Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Roman Jungblut

Entwicklung einer physikalischen Reinforcement Learning Umgebung für einen Quadruped-Roboter zum Lernen eines Bewegungsmusters mittels ROS und OpenAI Gym

Fakultät Technik und Informatik
Department Fahrzeugtechnik und Flugzeugbau

Faculty of Engineering and Computer ScienceDepartment of Automotive and Aeronautical Engineering

Roman Jungblut

Entwicklung einer physikalischen Reinforcement Learning Umgebung für einen Quadruped-Roboter zum Lernen eines Bewegungsmusters mittels ROS und OpenAI Gym

Bachelor-/Masterarbeit eingereicht im Rahmen der Bachelor-/Masterprüfung

im Studiengang Mechatronik
am Department Fahrzeugtechnik und Flugzeugbau
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Erstprüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitprüfer: Prof. Dr. Wolfgang Fohl

Abgabedatum: 11. April 2019

Zusammenfassung

Roman Jungblut

Thema der Bachelorthesis

Entwicklung einer physikalischen Reinforcement Learning Umgebung für einen Quadruped-Roboter zum Lernen eines Bewegungsmusters mittels ROS und OpenAI Gym

Stichworte

Reinforcement Learning, DQN, Neuronale Netze, ROS, Gazebo, OpenAI Gym

Kurzzusammenfassung

Das Ziel dieser Arbeit ist die Erstellung einer Reinforcement Learning Umgebung für einen Quadruped-Roboter. Der Roboter soll dabei innerhalb einer physikalisch simulierten Umgebung ein Bewegungsmuster zur Fortbewegung und Navigieren auf einen zufälligen Zielpunkt erlernen. Dazu wird mittels des Robotik-Frameworks ROS und der Simulationssoftware Gazebo ein Robotermodell innerhalb einer physikalischen Simulationsumgebung erstellt. Zum Erstellen der Reinforcement Learning Umgebung wird das Framework OpenAI Gym und das Zusatzpaket Openai_ros verwendet. Zum Erlernen des Bewegungsmusters wird das DQN-Lernalgorithmus verbunden mit Verbesserungsmethoden des Experience Replay Buffers und des Target Networks verwendet. Das erstellte DQN-Netzwerk wird anschließend trainiert und ausgewertet.

Title of the paper

Development of a physical reinforcement learning environment for a quadruped robot for learning a movement pattern using ROS and OpenAI Gym

Keywords

Reinforcement Learning, DQN, Neuronale Netze, ROS, Gazebo, OpenAI Gym

Abstract

The goal of this work is the creation of a reinforcement learning environment for a quadruped robot. Within a physically simulated environment, the robot's goal is to learn a movement pattern for navigating to a random target point. For this purpose, a robotic model is created within a physical simulation environment by means of the robotics framework called ROS and the simulation software Gazebo. The framework OpenAI Gym and the additional package Openai_ros are used to create the Reinforcement Learning environment. To learn the movement pattern, the DQN learning algorithm is used in conjunction with improvement methods of the Experience Replay Buffer and the Target Network. The created DQN network is then trained and evaluated.

Inhaltsverzeichnis

Abbildungsverzeichnis

Formelverzeichnis

1 Einleitung

Das Reinforcement Learning (RL) findet immer neue Anwendungsfelder in vielen Bereichen des Lebens. So wächst auch ihre Rolle bei zukünftigen Durchbrüchen in der Erstellung der künstlichen Intelligenz. Durch stetig weiterentwickelnde Hardware steigen auch die Möglichkeiten des RLs. Es wurden große Durchbrüche bei der Erstellung von Spiel-KIs erzielt. So wurden in den letzten Jahren die OpenAI Five erschaffe. Das ist eine KI, die sich in einem Computerspiel namens „Dota 2“ mit den besten Spielern der Welt messen kann. Im Bereich der Robotik findet das RL auch immer mehr Einsatz. So wird es bei dem Unternehmen Boston Dynamics eingesetzt, um innerhalb einer Simulationsumgebung einem humanoiden Roboter eine Laufbewegung beizubringen. So wird auch innerhalb dieser Arbeit versucht eine RL-Umgebung zu erstellen, in der ein Roboter verschiedene Aufgaben erlernen kann.

1.1 Motivation

Der Einsatz des RLs zum Erlernen diverser Aufgaben in virtuellen Umgebungen, wie der Spielkonsole „Atari“ oder bei diversen Computerspielen ist kein Einzelfall. Es existieren jedoch wenige Beispiele bei den eine physikalische Umgebung eingesetzt wird. Zudem gibt es auch wenige RL-Umgebungen, die für Robotikanwendungen konzipiert sind. Aufgrund dessen soll in Rahmen dieser Arbeit eine simple Robotikanwendung in Form eines Quadruped-Roboters innerhalb einer simulierten Welt realisiert werden. Zudem kann untersucht werden wie die physikalische Simulationsumgebung Gazebo und das Robotik-Framework ROS sich für diese Aufgabe eignen. Sollte dies zutreffen, wäre es möglich in einer Simulation trainierte Modelle anschließend auf reale Robotik-Anwendungen zu übertragen.

1.1 Aufgabenstellung

Im Rahmen dieser Bachelorthesis soll eine RL-Umgebung für einen Quadruped-Roboter entwickelt werden. Um die Umgebung zu testen, muss der Roboter eine Aufgabe lösen. Die Aufgabe ist dabei in einer physikalischen Simulationsumgebung ein Bewegungsmuster für eine Vorwärtsbewegung und eine Navigation zu einem beliebigen Zielpunkt zu erlernen.

Zu Beginn des Projektes soll das Robotermodell modelliert werden. Dabei wird der Kriechgang analysiert und anschließend umgewandelt, um das Training des Agenten möglichst zu vereinfachen. Anhand des in CAD angefertigten Roboters wird der Bewegungs- und Aktionsraum des Roboters bestimmt. Des Weiteren wird die physikalische Umgebung mittels ROS und Gazebo erstellt. Dabei wird der in CAD erstellter Quadruped-Roboter in ein für ROS und Gazebo passendes Format umgewandelt. Zum Ansteuern der Aktuatoren werden im Laufe der Arbeit dessen Regler sowie die physikalischen Eigenschaften der Bauteile, Gelenke und der Simulationsumgebung parametrisiert. Zum Erlernen des Bewegungsmusters wird das Deep Q-Network (DQN) verwendet, wobei der Algorithmus mit einigen Verbesserungstechniken ergänzt wird. Das Training des neuronalen Netzwerkes erfolgt durch mehrere Szenarien. Das erste Szenario deckt dabei die Fortbewegung und das zweite Szenario die Navigation ab. Das dritte Szenario wird zum endgültigen Auswerten des Modells verwendet.

1.2 Gliederung der Arbeit

Im ersten Teil der Arbeit werden die nötigen Grundlagen der verwendeten Techniken und Methoden erläutert. Dabei werden die Machine Learning Techniken RL und das Deep Learning erklärt sowie deren Zusammenhang beim Einsatz der DQN-Technik. Zudem wird das verwendete Framework ROS und die eingebaute Simulationssoftware Gazebo vorgestellt. Es wird der Aufbau des Frameworks und dessen wichtigsten Bausteine genannt und erklärt. Darüber hinaus werden die wichtigsten eingesetzten Pakete und Bibliotheken aufgezählt.

