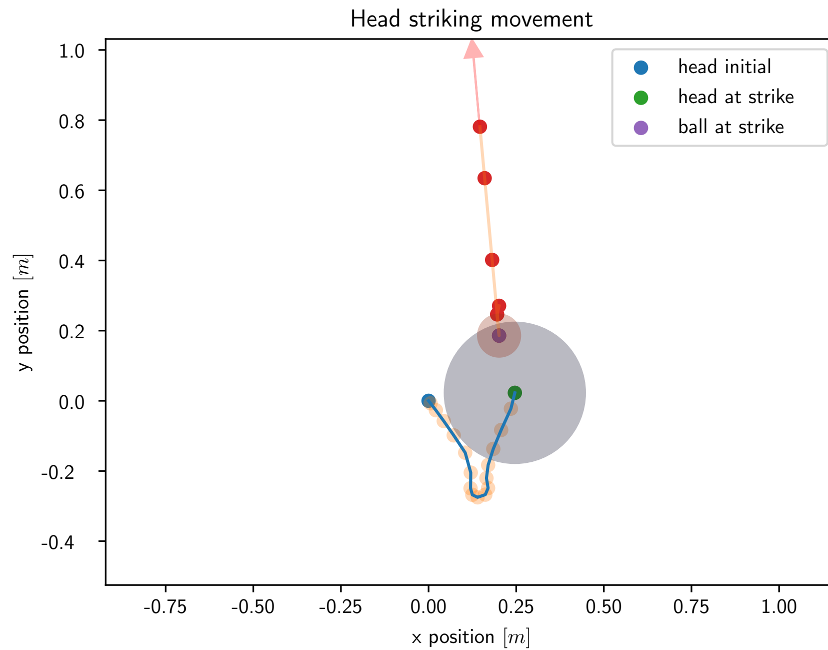


(b) Ausführung einer drei Achsenschlagbewegung auf dem Roboter

Abbildung 6.13: Beispiele von real ausgeführten Schlagbewegungen



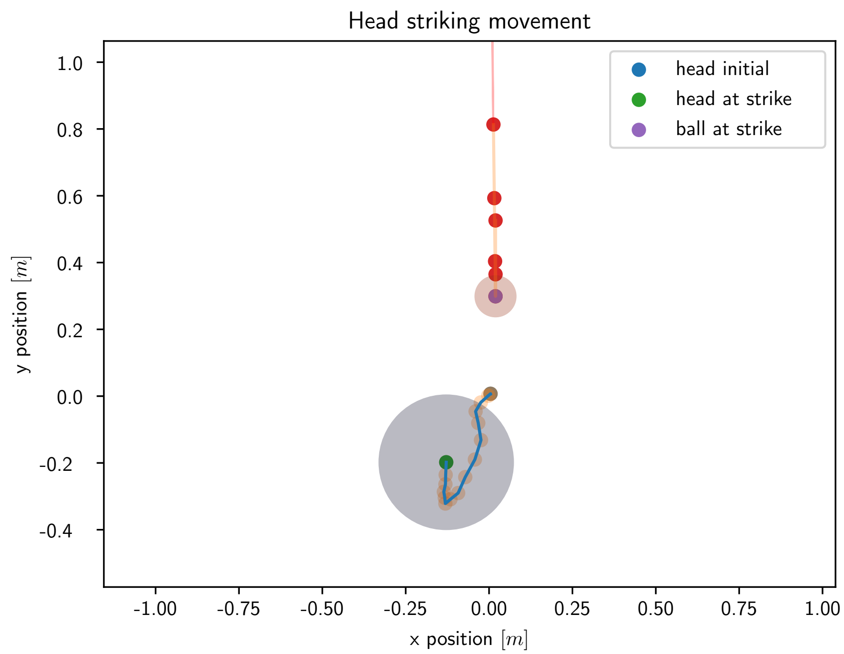
(a) Reale zwei Achsenschlagbewegung



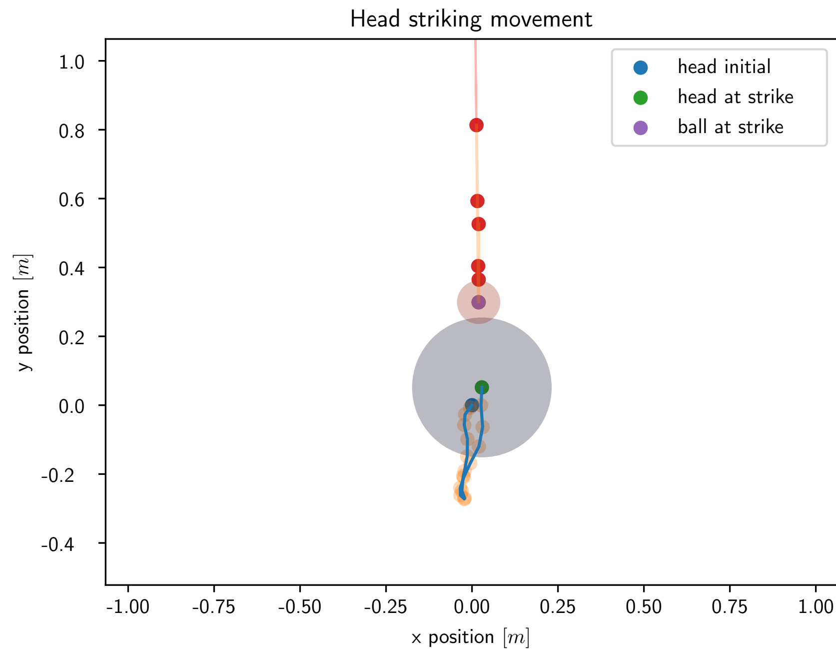
(b) Simulierte zwei Achsenschlagbewegung

Abbildung 6.14: Vergleich der realen und simulierten Schlagbewegung bei erfolgreicher Ausführung

Bei der Betrachtung der beiden Bewegungen ist festzustellen, dass hier kaum eine Abweichung vorliegt. Auf dem realen System ist die ausholende Bewegung nicht ganz so stark wie in der Simulation. Es ist dabei auch zu erkennen, dass das System länger an der Startposition bleibt bzw. nach den Daten des Gelenkwinkelschätzers hier verweilt. Dieses Verhalten kann durch die Latenz im Prozess des Schätzers erklärt werden. Dieser berechnet die aktuelle Gelenkposition durch Fusion der Daten der Motorencoder und der am Roboter vorhanden *IMUs*. Dabei ist die Abtastrate der Motorencoder höher als bei den *IMUs*, was zu einer gewissen Verzögerung in der Schätzung der aktuellen Gelenkpositionen und -geschwindigkeiten führen kann. Zusätzlich kann dieses Verhalten auch auf eine unzureichende Robustheit gegen die Verzögerung im Antriebsstrang hinweisen, was besonders bei weit ausholenden Schlagbewegungen zu Problemen führen kann.