Im Kapitel 3 erfolgt die Umsetzung der Arbeit. Es werden zunächst die Konstruktionsanforderungen eines Quadruped-Roboters aufgezählt. Anschließend werden sein Aktionsbereich und die verwendete Bewegungsart beschrieben. Im nachfolgenden Kapitel 3.3 erfolgt die Erstellung der ROS-Projektpakete. Dabei wird die Erstellung des Modells, die Konvertierung des CAD-Robotermodells in ein Simulationsmodell und die Parametrisierung der Aktuatoren beschrieben. In Kapitel 3.4 wird ein weiteres ROS-Projektpaket zur Erstellung der RL-Umgebung erstellt. Dabei werden die wichtigsten Funktionen der Lernumgebung und der verwendeten Lernmethode DQN erläutert. Dabei wird das neuronale Netz aufgebaut und schrittweise nach Szenarien trainiert. Anschließend wird in Kapitel 4 das trainierte Modell in Betrieb genommen und ausgewertet.

Im letzten Kapitel der Arbeit werden alle durchgeführten Schritte zusammengefasst und die resultierenden Erkenntnisse in einem Fazit besprochen. Abschließend folgt ein Ausblick zum Erweitern und Verbessern dieser Arbeit.

2 Grundlagen

2.1 Reinforcement Learning

Reinforcement Learning ist ein Teilgebiet des Bereiches Machine Learning, welches sich mit der Entscheidungsfindung beschäftigt. Es zielt darauf ab, wie sich ein System innerhalb einer Umgebung verhält, um eine aktionsbezogene Belohnung zu maximieren. RL-Algorithmen untersuchen den Ablauf der ausgeführten Aktionen und entwickeln eine Strategie zum Erreichen eines Ziels. Nimmt man ein Computerspiel als Beispiel für eine Umgebung geht es bei RL darum, wie der Spieler Aktionen ausführen kann. Eine Aktion kann dabei eine Bewegung in eine bestimmte Richtung sein, die eine Belohnung oder Bestrafung nach sich zieht. Diese bedeuten in einem Spiel eine mögliche Anzahl an Punkten.

Im Rahmen des RLs gibt es einen Entscheidungsträger (Agent), der mit der Umgebung mittels seiner Aktionen interagiert (siehe Abb. 1). Bei jedem Zeitschritt t = 0, 1, 2, … erhält der Agent durch die ausgeführte Aktion $\mathbf{a_t \in A}$ die resultierende Darstellung des Zustandes $\mathbf{s_t \in S}$ der Umgebung sowie eine Belohnung $\mathbf{r_t \in R}$. Der Zustand beschreibt dabei den aktuellen Zustand des Agenten innerhalb der Umgebung. Anhand der neuen Darstellung des Zustandes wählt der Agent eine weitere Aktion aus. Die Umgebung wird dann in einen neuen Zustand s_{t+1} versetzt. Der Agent erhält wieder eine Belohnung r_{t+1} als Folge der vorherigen Aktion.

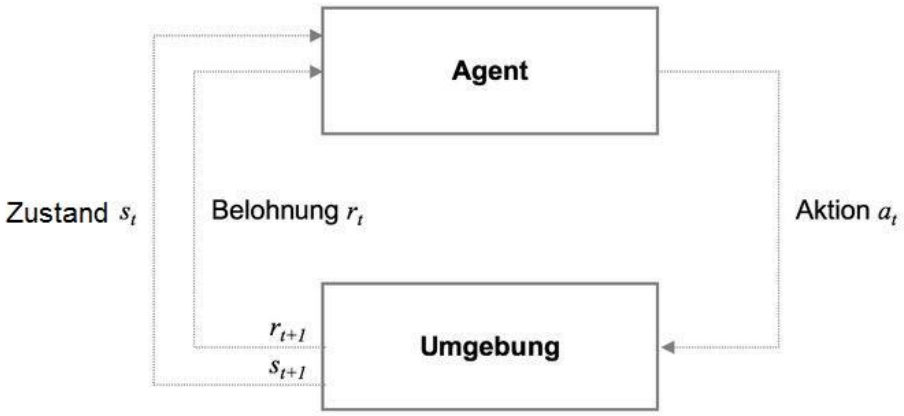

Abbildung 1 Konzept des RLs (Angepasst von: Barto (2017))

Die Interaktionen werden im Zeitverlauf nacheinander ausgeführt. Durch diesen kontinuierlichen Prozess entsteht eine Trajektorie. Diese beschreibt den sequenziellen Prozess der Auswahl einer Aktion aus einem Zustand, den Übergang zu einem neuen Zustand und dem Erhalt einer Belohnung.

$$s_0, a_0, r_1, s_1, a_1, r_2, s_2, a_2, r_3, \ldots$$

Das Ziel des Agenten wird durch ein Belohnungssystem definiert. Der Agent hat die Aufgabe den Gesamtbetrag der Belohnung zu maximieren. Die Belohnung ist ein Wert, welcher davon abhängig ist wie sich sein Zustand oder die Umgebung nach einer Aktion verändert hat. War die letzte Aktion besonders vorteilhaft zum Erreichen des Ziels, so kann die Belohnung besonders groß ausfallen. Das Wichtige dabei ist nicht

die sofortige Belohnung zu maximieren, sondern die auf lange Sicht kumulative Belohnung. Die Schwierigkeit liegt darin solch ein Belohnungssystem zu entwerfen, wodurch der Agent bei der Maximierung der Belohnung das gedachte Ziel erreicht. Die Belohnungen müssen daher den Weg zum Endziel aufweisen.

RL wird in zwei Arten aufgeteilt. Das Lernen kann dabei mit oder ohne ein Modell der Umgebung durchgeführt werden. Bei dem modelbasierten Lernen wird durch gesammelte Daten ein Modell erstellt, welches dazu benutzt wird mittels Wertefunktionen eine Policy zu entwickeln. Diese Variante ist sehr aufwendig und risikobehaftet, denn ein falsches Modell kann das Lernen stark beeinflussen. Vorteil dabei ist, dass der Agent in die nächsten Zustände vorausschauen kann und somit seine Aktionen und die Policy optimal planen kann (siehe Abb. 2). Dahingegen wird bei modellosem Lernen direkt aus ausführenden Aktionen und den resultierenden Daten eine Policy entwickelt. Der Agent kann dabei auch in terminierenden Zuständen landen, in denen das Zurücksetzen der Umgebung nötig ist. Innerhalb dieser Arbeit wird das modelfreie Lernen angewendet.

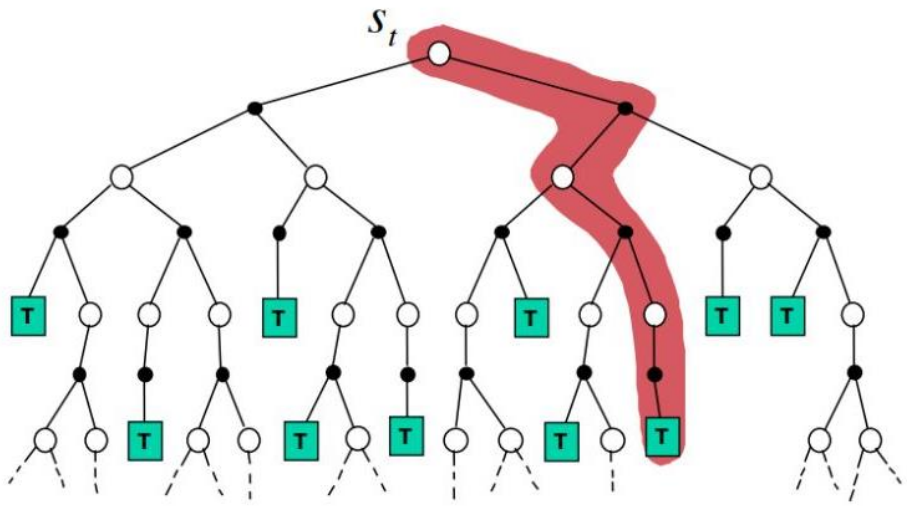

Abbildung 2 Beispiel einer erlernten Policy Barto (2017)

2.1.1 Markov Decision Process (MDP)

Markov Decision Process (MDP) ist der Grundstein des RLs. Es dient dazu sequenzielle Entscheidungen zu formalisieren. Diese Formalisierung ist die Grundlage für die Strukturierung von Problemen, die durch RL gelöst werden. Fast alle RL-Probleme können als MDPs formalisiert werden.

Ein wichtiger Teil des MDP ist die Markov Property. Diese besagt, dass falls der gegenwärtige Stand des Wissens bekannt ist, kann der bisherige Informationsverlauf verworfen werden. Denn dieser Zustand ist eine ausreichende Information, die den gesamten bisherigen Informationsverlauf beinhaltet. Markov Property beschreibt, dass man die Zustände der Vergangenheit nicht kennen muss, um eine optimale Entscheidung aus dem jetzigen Zustand zu treffen. Man benötigt lediglich den aktuellen Zustand und die aktuelle Aktion zum Zeitpunkt t, denn diese können bereits alle nötige Information enthalten. Damit diese Voraussetzung erfüllt werden kann, müssen sich alle vorherigen Ereignisse in einer Weise zusammenfassen können, so dass alle relevanten Informationen erhaltet bleiben.

$$p\,(s_{t+1}|s_t) = p[s_{t+1}|s_1 = s, \dots, s_t] \qquad (1)$$

Trifft bei einer Umgebung diese Voraussetzung zu, so lässt sich die Komplexität beim Beschreiben der Umgebung deutlich reduzieren. Diese Umgebung kann als MDP bezeichnet werden.

Eine MDP-Umgebung wird durch eine endliche Anzahl an Zuständen, Aktionen und dessen Beschreibung definiert. In Anbetracht jedes Zustandes und jeder Aktion lässt sich die Wahrscheinlichkeit jedes möglichen nächsten Zustandes wie folgt beschreiben:

$$p\,(s', r|s, a) = p[s_{t+1} = s', r_{t+1} = r|s_t = s, a_t = a,] \qquad (2)$$

Diese Werte werden Übergangswahrscheinlichkeiten (*transition probabilities*) genannt. Diese beschreiben die Wahrscheinlichkeit, dass die aus Zustand s_t durchgeführte Aktion a_t und resultierend in Zustand s_{t+1} berechnete Belohnung die folgende sein kann:

$$r\,(s, a, s') = E[r_{t+1} | s_t = s, a_t = a, s_{t+1} = s'] \qquad (3)$$

Die Größen $p\,(s', r|s, a)$ und $r\,(s, a, s')$ geben die wichtigsten Aspekte der Dynamik eines MDP an.

2.1.2 Belohnungen

Das Ziel des Agenten besteht also darin, die Belohnung, die er auf lange Sicht erhält, zu maximieren. Die Ziele müssen zunächst zusammengefasst und formalisiert werden. Dazu benutzt man das Konzept des erwarteten Wertes (Return) der Belohnungen. Der Erwartungswert **G** wird dabei als eine Summe aller erhaltenen Belohnungen über den Zeitraum t = 0 bis t = T definiert, wobei T der finale Zeitschritt ist. Mathematisch kann der Erwartungswert zum Zeitpunkt t wie folgt formuliert werden:

$$G_t = r_{t+1} + r_{t+2} + r_{t+3} + \cdots + r_T \qquad (4)$$

Sinnvoll ist dieses Konzept vor allem bei Anwendungen, in welchen ein natürlicher Endzeitschritt und folgend ein Endzustand vorkommt. Solche Endzustände können neben dem endgültigen Ziel auch durch Transitionen hervorgerufen werden, die zum Tod des Agenten führen. Wird solch ein Endzustand eingenommen, wird der Agent inkl. der Umgebung in einen Startzustand versetzt. Sind die Zeitschritte endlich, so wird diese Art von Aufgaben als *episodic tasks* bezeichnet. Es gibt jedoch Fälle, bei denen die Interaktionen zwischen dem Agenten und der Umgebung kontinuierlich sind. Diese enden somit in keinem Endzustand. In dem Fall ist Gt = **∞,** da es keinen definierten finalen Zeitschritt gibt. Diese Aufgaben werden als *continuing tasks bezeichnet.*

Ein weiteres wichtiges Konzept ist das *discounting*. Bei diesem Konzept wird ein Discount Faktor γ eingeführt, wobei es $0 \leq \gamma \leq 1$ gilt. Darüber hinaus wird nicht die Summe aller Belohnungen, sondern die Summe der ermäßigten Belohnungen

(Discount Summe) maximiert. Insbesondere wählt der Agent zu jedem Zeitpunkt t die Aktion *At*, um die erwartete Discount Summe zu maximieren.

$$Gt = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \qquad (5)$$

Der Discount Faktor bestimmt den Wert der zukünftigen Belohnung. Eine Belohnung nach k Schritten hat den Wert γ^{k-1} von dem was sie Wert wäre, wenn diese sofort erhalten wurde. Ist $\gamma = 0$ so agiert der Agent „kurzsichtig“ und zieht Aktionen vor, die eine sofortige Belohnung nach sich ziehen. Bei $\gamma < 1$ hingegen berücksichtigt der Agent zukünftige Belohnungen stärker und wird somit weitsichtiger.

Um *episodic* und *continuing* Aufgaben mit nur einer Notation zu beschreiben, wird eine weitere Konvention eingeführt. Für die erste Art wurde der Erwartungswert als eine Summe über eine endliche Anzahl von Zeitschritten definiert. Für die zweite Art wird das als eine unendliche Anzahl von Zeitschritten definiert. Beide Arten können vereinheitlicht werden, in dem das Beenden einer Episode als einen Eintritt in einen Zustand betrachtet wird, der eine Dauerschleife darstellt (siehe Abb. 3).

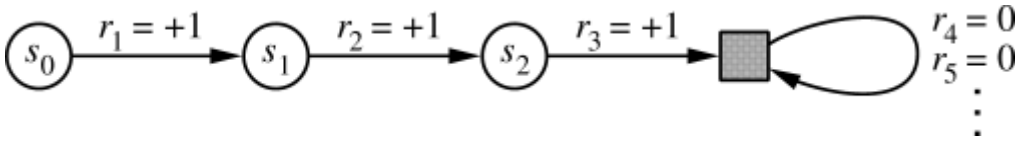

Abbildung 3 Beispiel eines in sich wiederkehrenden Zustandes Barto (2017)

Das Quadrat stellt ein Zustand dar, der dem Ende einer Episode entspricht. Ausgehend von Zustand s_0 ist die Belohnungssequenz 1+1+1+0+0+ …. Die endgültige Summe der Belohnungen bleibt bei $T = 3$ oder $T = \infty$ gleich. Auch beim Einsatz des Discount Faktors würde sich die Summe nicht ändern. Die Notation, die für beide Fälle gelten soll, lässt sich folgendermaßen definieren:

$$Gt = \sum_{k=0}^{T} \gamma^k R_{t+k+1} \qquad (6)$$

Dies erlaubt, dass innerhalb einer Aufgabe entweder $T = \infty$ oder $\gamma = 1$ betragen kann.