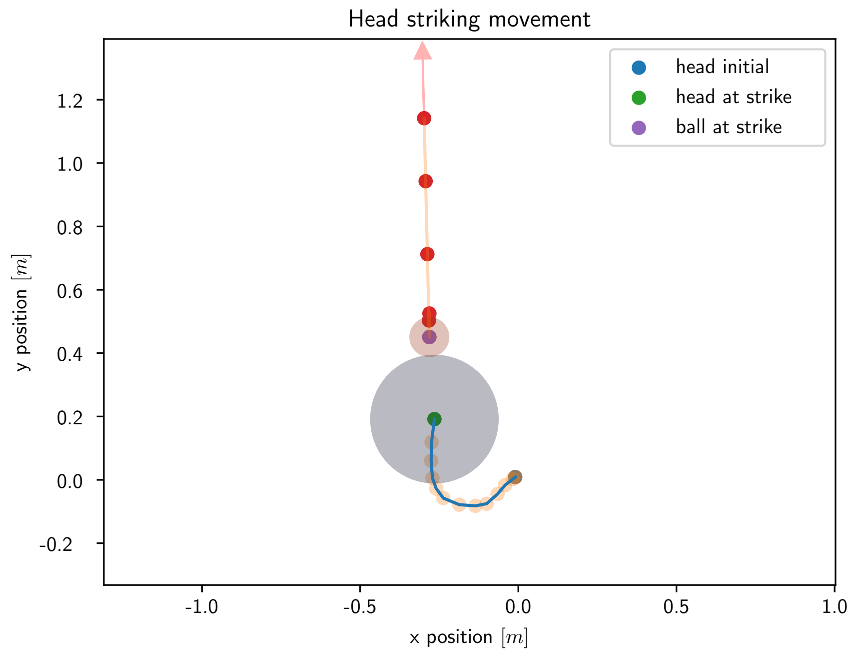
(a) Reale zwei Achsenschlagbewegung



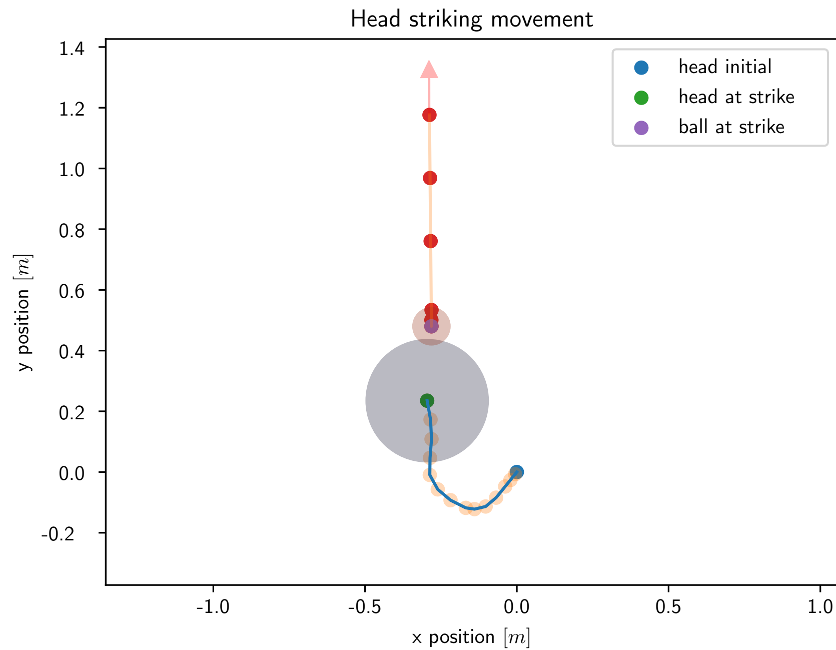
(b) Simulierte zwei Achsenschlagbewegung

Abbildung 6.15: Vergleich der realen und simulierten Schlagbewegung bei nicht erfolgreicher Ausführung

In dieser Bewegung wurde eine mittige Schlagbewegung ausgeführt. Hierbei ist zu erkennen, dass die Bewegung auf dem realen System nicht optimal ausgeführt wurde, da der Roboter zu lange an der Startpose verweilt. Es davon auszugehen, dass das System hier den geworfenen Ball verpasst bzw. nicht mit der nötigen Geschwindigkeit zurückgespielt hätte.



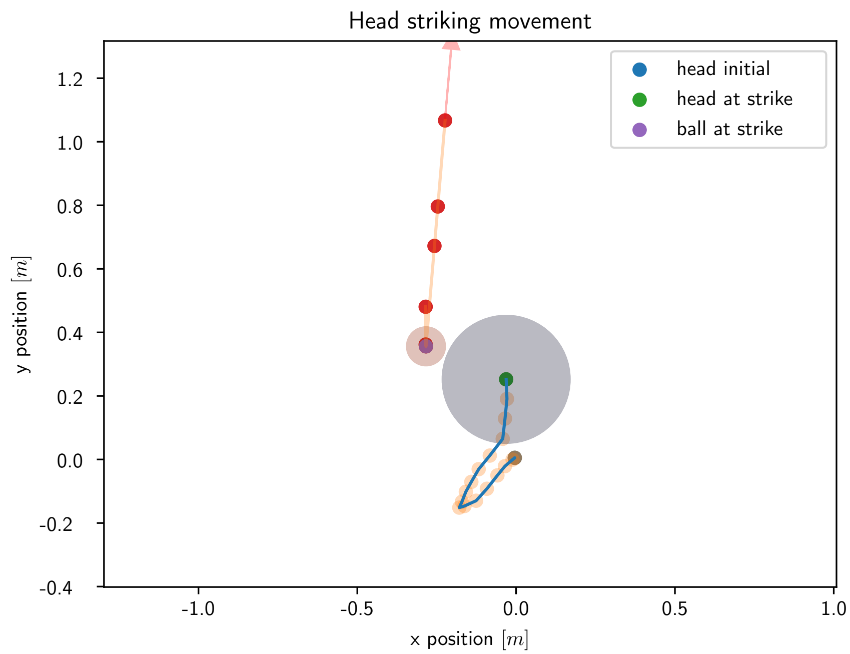
(a) Reale drei Achsenschlagbewegung



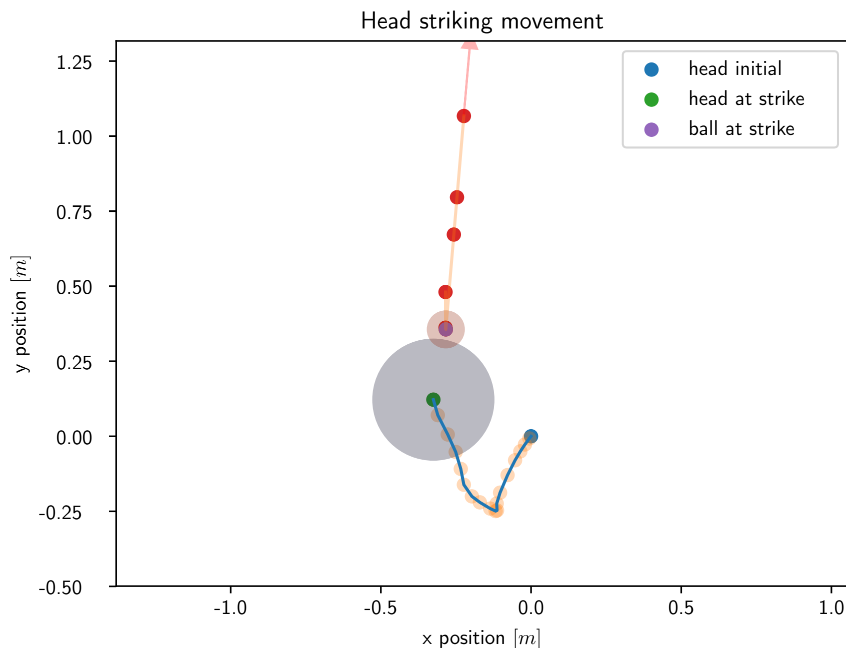
(b) Simulierte drei Achsenschlagbewegung

Abbildung 6.16: Vergleich der realen und simulierten Schlagbewegung bei erfolgreicher Ausführung

In dieser ausholenden Bewegung nach links zeigt sich eine gute Übertragung des in Simulation gelernten Verhaltens. Das reale Bewegungsmuster ist sehr ähnlich zu dem simulierten, wenngleich die Bewegung nicht so weit ausholt wie in der Simulation. So ist es hier sehr wahrscheinlich, dass der Ball sicher gespielt wurde.



(a) Reale drei Achsenschlagbewegung



(b) Simulierte drei Achsenschlagbewegung

Abbildung 6.17: Vergleich der realen und simulierten Schlagbewegung bei nicht erfolgreicher Ausführung

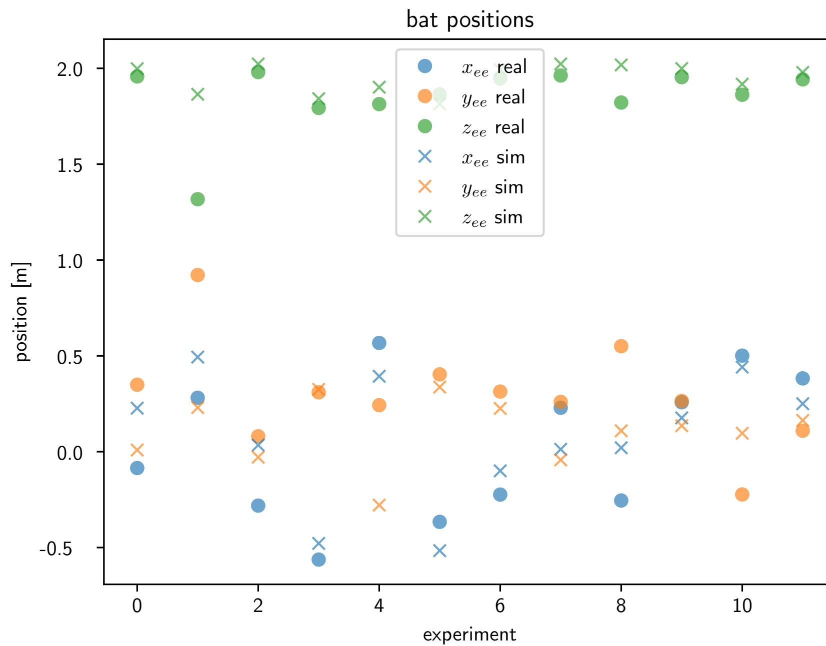
In dieser scharfen Bewegung zeigt sich, dass das reale System den Ball deutlich verfehlt. Hier ist eine deutliche Abweichung in Richtung der Startposition zu erkennen. Die Ausholbewegung an sich ist relativ ähnlich zur Simulation. Hingegen ist die Abweichung bei der eigentlichen Schlagbewegung zu erkennen.

6.5.2 Untersuchung der Abweichung zur Simulation im Schlag

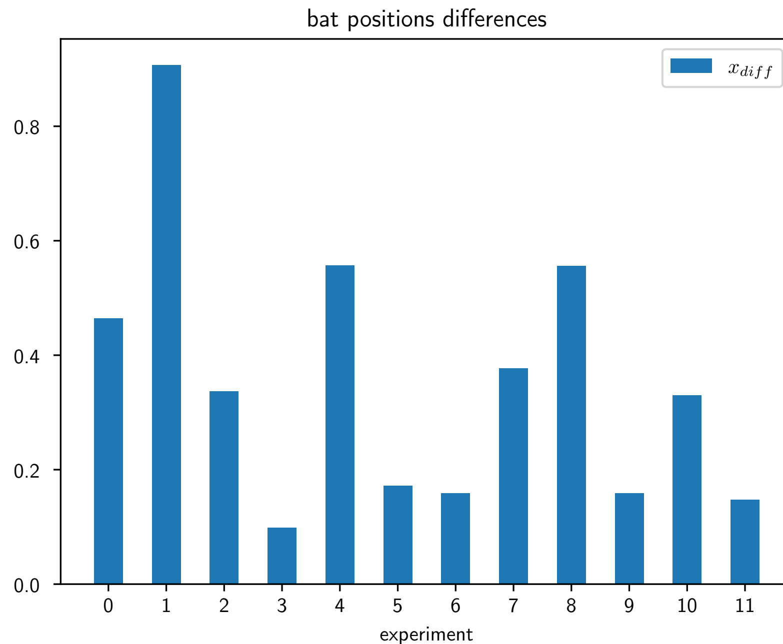
Im Nachfolgenden werden die einzelnen Experimente auf die Abweichung zur Simulation hin untersucht. Dafür werden jeweils die Positionen des Schlägers, der Gelenkpositionen und Gelenkgeschwindigkeiten sowie deren grafisch dargestellt. Anhand dieser Plots kann nun untersucht werden wie gut die jeweilige Bewegung in die Realität übertragen werden konnte. Dabei ist insbesondere die Abweichung des Schlägers interessant, da hieran festgestellt werden kann, ob die Position überhaupt erreicht wurde. Anhand der Abweichung der jeweiligen Gelenkgeschwindigkeiten kann ermittelt werden, ob eine ausreichend schnelle Schlagbewegung ausgeführt wurde, die den Ball in Richtung des Ziels schlägt. Als Maß für die Abweichung wird die euklidische Distanz zwischen jeweiligen Werten genutzt.

6.5.2.1 Experimente mit der zwei Achsenpolicy

Zunächst werden die Ergebnisse der Experimente mit der zwei Achsenpolicy untersucht. Insgesamt wurden mit dieser Policy 12 Experimente auf dem Roboter durchgeführt.



(a) Positionen des Schlägers in Simulation und Realität

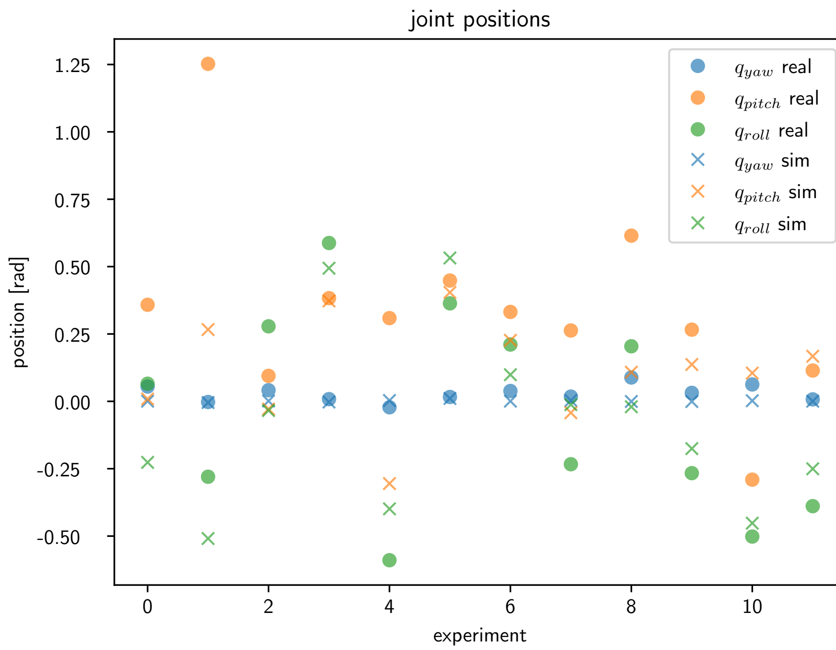


(b) Abweichungen der Schlägerpositon

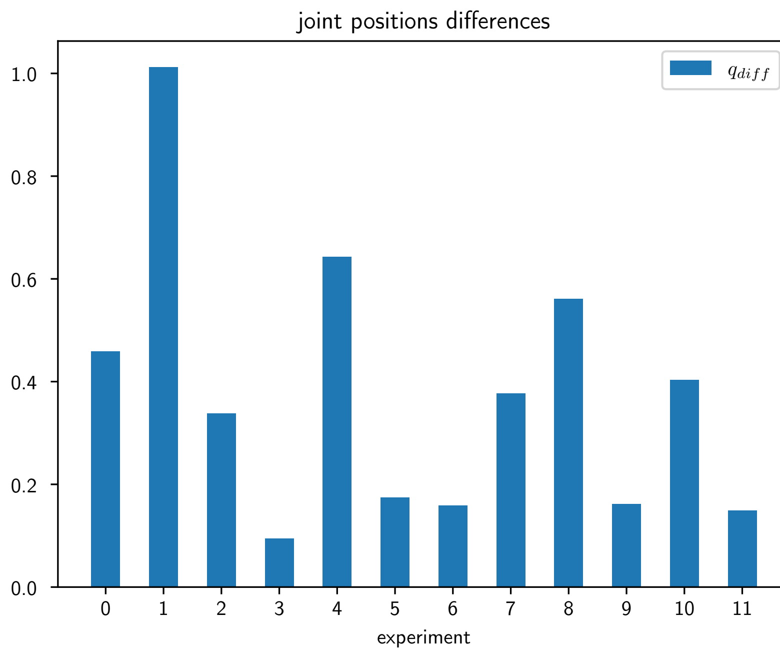
Abbildung 6.18: Schlägerpositionen in Simulation und Realität sowie deren Abweichungen mit der zwei Achsenpolicy