2.1.3 Policy

Die Policy ist eine Art Strategie und Regelwerk, nach welcher der Agent in RL agiert. Das Konzept der Policy π beschreibt die Wahrscheinlichkeit zum Ausführen einer Aktion $a \in A(s)$, aus einem Zustand $s \in S$. Folgt der Agent der Policy π zum Zeitpunkt t, so ist die Wahrscheinlichkeit $\pi = (a \mid s)$, dass die Aktion $A_t = a$ ausgewählt wird, falls $S_t = s$ beträgt. Die Policy beinhaltet im Endeffekt eine Trajektorie der Aktionen zum Erhalt der maximalen Belohnung auf langer Sicht. Mit Hilfe von verschiedenen Methoden werden die Policys entwickelt und miteinander verglichen (Siehe Abb. 2).

2.1.4 Value Funktion

Die meisten RL-Algorithmen basieren auf dem Einsatz der *Value* Funktion. Mit Hilfe dieser Funktionen wird abgeschätzt wie belohnend eine Aktion aus einem bestimmten Zustand sein wird oder wie vorteilhaft es ist für den Agenten sich in einem bestimmten Zustand zu befinden. Wie belohnend oder vorteilhaft eine Aktion ist, bezieht sich dabei auf die sofortigen oder zukünftigen Belohnungen der Aktionen. Die zukünftigen Belohnungen des Agenten sind davon abhängig, welche Maßnahmen er ergreifen wird. Dementsprechend werden Value Funktionen in Bezug auf bestimmte Richtlinien definiert. In RL gibt es zwei Arten der Value Funktionen. Diese sind das State-Value und Action-Value Funktionen.

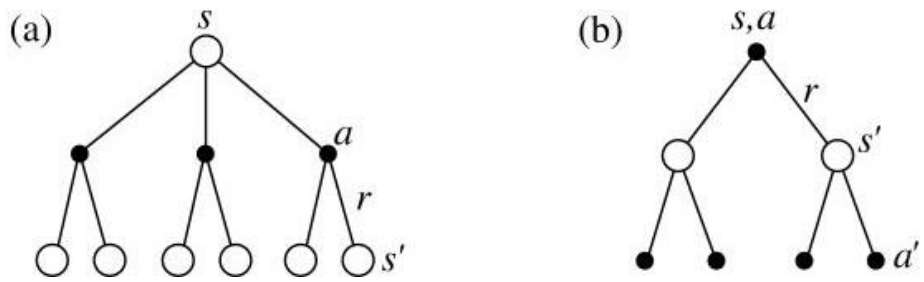

Abbildung 4 (a) V_π (b) Q_π Barto (2017)

Die State-Value Funktion V_π für Policy π, gibt an wie gut ein Zustand für einen Agenten ist, welche die Policy π verfolgt. Es beschreibt somit den Wert (Value) eines Zustandes unter Policy π, welches der rabattierten Belohnung beim Starten aus Zustand s zum

Zeitpunkt t und folgen der jeweiligen Policy π gleicht. Mathematisch wird $V\pi$ wie folgt beschrieben, wobei die Komponente E_π die erwartete rabattierte Belohnung angibt.

$$V_\pi(s) = E_\pi[\, G_t \,|S_t = s] = E\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s\right] \qquad (7)$$

Ähnlich wird Action-Value Funktion Q_π für eine Aktion a im Zustand s unter Policy π beschrieben. Diese Funktion erweitert die V_π Formel um die Aktion A_t. Dadurch wird die erwartete rabattierte Belohnung anhand der Aktion aus einem Zustand unter einer bestimmten Policy berechnet.

$$Q_\pi(s,a) = E_\pi[\, G_t \,|S_t = s, A_t = a] = E\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s, A_t = a\right] \qquad (8)$$

Andere Bezeichnung für Q_π ist die *Q-Funktion.* Das Zustand-Aktionspaar wird in dem Zusammenhang als *Q-Value* bezeichnet. Das „Q“ steht dabei für Qualität der durchgeführten Aktion aus einem bestimmten Zustand.

Die beiden Größen V_π und Q_π können durch Erfahrungswerte bestimmt werden. Es werden alle Belohnungen der Zustände einer ganzen Episode ermittelt. Der daraus berechnete Durchschnittwert konvergiert je nach der Value Funktion gegen V_π oder Q_π. Schätzungsmethoden dieser Art werden *Monte Carlo Methods* genannt. Voraussetzung hierbei ist Vorhandensein eines Endzustands, da sonst die rabattierte Belohnung Gt nicht berechnet werden kann. Somit gelten diese Methoden nur für episodische Aufgaben.

Eine grundlegende Eigenschaft von Value-Funktionen, besteht darin, dass sie bestimmte rekursive Beziehungen erfüllen. Für jede Policy und jeden Zustand gilt zwischen dem Zustand S_t und dem möglichen Folgezustand S_{t+1} die folgende Bedingung namens Bellmann Gleichung:

$$V_\pi(s) = E_\pi[\, R_t \,|S_t = s]$$

$$= E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \,|S_t = s\right]$$

$$= E_\pi\left[R_{t+1} + \gamma \sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_t = s\right]$$

$$= \sum_a \pi = (a \mid s) \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma E_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+2} \mid S_{t+1} = s'\right]\right]$$

$$= \sum_a \pi = (a \mid s) \sum_{s'} p(s' \mid s, a) \left[r(s, a, s') + \gamma V_\pi(s')\right] \quad (9)$$

Diese Gleichung ermöglicht es in den Folgezustand zu blicken bzw. den zu beschreiben. Ist das Value des Zustandes S_{t+1} bekannt, so lässt sich auch das Value für S_t berechnen. Wie im Backupdiagramm (siehe Abb. 4a) zu sehen ist, lässt es sich aus Startzustand s (leerer Kreis) eines aus drei möglichen Aktionen (gefüllter Kreis) ausführen. Die Umgebung reagiert mit einer Belohnung r und einem Folgezustand s'. Die Bellman Gleichung bildet einen Durchschnitt über alle Übergangsmöglichkeiten und gewichtet jede mit ihrer Eintrittswahrscheinlichkeit. Es wird definiert, dass State-Value eines Startzustandes s laut der Bellman Gleichung die rabattierte State-Value des eintretenden Folgezustandes s' addiert mit der unmittelbaren Belohnung entlang des Weges entsprechen muss.