Anhand der Abweichungen in Abbildung 6.18b ist festzustellen, dass die Policy in 5 von 12 Experimenten die Position relativ präzise mit einer Abweichung von $\leq 0{,}2\,\mathrm{m}$ erreicht. Bei einer Abweichung von $0{,}2\,\mathrm{m} > x_{diff} \leq 0{,}4\,\mathrm{m}$ besteht aufgrund des Radius des Kopfes von 0,2 m noch die Chance, dass der Roboter auf Basis des Geschwindkigkeitsvektoren von Ball und Gelenken diesen wenigstens noch schlägt und nicht verfehlt, da die Schlägerposition den Mittelpunkt des Kopfes im Raum darstellt. Dies ist hier in 3 von 12 Experimenten der Fall. In den restlichen Fällen zeigt sich eine zu große Abweichung, um den Ball noch sinnvoll zu spielen.

Da die Position des Schlägers aus den Gelenkwinkeln mittels Vorwärtskinematik berechnet wird, zeigt sich in den Abweichungen in 6.19b kein Unterschied zu den Abweichungen des Schlägers. Interessanter ist hier die Betrachtung der einzelnen Komponenten in Abbildung 6.19. Hier zeigt sich, dass die größten Unterschiede zwischen simulierten und realen Gelenkpositionen zumeist in der Pitch-Achse liegen. Diese Achse hat im System den größten Bewegungsraum und die Abweichungen hierbei deuten darauf hin, dass sich das simulierte Verhalten deutlich von dem realen Verhalten unterscheidet.

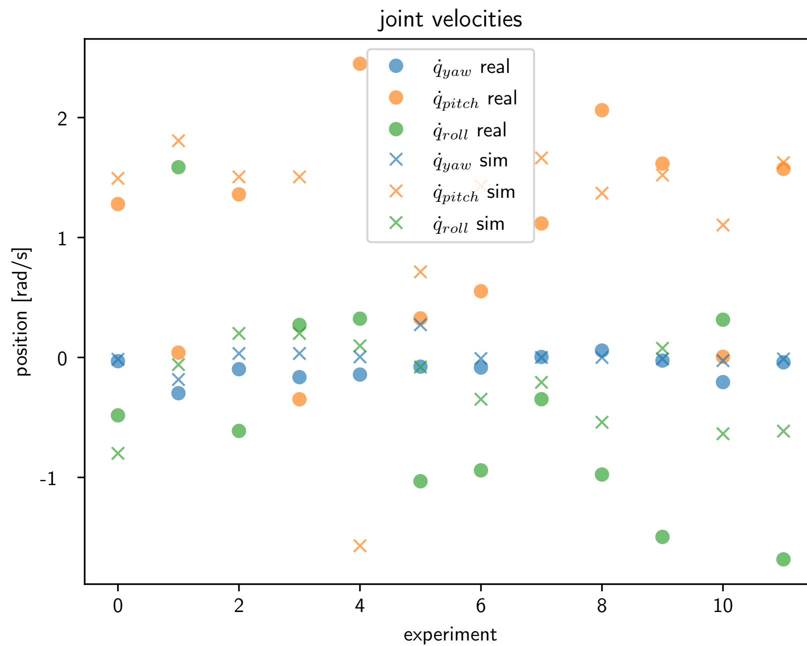


(a) Positionen der Gelenke in Simulation und Realität

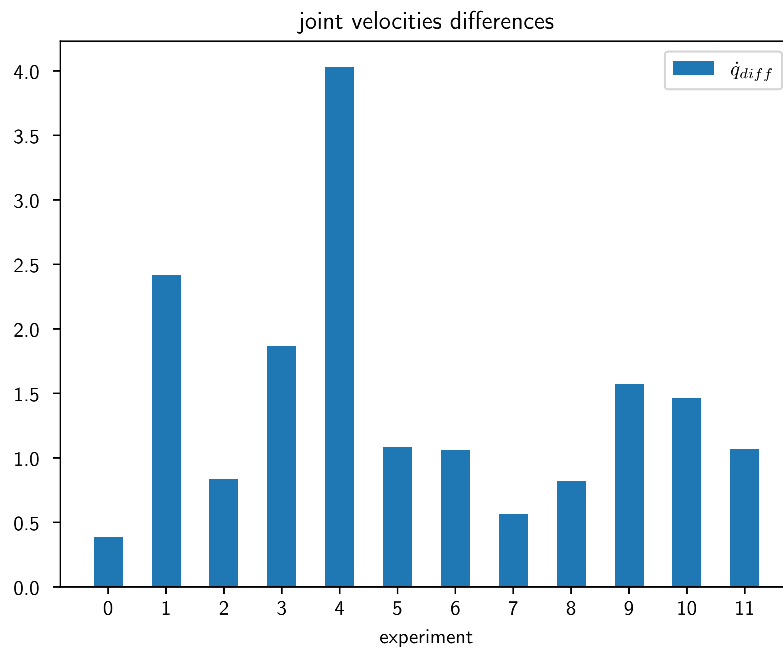


(b) Abweichungen der Gelenkpositionen

Abbildung 6.19: Gelenkpositionen in Simulation und Realität sowie deren Abweichungen mit der zwei Achsenpolicy



(a) Geschwindigkeiten der Gelenke in Simulation und Realität



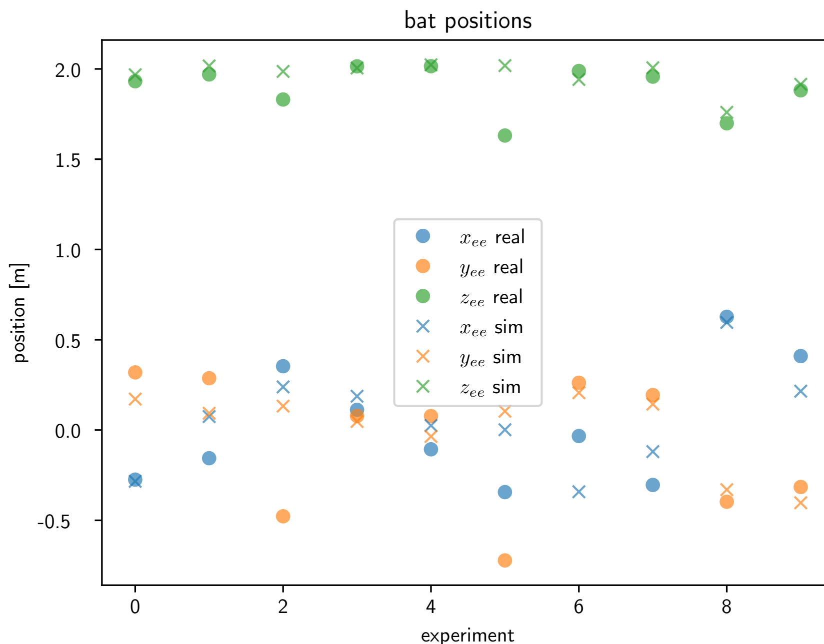
(b) Abweichungen der Gelenkgeschwindigkeiten

Abbildung 6.20: Gelenkgeschwidigkeiten in Simulation und Realität sowie deren Abweichungen.

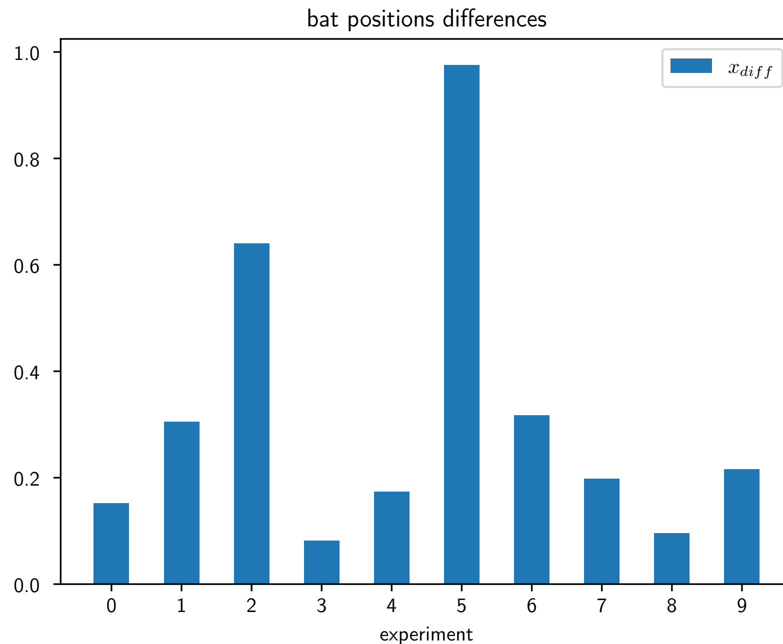
Bei den Gelenkgeschwindigkeiten zeigen sich vergleichsweise hohe Abweichungen zwischen den realen und den simulierten Daten. Dies kann in einigen Fällen zur Folge haben, dass der Agent den Ball nicht genau so spielt wie in der Simulation. Dabei ist es insbesondere problematisch, wenn die reale Geschwindigkeit deutlich unter der Simulation liegt. In diesen Fällen kann der Schläger den Zielpunkt nicht mehr mit einer ausreichenden Geschwindigkeit erreichen, um den Ball weit genug zu schlagen. Deutliche Abweichungen sind besonders bei Experimenten zu erkennen, in denen schon die Schlagposition stark abweicht. Aber auch in den Szenarien, in denen der Schläger eine geringe Abweichung zur simulierten Position hat, sind deutliche Unterschiede in der Geschwindigkeit zu erkennen. In der Richtung der Abweichung ist hingegen kein eindeutiger Trend zu einer hohen oder zu niedrigen Geschwindigkeit zu erkennen.

6.5.2.2 Experimente mit der drei Achsenpolicy

Mit der drei Achsenpolicy wurden insgesamt 10 Experimente auf dem Roboter durchgeführt.



(a) Positionen des Schlägers in Simulation und Realität

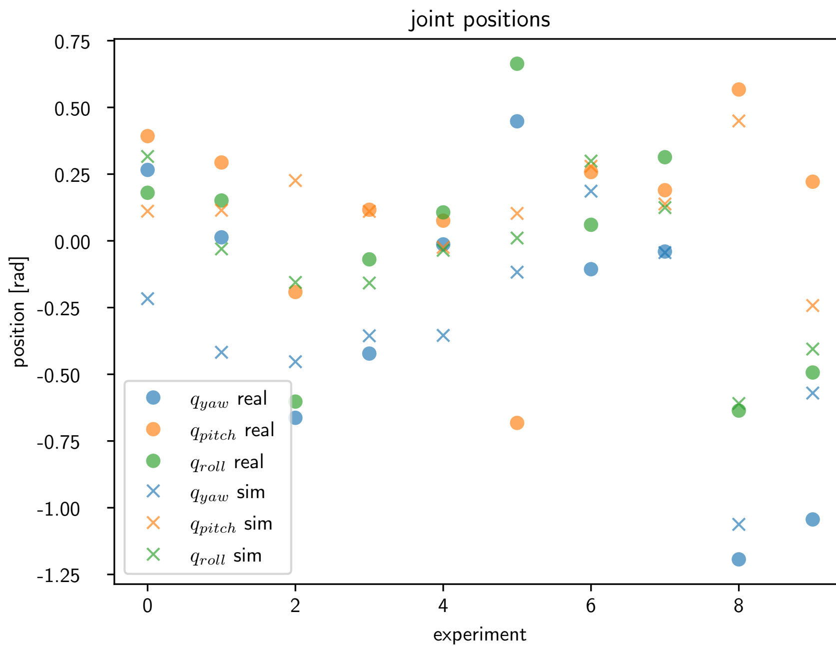


(b) Abweichungen der Schlägerpositon

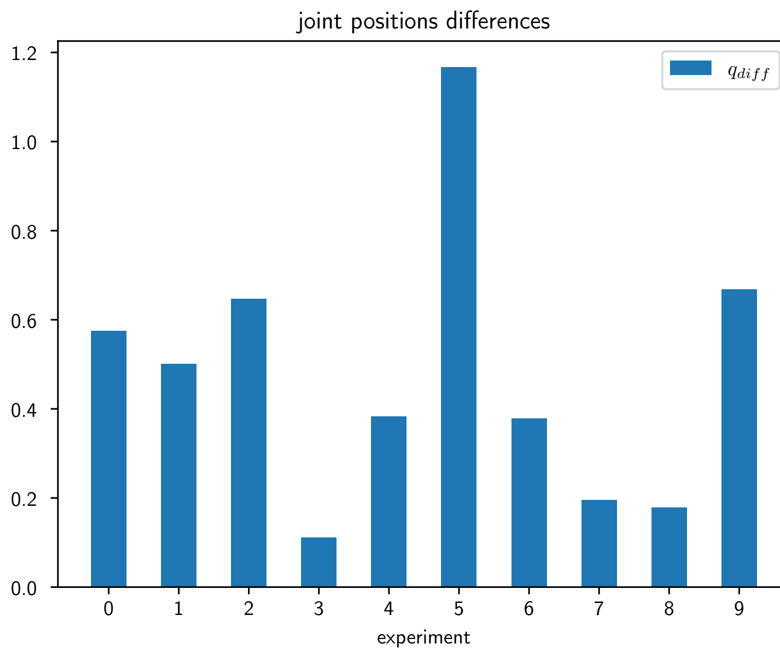
Abbildung 6.21: Schlägerpositionen in Simulation und Realität sowie deren Abweichungen mit der drei Achsenpolicy

Anhand der Abweichungen in Abbildung 6.21b ist festzustellen, dass die Policy in 5 von 10 Experimenten die Position relativ präzise mit einer Abweichung von $\leq 0{,}2\,\mathrm{m}$ erreicht. Betrachtet man die Abweichung von $0{,}2\,\mathrm{m} > x_{diff} \leq 0{,}4\,\mathrm{m}$, besteht hier in 2 von 10 Experimenten die Möglichkeit den Ball wenigstes zu treffen. Im Rest der Fälle ist es aufgrund der sehr hohen Abweichungen sicher, dass der Ball verfehlt wurde.