2.1.5 Optimale Value Funktionen

Das erfolgreiche Lösen einer Aufgabe in RL bedeutet eine optimale Policy zu finden. Mit Hilfe der Value Funktionen ist es möglich die Policys bezüglich ihrer Qualität zu vergleichen. Eine Policy π gilt als gleicht gut oder besser als eine andere Policy π', wenn der Erwartungswert aller Zustände größer oder gleich der Policy π' ist. Somit gilt, dass $\pi \geq \pi'$ nur dann gilt, wenn $V_\pi(s) \geq V_{\pi'}(s)$ für alle Zustände zutrifft. Zudem existiert immer mindestens eine Policy die gleicht gut oder besser ist als die restlichen. Diese gilt als die optimale Policy π* und besitzt die optimale State-Value Funktion V*. Diese wird wie folgt für alle $s \in S$ definiert:

$$V^*(s) = \max_\pi V_\pi(s) \quad (10)$$

Optimale Policys besitzen zudem auch eine optimale Action-Value Funktion Q*, die wie folgend für alle $s \in S$ und $a \in A(s)$ definiert wird:

$$Q^*(s, a) = \max_\pi Q_\pi(s, a) \quad (11)$$

Für jedes Zustand-Aktionspaar (s, a) liefert diese Funktion die erwartete Belohnung für das Ausführen der Aktion a im Zustand s und folgen der optimalen Policy. Demnach lässt sich $Q*$ als V^* folgend definieren:

$$Q^*(s,a) = E\ [R_{t+1} + \gamma\, V^*(s_{t+1}) |\ s_t = s, a_t = a\ ] \qquad (12)$$

Für die optimale Policys muss ebenfalls die optimale Bellman Gleichung erfüllt werden. In dem Fall wird diese Bellman Optimalitätsgleichung genannt. Bei der optimalen Value-Funktion lässt sich die Bellman Optimalitätsgleichung ohne einen Bezug auf eine Policy beschreiben. Es sagt aus, dass die optimale State-Value der erwarteten Belohnung betragen muss, die von der bestmöglichen Aktion aus dem jeweiligen Zustand stammt.

Die Bellman Optimalitätsgleichungen für $V*$ und Q^* werden wie folgt definiert:

$$V^*(s) = \max_{a \in A(s)} \sum_{s'} P(s'|s,a)\ [R(s,a,s') + \gamma V^*(s')] \qquad (13)$$

$$Q^*(s,a) = \sum_{s'} P(s'|s,a)\ \left[R(s,a,s') + \gamma \max_{a'} Q^*(s',a')\right] \qquad (14)$$

In Abbildung werden Spannweiten der Aktionen und Zustände bezüglich der beiden Bellman Optimalitätsgleichungen graphisch dargestellt. Dabei soll verdeutlicht werden, dass nun die optimale Policy gewählt wird.

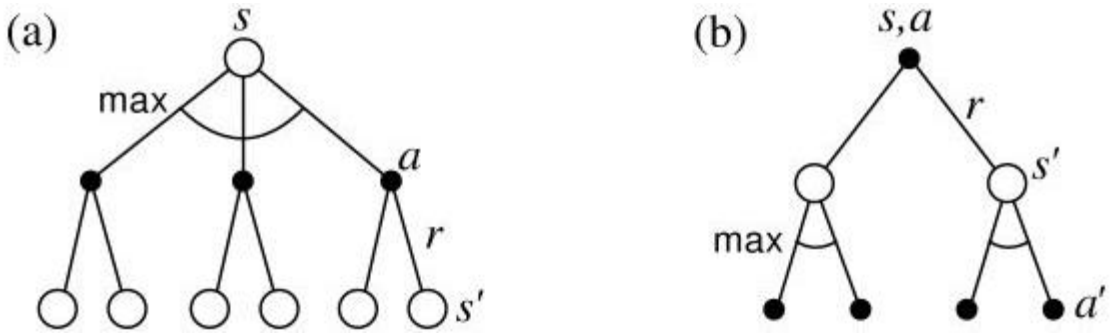

Abbildung 5 Backupdiagramme für a) $V*$ und b) Q^* Barto (2017)

Mittels der Bellman Optimalitätsgleichungen lassen sich endliche MDPs lösen. Es wird in Form eines ganzen Gleichungssystems realisiert, sodass für N Zustände sich N Gleichung mit N unbekannten ergeben. Voraussetzung zum Lösen der Values $V*$ und Q^* ist, dass $P(s'|s,a)$ und $R(s,a,s')$ dabei bekannt sein müssen.

Soweit V^* vorhanden ist, lässt sich die optimale Policy ermitteln. Für jeden Zustand s, gibt es mindestens eine Aktion bei dem sich das maximum in der Bellman Optimalitätsgleichungen ergibt. Jede Policy, die diesen Aktionen eine Wahrscheinlichkeit ungleich Null zuweist, gilt als optimal. Es ist eine Suche in einer Reichweite eines Schritts. Andersrum gesagt verhält sich diese Policy bezüglich V^* greedy und ist optimal. Dieser Begriff wird verwendet, um Suchverfahren zu beschreiben, die auf kurzsichtige Belohnungen basieren und vernachlässigt die Möglichkeit zukünftig bessere Alternativen zu erlangen. In bestimmten Fällen kann dies aber auch zur langfristigen optimalen Policy werden.

Bei Vorhandensein des Q^* ist lediglich die Aktion aus Zustand s zu finden die $Q^*(s, a)$ maximiert. Die Action-Value Funktion speichert bereits die beste Aktion zum Maximieren der Belohnung. Dank des Aktion-Zustandspaares kann die optimale Action-Value Funktion die beste Aktion auswählen, ohne etwas über Folgeschritte wissen zu müssen.

Auf diese Weise ist es nun möglich die optimale Policy zu finden. Da mit diesem Algorithmus alle Möglichkeiten der erwarteten Belohnung berechnet werden müssen, ist eine große Rechenleistung nötig. Außerdem müssen mehrere Voraussetzungen erfüllt werden. Solche sind das nötige Wissen über die Dynamik der Umgebung, genügend Rechenleistung und die Markov Property. Diese Voraussetzungen können nicht in jedem System erfüllt werden. Es gibt eine Reihe an Methoden zur Policy-Auswertung und Policy-Verbesserung wie das dynamische Programmieren oder die Monte Carlo-Methoden. Sie stellen zwar die nötigen Grundlagen dar, sind aber eher theoretischer Natur. Daher muss bei der Ermittlung der Policy auf Approximation gesetzt werden.

2.1.6 Q-Learning

Das Q-Learning ist ein modelloses Lernalgorithmus zum Finden einer Policy in einem MDP. Dabei handelt es sich um einen Off-Policy-Algorithmus, da der Schätzwert der nächsten Aktion ohne Nutzung der Policy bestimmt wird. Bei diesem Verfahren wird mittels der Q-Funktion ein Tupel des Zustandes mit jeder Aktion gebildet. Das Ziel

besteht darin, die optimale Policy zu finden, in dem die optimalen Q-Values für jedes Zustand-Aktionspaar ermittelt werden. Die Werte der Q-Funktion werden mit jeder Episode neu berechnet. Zur Berechnung des Q-Values wird bei diesem Verfahren die folgende Gleichung eingesetzt:

$$Q(s_t, a_t) = Q(s_t, a_t) + \alpha\,[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)] \qquad (15)$$

Wobei:

- $Q(s_t, a_t)$: Q-Value des Zustandes s_t
- $0 \leq \alpha \leq 1$: Formelzeichen α beschreibt die eingesetzte Lernrate. Bei einer hohen Lernrate nimmt der Agent den neuen Q-Wert schneller an. So wird bei einer Lernrate von 1 der neue Q-Wert komplett übernehmen, anstatt den alten Wert anzupassen.
- r_{t+1}: Resultierende Belohnung der Aktion a_t aus Zustand s_t
- $0 \leq \gamma \leq 1$: Discount Faktor. Ist dieser Wert gleich null, so werden Aktionen mit sofortiger Belohnung vorgezogen. Werte von eins lassen den Agenten auf lange Sicht größere Belohnungen einsammeln.
- $\gamma \max_a Q(s_{t+1}, a)$: Der höchste Q-Value aus Zustand s_{t+1} mit Aktion a