Bei den Abweichungen in den Gelenkpositionen zeigt sich hier ein leicht anderes Bild, da auch die Yaw-Achse bewegt wird. Diese zeigt hier vergleichsweise hohe Abweichungen in der Bewegung (vgl. Abbildung 6.22b). Dies kann möglicherweise mit Umbauarbeiten am System erklärt werden, wodurch die in [SWF16] identifizierten Dynamikparameter nicht mehr genau genug waren. Auch scheint hier die Randomisierung der selbigen nicht ausgereicht zu haben, um die vom Agenten gelernte Policy entsprechend robust zu gestalten. Da sich in der Abweichung der Gelenkwinkel zu den Schlägerpositionen jedoch kein konsistentes Bild zeigt, besteht hier auch die Möglichkeit, dass der Agent auf dem realen System die Redundanz der Achsen anders ausnutzt, als in der Simulation. Dies ist durch die erweiterten Bewegungsmöglichkeiten durch die Nutzung der Yaw-Achse möglich.

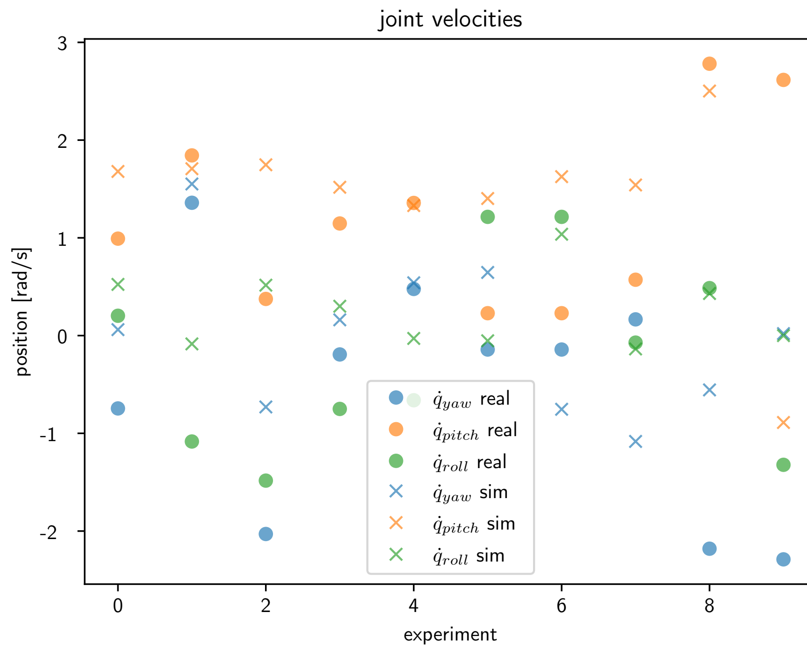


(a) Positionen der Gelenke in Simulation und Realität

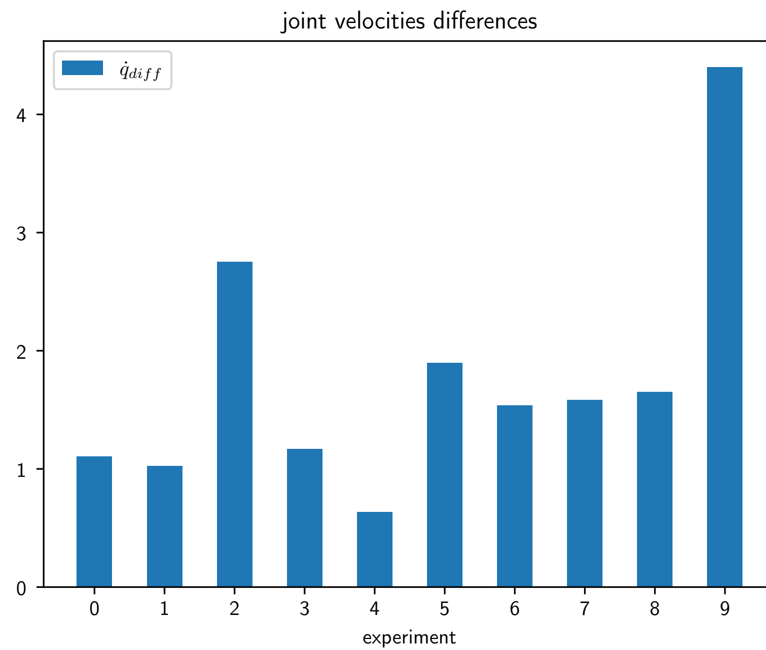


(b) Abweichungen der Gelenkpositionen

Abbildung 6.22: Gelenkpositionen in Simulation und Realität sowie deren Abweichungen mit der drei Achsenpolicy



(a) Geschwindigkeiten der Gelenke in Simulation und Realität



(b) Abweichungen der Gelenkgeschwindigkeiten

Abbildung 6.23: Gelenkgeschwindigkeiten in Simulation und Realität sowie deren Abweichungen mit der drei Achsenpolicy

Bei der Betrachtung der Gelenkgeschwindigkeiten zeigt sich im Vergleich zur zwei Achsenpolicy eine etwas höhere Abweichung, was auf die zusätzlich Bewegung der Yaw-Achse zurückzuführen ist. Bei der Betrachtung der einzelnen Abweichungen in Abbildung 6.23a zeigt sich deutlich, dass die Yaw-Achse häufig recht hoch abweicht, was zu einer insgesamt ungenaueren Bewegung führt.

6.6 Fazit

Zusammenfassend lässt sich feststellen, dass die in der Simulation gelernten Policies in der Lage sind, in einem Ballspielszenario mit festem Ziel den Ball recht sicher zurückzuspielen. Dabei hat sich in der Untersuchung gezeigt, dass die Nutzung der Yaw-Achse kaum positive Auswirkungen auf die Verhalten hat. Insgesamt haben sich in beiden Szenarien die zweiachsigen Verhalten als robuster herausgestellt. Das Verlagern der Startposition hat nur im Szenario von 4 m Zielentfernung eine leichte positive Auswirkung auf das Spielverhalten gehabt. Bei der Betrachtung der besten Policies ist das Bild ähnlich. Auch hier sind die auf zwei Achsen beschränkten Policies leicht überlegen. Dies ist ein überraschendes Ergebnis, da die Yaw-Achse die Möglichkeiten der Bewegung an sich vergrößert und der Agent mit den übrigen zwei Achsen tendenziell kürzere Strecken für einen Schlag zurücklegen muss, was die Limitierungen in der Beweglichkeit des Systems an sich überwinden sollte. Eine Ursache für das schlechtere Abschneiden der dreiachsigen Policies ist der größere Aktionsraum, was für den Agenten deutlich mehr Möglichkeiten der Bewegung bedeutet, von denen natürlich viele nicht optimal sind. Bei einem solchen Problem besteht immer die Gefahr, dass der Lernalgorithmus trotz Exploration in einem lokalen Minimum verweilt.
Bei der Betrachtung der Schlagbewegungen ist festzustellen, dass alle Policies die Tendenz zu einer ausholenden Schlagbewegung gezeigt haben. Daher ist davon auszugehen, dass die zusätzliche Zeitkomponente im Zustandsraum und in der Rewardfunktion den Agenten dazu ermutigt hat, sich nicht sofort zur vorhergesagten Schlagposition zu bewegen, sondern erst auszuholen und dann mit größerer Geschwindigkeit den Ball in Richtung des Ziels zu schlagen.
Bei der Übertragung in die Realität konnten knapp 50 % der Bewegungen erfolgreich umgesetzt werden. Das größte Problem hierbei scheint die Latenz des Schätzers und die Abweichungen bei der Bewegung der Pitch-Achse zu sein. Die Latenzen im Schätzer sorgt in den Szenarien, in denen eine mittige Schlagbewegung ausgeführt werden soll, wahrscheinlich dafür, dass Roboter zu viel Geschwindigkeit auf die Gelenke gibt, was zur Folge hat, dass er zu lange ausholt und dann aufgrund der Reibung in den Gelenken nicht schnell genug abbremsen kann, um noch rechtzeitig die Schlagposition zu erreichen.