Die Funktionsweise von Q-Learning lässt sich am besten anhand eines Beispiels in Abbildung 5 erklären. Bei diesem Scenario startet der Agent in dem Startfeld und hat die Aufgabe in der kürzesten Zeit seinen Punktestand zu maximieren. Diese Welt hat zwei terminierende Zustände. Eines mit einer positiven Belohnung von zehn und ein weiteres mit einer negativen Belohnung von zehn. Der Bewegungsraum des Agenten beschränkt sich auf die Bewegung nach oben, unten, links und rechts um jeweils ein Feld pro Zeitschritt. Der Zustand des Agenten entspricht dabei dem Ort in dieser 3x3 Spielumgebung. Neben den zwei beschriebenen Feldern kriegt der Agent eine negative Belohnung von 1, wenn er in ein leeres Feld übergeht und eine Belohnung von 1 wenn er das Feld 1/1 erreicht.

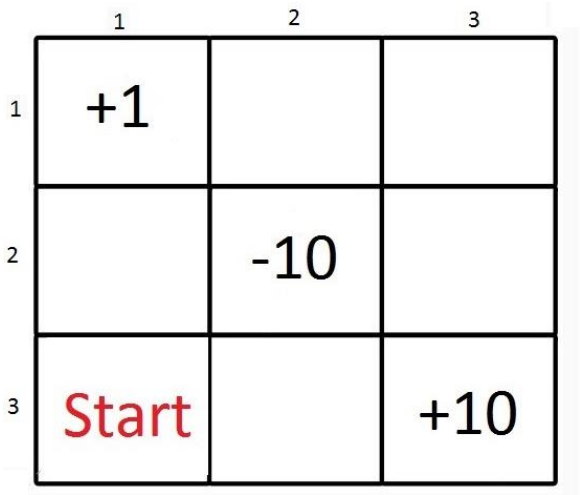

Abbildung 6 Eine 3x3 Spielwelt

Zu Beginn des Lernprozesses wird eine Q-Tabelle erstellt. Bei dieser Tabelle werden über alle Zustände alle Aktionen aufgespannt. Die Dimension der Tabelle entspricht dabei Anzahl der Aktionen mal die Anzahl der Zustände.

	Oben	Unten	Links	Rechts
1/3	-1	0	0	-1
1/2	+1	-1	0	-10
1/1	0	-1	0	-1
2/3	-10	0	-1	+10
2/2	0	0	0	0
2/1	0	-10	+1	-1
3/3	0	0	0	0
3/2	-1	+10	-10	0
3/1	0	-1	-1	0

Abbildung 7 Beispiel einer Q-Tabelle

Da noch kein Wissen über die Umgebung gesammelt wurde, wird die Tabelle am Anfang des Prozesses mit nullen initialisiert. Im Laufe des Trainings sollen darin die Q-Values aus Gleichung für jedes Zustand-Aktionspaar abgespeichert und mit jeder Lernepisode aktualisiert werden. Nach einigen Lernepisoden, wenn die Tabelle immer

mehr ausgefüllt wird, kann der Agent basierend auf seinen Zustand die beste Aktion auswählen. Dabei maximiert er seine Belohnung und weicht Zuständen aus, die zu einer negativen Belohnung führen.

Das Lernen erfolgt dabei episodenweise. Jede Episode beginnt der Agent mit der Auswahl einer Aktion aus dem Startzustand basierend auf den aktuellen Q-Values in der Tabelle. Es wird die Aktion gewählt, die den höchsten Q-Value in der Tabelle für den aktuellen Zustand aufweist. Da aber die Tabelle noch leer ist, kann der Agent theoretisch keine Aktion auswählen. Dazu wird eine Technik namens *Exploration* und E*xploitation* eingeführt. Unter Exploration versteht man die Erkundung der Umgebung, um erste Informationen zu sammeln. Exploitation ist hingegen der Akt der Ausnutzung der bereits vorhandenen Information der Umgebung, um zum Ziel zu kommen. Diese Technik hilft dabei alle Möglichkeiten und Zustände der Umgebung zu erkunden, um letztendlich die beste Policy entwickeln zu können. Den sollte der Agent bereits nach einigen Episoden nur noch auf die vorhandenen Daten zurückgreifen, könnte der Zustand mit der größten Belohnung nicht erforscht werden.

Um eine Balance zwischen Exploration und Exploitation zu kriegen, wird die Epsilon Greedy Technik angewendet. Dabei wird der Wert für Exploration (Epsilon) auf eins gesetzt. Es beschreibt eine Wahrscheinlichkeit zum Ausführen einer zufälligen Aktion, wobei 1 = 100% entspricht. Des Weiteren wird für diesen Wert ein Discountfaktor eingeführt, welcher den Wert der Exploration nach jeder ausgeführten Episode verkleinert. Dadurch wird der Agent in Bezug auf die Nutzung der Umgebung „gierig“ (greedy) und fängt nun mittels der Q-Tabelle an die Aktionen auszuwählen. Im Falle von Programmcode wird dies in Form eine If-Abfrage realisiert.

Um die Q-Values zu aktualisieren, wird die bereits vorgestellte Gleichung (15) verwendet. Mit jedem Lernschritt fließt der neue und der alte Q-Wert in die Berechnung mit ein. Der erste Q-Wert kann wie folgt berechnet werden, mit der Annahme, dass die Lernrate 0.5 und Epsilon 0.99 beträgt:

$$Qneu(s_t, a_t) = Qalt(s_t, a_t) + \alpha\,[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Qalt(s_t, a_t)]$$

$$= 0 + 0.5\,[(-1) + 0.99 * 0 - 0]$$

$$= -0.5$$

Der neue Q-Wert beträgt somit -0.5 und wird in der Q-Tabelle gespeichert. Am Ende des Lernvorgang entsteht anhand der Tabelle eine Policy, welcher der Agent folgen kann. Dieses Verfahren ist vor allem für stochastische Umgebungen sehr sinnvoll. Denn unabhängig von der verfolgten Policy ist dieses Verfahren in der Lage bei genügend Erfahrungswerten aus besuchten Zuständen die optimale Policy zu finden.

2.2 DQN

Das Deep Q-network (DQN) ist ein weiterer Algorithmus zur Approximation des Q-Wertes. Dabei verbindet es das RL mit Deep Learning und neuronalen Netzwerken. Das neuronale Netz hat dabei die Rolle des Agenten und erlaubt es für die Approximation der Q-Werte den Zustandsbereich der Anwendung drastisch zu erweitern, wodurch sich komplexere Umgebungen realisieren lassen. Das Training erfolgt dabei nach dem Supervised Learning Prinzip.

2.2.1 Zusatzfunktionen

Für das DQN-Algorithmus gibt es eine Reihe an Verbesserungsmethoden, um die Effektivität und die Trainingsgeschwindigkeit zu steigern. Innerhalb dieser Arbeit werden einige dieser Methoden erklärt und in den Lernalgorithmus implementiert.

Experience Replay Speicher

Durch den Einsatz eines spezifischen Roboters für eine spezielle Aufgabe sind fertige Datensätze nicht vorhanden. Aufgrund dessen müssen eigene Trainingsdaten gesammelt werden. Die Datensätze werden mittels der Epsilon Greedy Strategie und anschließender Vorhersage durch das Netzwerk gesammelt. Um die Datensätze anschließend abzuspeichern, wird die Methode des Memory Replay Speichers angewendet. Dabei handelt es sich um einen Buffer mit einer festen Größe. In diesen Buffer werden schrittweise die gesammelten Erfahrungen in Form eines Datensatzes

(s, a, r, s‘, d) des Agenten gespeichert. Zudem wird der Datensatz zusätzlich in einer CSV-Datei abgespeichert, um den Experience Replay Speicher bei nächstem Versuch neu zu füllen. Vor dem Training wird per Zufallsgenerator ein Teil des Speichers entnommen, wobei die Größe der entnommenen Datenreihe die Batchsize des Netzwerkes vorgibt. Durch diesen Vorgang wird die Korrelation innerhalb der Daten entfernt, wodurch das Netzwerk besser trainiert werden kann. Anschließend wird die Batchsize in das Netzwerk eingespeist und das Modell trainiert.

s_1, a_1, r_2, s_3, d_1
s_2, a_2, r_2, s_3, d_2
…
$s_t, a_t, r_t, s_{t+1}, d_t$

Abbildung 8 Beispiel des Experience Replay Speichers

Target Network

Beim Trainieren eines Netzwerkes mit der Supervised Learning Methode wird während des Trainings der Wert zwischen der Soll- und Ist-Ausgabe berechnet. In RL wird analog eine Differenz zwischen Zielwert des Q-Wertes (Q-Target) und dem aktuellen Q-Wert berechnet. Der Soll-Q-Wert ist dabei aber nicht bekannt und muss mittels der Bellman-Gleichung ermittelt werden. Die Bellman-Gleichung für diesen Algorithmus wird dabei wie folgt definiert:

$$Q(s_t, a) = r + \gamma * \max_a Q(s_{t+1}, a) \quad (16)$$

Wobei:

- Q: Berechneter Q-Wert
- r: Belohnung
- γ: Discountfaktor Gamma

- $\max_a Q(s_{t+1}, a)$: Der maximale Q-Wert aller möglichen Aktionen aus nächstem Zustand. Innerhalb dieser Arbeit wird es als $next\_target$ definiert.

Im Normalfall werden zum Bestimmen des Q-Targets und des aktuellen Q-Wertes die gleichen Netzwerkparameter benutzt. Dadurch besteht eine große Korrelation zwischen dem Q-Target und der Änderung der Gewichtseinstellung des Netzwerkes. Dies führt dazu, dass bei jedem Trainingsschritt sich neben den Q-Werten auch das Q-Target verändert. Man versucht also ein Zielwert zu erlangen, welcher sich ebenfalls ständig ändert. Diese Vorgehensweise führt zu einer Oszillation während des Trainings.

Um das Problem zu beheben, wird neben dem DQN-Netzwerk ein Target-Netzwerk eingeführt. Die beiden Netzwerke haben dabei einen identischen Aufbau. Das DQN-Netzwerk hat weiterhin die Aufgabe die Q-Werte auszugeben und das Netzwerk für das aktuelle Q-Target anzupassen. Das Target Network aktualisiert dabei alle n Schritte das Q-Target des DQN-Netzwerkes. Dadurch kann das DQN-Netzwerk n Schritte lang mit festem Q-Target trainiert werden. Als Folge entsteht ein stabileres Trainingsverhalten.

2.3 Deep Learning

Wie auch das RL ist Deep Learning ein Teilgebiet des Machine Learnings. Es ist eine Methode der Informationsverarbeitung und beruht auf dem Einsatz der neuronalen Netzwerke. Es geht dabei, um das Finden von Mustern und Gemeinsamkeiten aus einem vorhandenem Datensatz. Sind diese gefunden, ist das Programm in der Lage Entscheidungen auf Basis der Zusammenhänge der gelernten und der neuen Daten zu treffen. Es kann bei Klassifikation von Bildern, Prognosen über einen Aktienstand, eine Spracherkennung oder sogar beim autonomen Fahren verwendet werden. Die Einsatzbereiche des Deep Learnings sind da, wo sich genügend Beispieldaten sammeln lassen. Im Wesentlichen wird das Deep Learning in zwei Teile aufgeteilt. Es ist das Supervised und Unsupervised Learning.

Bei dem Supervised Learning ist ein vorhandener Datensatz und deren richtige Klassifizierung eine Voraussetzung. Man gibt Daten und die richtigen Ausgaben somit vor. Das System versucht aus dem vorgegebenen Datensatz aus Musterbeispielen Gemeinsamkeiten zu finden. Dies geschieht unter richtiger Einstellung diverser Parameter innerhalb des neuronalen Netzes.

Bei Unsupervised Learning ist ebenso ein vorhandener Datensatz erforderlich. Der Unterschied ist, dass hierbei die richtigen Ausgaben, also die Klassen nicht bekannt sind. Das System muss demnach selbst den Datensatz nach Kategorien aufteilen, welche sich nach gelernten Merkmalen unterscheiden.

Das Anpassen der Parameter des Netzwerkes und somit das Anpassen des Modells an die Trainingsdaten nennt sich das Trainieren oder Lernen. Dies geschieht unter der Verwendung von künstlichen neuronalen Netzwerken. Ein neuronales Netzwerk im Deep Learning besteht mindestens aus drei Schichten (Layer) (siehe Abb.9). Die erste und die letzte Schicht sind die Ein- und Ausgabeschichten. Als Eingabe fungieren physikalische Daten wie Sensordaten, wogegen die Ausgabeschicht als eine Klassifikationsausgabe dient. Die zwischen der Ein- und Ausgabeschicht sich befindlichen Schichten sind sogenannte verdeckte Schichten. Diese Schichten sind verantwortlich für das Lernen eines Netzwerkes. Je nach Anwendungsfall existieren viele Arten dieser Schichten, ob für Verarbeitung von 2D Bilddaten oder einfachen Zahlen. Die Schichten bestehen dabei aus einer voreingestellten Anzahl an Neuronen.