7 Zusammenfassung

Zunächst wurde eine möglichst realitätsnahe Simulationsumgebung erstellt, die durch die Nutzung von *ROS*-Komponenten dem realen Systembetrieb ähnelt. Die Simulation bildet so gut wie möglich das reale System und seine Kommunikation zwischen den einzelnen Kompetenten ab. Zusätzlich wurde eine Lernumgebung erstellt die mit beliebigen Algorithmen interagieren kann und dabei die Schnittstelle zwischen Simulation und Lernalgorithmus bildet.

Beim Entwurf der Rewardfunktion hat sich herausgestellt, dass ein ausschließlich zeitbewusster Agent ohne Feedback im Reward zur Zeit nicht in der Lage ist eine Schlagbewegung zu erlernen. Hat der Agent hingegen Wissen über die relative Zeit zum Schlagzeitpunkt und bekommt Feedback mit der Zeitkomponente in der Rewardfunktion, ist der Agent in der Lage kraftvolle Schlagbewegungen zu erlernen. Dies ist eine besonders für zeitkritische Probleme wertvolle Erkenntnis.

Die Ergebnisse des Trainings in der Simulation zeigen, dass aktuelle *Deep Reinforcement Learning* Algorithmen ohne Vorwissen in der Lage sind, das Spielen eines Balls zu einem definierten Ziel zu erlernen. Bisherige Forschungen haben hierfür zumeist Imitationslernen mit anschließendem *Reinforcement Learning* genutzt (vgl. [Mül+13]). Dafür wurden drei Policies mit jeweils unterschiedlichen Bewegungsmöglichkeiten trainiert und in einem Testszenario auf ihr Leistung verglichen. Die Präzision des Zurückspielens wird im Falle des Roboters *Doggy* durch die Nutzung der redundanten Hüftachse nicht maßgeblich verbessert. Die Trefferaten der finalen Policies sind dabei nicht für ein gutes Ballspiel geeignet. Werden hingegen die manuell identifizierten besten Policies der einzelnen Szenarien genutzt, ist ein gutes Spiel mit dem System möglich.

Die Untersuchung der Übertragbarkeit in die Realität hat ergeben, dass das eingesetzte Verfahren der Dynamikrandomisierung die in Simulation gelernten Policies mit einer ordentlichen Robustheit gegen nicht modellierbare Phänomene im realen Betrieb versehen hat. So konnten mindestens die Hälfte der Schlagbewegungen mit simulierten Balltracks auch auf dem realen System mit einer guten Präzision ausgeführt werden. Die übertragenen Bewegungen liegen bei guter Ausführung sehr nah an den simulierten Schlagbewegungen. Bei nicht erfolgreicher Übertragung in die Realität handelt es sich meist um weit ausholende Schlagbewegungen, bei denen eine Latenz in der Ansteuerung der Gelenke beobachtet werden konnte. Dies führt dazu, dass der Agent in diesen Fällen zu lange ausholt und sich nicht mehr schnell genug zur Schlagposition bewegen kann.

7.1 Ausblick

Ein noch zu untersuchender Aspekt des Systems ist die Robustheit der gelernten Policies in Kombination mit dem realen Trackingsystem. Dafür muss dieses zunächst auf die Nutzung der Hüftachse erweitert werden. Anschließend kann hier vergleichend untersucht werden, wie gut sich die zwei Achsenpolicy im Vergleich zur drei Achsenlpolicy mit echten Balltracks schlägt. Dafür muss auch untersucht werden, wie das Übertragen von weitausholenden Bewegungen verbessert werden kann, da diese Bewegungen in den Experimenten zumeist schlechte Ergebnisse erzielt haben.

Ein weiteres zu untersuchendes Phänomen ist das Underfitting während des Trainingsporzesses. Zumal die Probleme hier in den Testszenarien nicht konsistent waren. Hierbei war im Verlauf der Arbeit nicht endgültig zu klären, ob dies mit nicht optimalen Hyperparametern für den verwendeten Lernalgorithmus oder der verwendeteten Netzstruktur zusammenhängt. Auch ein Zusammenhang mit den Parametern des verwendeten Explorationsverfahrens kann hier nicht abschließend ausgeschlossen werden. Eine tiefergehende Untersuchung sollte eine erschöpfende Gridsuche mit einer definierten Menge und Variation von Hyperparamtern auf verschiedenen Netzstrukturen enthalten.

Weiterhin ist eine Untersuchung der aktuellen Fortschritte in Bezug auf *Deep Reinforcement Learning* Algorithmen wie *Proximal Policy Optimization (PPO)* [Sch+17] oder *Soft Actor-Critic (SAC)* [Haa+18] in Betracht zu ziehen. Diese haben insbesondere sehr gute Fortschritte in Bezug auf die effiziente Nutzung der Beispiele gemacht, was im momentanen Aufbau die meiste Zeit des Trainings in Anspruch nimmt.

Glossar

Deep Reinforcement Learning Eine Unterkategorie des Reinforcement Learnings, welches mittels tiefer, mehrschichtiger, künstlicher neuronaler Netze z.B. die Q-Funktion lernt oder die Repräsentation der Policy ist ein neuronales Netz. 2, 3, 10, 28, 81, 82

Gazebo Ein quelloffener Multi-Roboter-Simulator mit akkurater Physiksimulation und einer nahtlosen Integration in das Robot Operating System. 22, 32, 35, 37

Machine Learning Klasse von Verfahren aus dem Bereich der Künstlichen Intelligenz, welche Vorhersagen anhand von gelernten Beispielen treffen. Dabei erkennen diese Algorithmen Muster in den Beispielen und verallgemeinern diese. 3

Markov Eigenschaft Die Markov Eigenschaft gibt bei einem stochastischen Prozess, dass alle Folgezustände nur vom aktuellen Zustand abhängen. Ein solcher Prozess wird auch als Markov Prozess bezeichnet. 6

Reinforcement Learning zu deutsch „bestärkendes Lernen“, eine Unterdisziplin des Machine Learnings, in der ein Agent auf Basis des Umgebungszustands ein Verhalten erlernt, mit dem eine gestellte Aufgabe gelöst werden kann. 2, 3, 5, 36, 81

Akronyme

BPTT Backpropagation Through Time. 12, 13

CNN Convolutional Neural Network. 9, 17

DDPG Deep Deterministic Policy Gradient. 11, 12, 19, 23, 27, 29, 41

DQN Deep Q-Networks. 10, 11

GAN Generative Adversial Network. 9, 17

HER Hindsight Experience Replay. 16, 19

IMU Inertial Measurement Unit. 14, 32, 64

LSTM Long-Short-Term Memory. 9, 12, 16

MDP Markov Decision Process. 4

MHT Multiple Hypothesis Tracker. 21, 31, 61

MSE Mean Squared Error. 10

ODE Open Dynamics Engine. 22

POMDP Partially Observable Markov Decision Process. 12

PPO Proximal Policy Optimization. 82

RDPG Recurrent Deterministic Policy Gradient. 12, 16, 23, 41

ReLU Rectified Linear Unit. 9

ROS Robot Operating System. 31, 32, 34, 37, 81

SAC Soft Actor-Critic. 82

TD Temporal Difference. 6, 7

Literatur

[And+17] Andrychowicz, M. et al. Hindsight experience replay. In: *Advances in Neural Information Processing Systems*. 2017, 5048–5058.

[Aru+17] Arulkumaran, K. et al. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866* (2017).

[Bel52] Bellman, R. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences* 38, 8 (1952), 716–719.

[BF09] Birbach, O. und Frese, U. A Multiple Hypothesis Approach for a Ball Tracking System. In: *Computer Vision Systems*. Hrsg. von J. Piater B. Schiele, M. F. Bd. 5815. Lecture Notes in Computer Science. Springer, 2009, 435–444.

[BKC17] Badrinarayanan, V., Kendall, A. und Cipolla, R. SegNet: A Deep Convolutional Encoder-Decoder Architecture for Image Segmentation. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 39, 12 (Dez. 2017), 2481–2495. issn: 0162-8828. doi: `10.1109/TPAMI.2016.2644615`.

[BKH16] Ba, J. L., Kiros, J. R. und Hinton, G. E. Layer normalization. *arXiv preprint arXiv:1607.06450* (2016).

[Bou+17] Bousmalis, K. et al. Using simulation and domain adaptation to improve efficiency of deep robotic grasping. *arXiv preprint arXiv:1709.07857* (2017).

[Bro+16] Brockman, G. et al. *OpenAI Gym*. 2016. eprint: `arXiv:1606.01540`.

[Cha+15] Chang, A. X. et al. Shapenet: An information-rich 3d model repository. *arXiv preprint arXiv:1512.03012* (2015).

[Dha+17] Dhariwal, P. et al. *OpenAI Baselines*. `https://github.com/openai/baselines`. 2017.

[DHS11] Duchi, J., Hazan, E. und Singer, Y. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research* 12, Jul (2011), 2121–2159.

[Goo+14] Goodfellow, I. et al. Generative adversarial nets. In: *Advances in neural information processing systems*. 2014, 2672–2680.

[Haa+18] Haarnoja, T. et al. Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor. *arXiv preprint*

arXiv:1801.01290 (2018).

[Hee+15] HEESS, N. et al. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455* (2015).

[HS97] HOCHREITER, S. und SCHMIDHUBER, J. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[KB14] KINGMA, D. P. und BA, J. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[KBP13] KOBER, J., BAGNELL, J. A. und PETERS, J. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* 32, 11 (2013), 1238–1274.

[KH04] KOENIG, N. und HOWARD, A. Design and Use Paradigms for Gazebo, An Open-Source Multi-Robot Simulator. In: *In IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2004, 2149–2154.

[KSH12] KRIZHEVSKY, A., SUTSKEVER, I. und HINTON, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: *Advances in Neural Information Processing Systems 25*. Hrsg. von PEREIRA, F. et al. Curran Associates, Inc., 2012, 1097–1105.

[Lil+15] LILLICRAP, T. P. et al. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[Mni+13] MNIH, V. et al. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[Mni+15] MNIH, V. et al. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[MP43] MCCULLOCH, W. S. und PITTS, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics* 5, 4 (1943), 115–133.

[Mül+13] MÜLLING, K. et al. Learning to select and generalize striking movements in robot table tennis. *The International Journal of Robotics Research* 32, 3 (2013), 263–279.

[Pen+17] PENG, X. B. et al. Sim-to-real transfer of robotic control with dynamics randomization. *arXiv preprint arXiv:1710.06537* (2017).

[Pin+17] PINTO, L. et al. Asymmetric Actor Critic for Image-Based Robot Learning. *arXiv preprint arXiv:1710.06542* (2017).

[Pla17] PLAPPERT, M. Parameter Space Noise for Exploration in Deep Reinforcement Learning. Masterarbeit. Karlsruhe Institute of Technology, 2017.

[RHW86] RUMELHART, D. E., HINTON, G. E. und WILLIAMS, R. J. Learning

representations by back-propagating errors. *nature* 323, 6088 (1986), 533.

[Ros58] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.

[SB98] Sutton, R. S. und Barto, A. G. *Reinforcement learning: An introduction.* Adaptive computation and machine learning. MIT Press, 1998.

[Sch+17] Schulman, J. et al. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017). arXiv: `1707.06347`. url: `http://arxiv.org/abs/1707.06347`.

[Sch17] Schüthe, D. Dynamic Bat-Control and Human-Robot-Interaction of a redundant Ball Playing Robot. Diss. Universität Bremen, 2017.

[Sil+14] Silver, D. et al. Deterministic policy gradient algorithms. In: *ICML*. 2014.

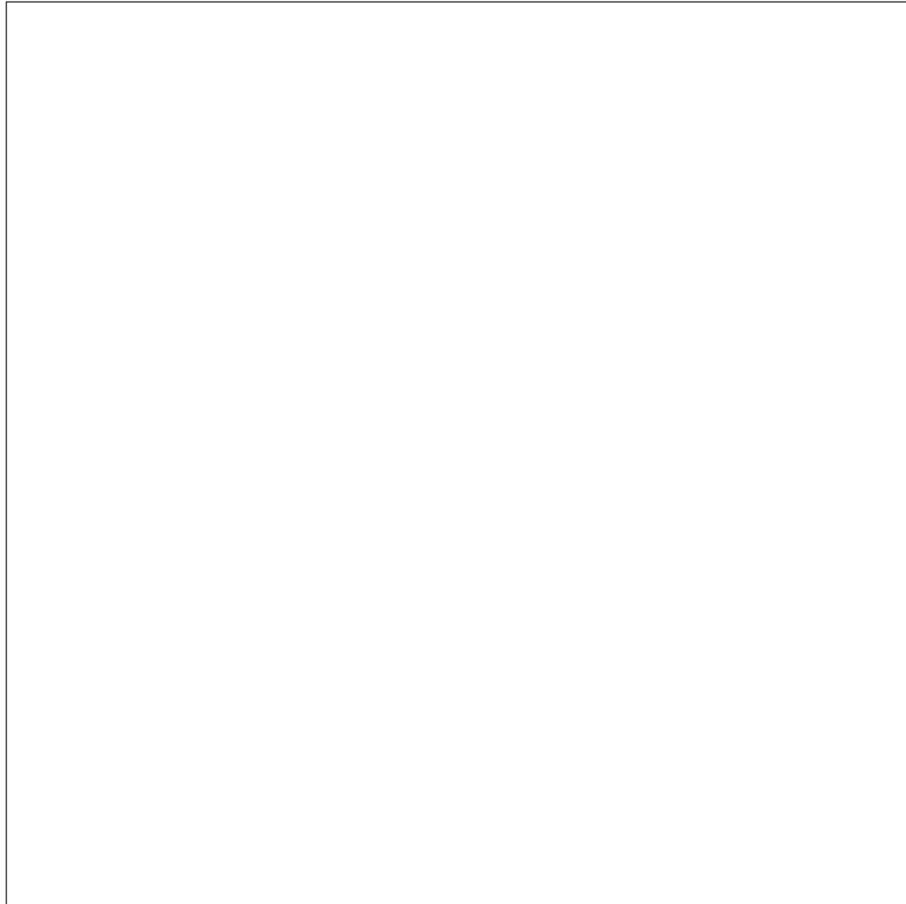
[SWF16] Schüthe, D., Wenk, F. und Frese, U. Dynamics Calibration of a Redundant Flexible Joint Robot based on Gyroscopes and Encoders. In: *ICINCO 2016 – 13th International Conference on Informatics in Control, Automation and Robotics*. SCITEPRESS, 2016.

[Tob+17] Tobin, J. et al. Domain randomization for transferring deep neural networks from simulation to the real world. In: *Intelligent Robots and Systems (IROS), 2017 IEEE/RSJ International Conference on*. IEEE. 2017, 23–30.

[UO30] Uhlenbeck, G. E. und Ornstein, L. S. On the theory of the Brownian motion. *Physical review* 36, 5 (1930), 823.

Anhang

CD



Auf der CD ist enthalten:

- Die Masterthesis als PDF
- Entwickelter Quelltext
- Logdaten der Experimente