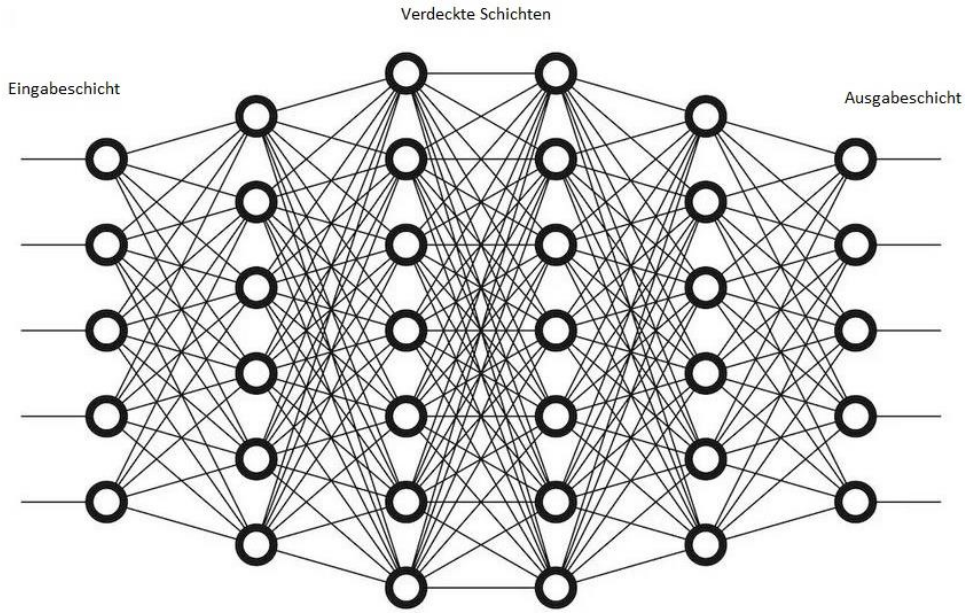

Abbildung 9 Neuronales Netzwerk

2.3.1 Aufbau eines Neurons

Aufbau eines Neurons (siehe Abb.10) besteht aus Eingängen O_n, Schwellwert θ_j (Bias), Aktivierungsfunktion und einer Ausgabe O_j. Ob ein Eingang zu dem Neuron aktiviert wird, hängt von dem Gewicht w_{nj} des vorherigen Neurons ab, wobei n die Eingangsnummer und j die Nummer des Neurons angibt. Das Zwischenergebnis net_j eines Neurons wird durch folgende Gleichung beschrieben:

$$net_j = \sum_n O_{j-1} w_{nj} + \theta_j \qquad (17)$$

Anschließend wird das Zwischenergebnis net_j in eine nicht lineare Funktion $f()$ (Aktivierungsfunktion) eingespeist. Dies ermöglicht verschiedene Daten zu klassifizieren, die mittels einer linearen Funktion nicht kategorisiert werden können (siehe Abb. 11). Die Ausgabe O_j gilt als Eingangsgewicht für weitere Neuronen. Das Hinzufügen weiterer Schichten mit neuen Neuronen erweitert die Komplexität des Netzwerkes. Dadurch ist es möglich die Daten so anzupassen, dass diese in der Ausgabeschicht durch eine lineare Funktion klassifiziert werden können.

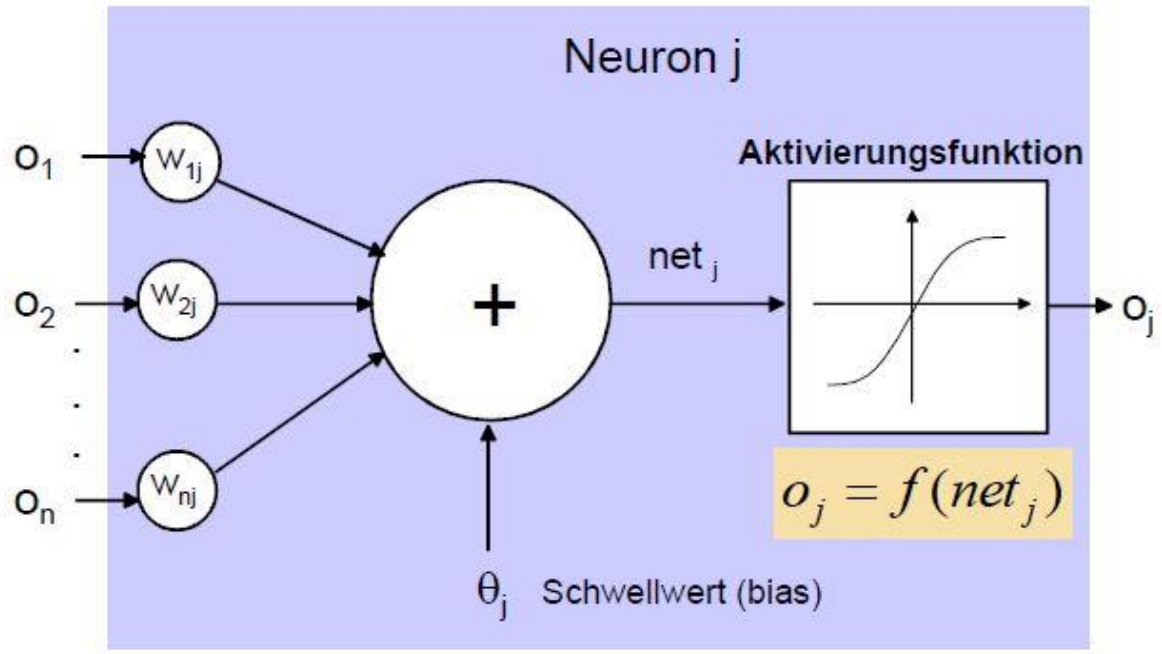

Abbildung 10 Aufbau eines Neurons Meisel (2012)

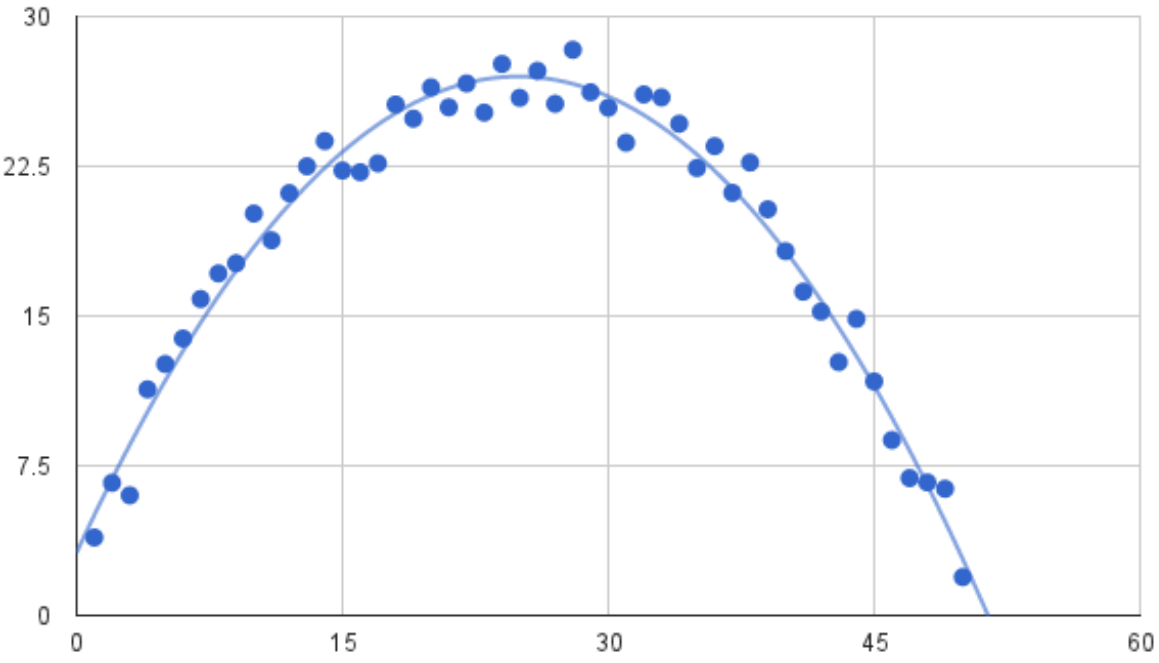

Abbildung 11 Einsatz einer nicht linearen Funktion Sagar (2017)

2.3.2 Gradient Descent

Beim Trainieren wird das Modell so an die Trainingsdaten angepasst, dass bei der Eingabe der Daten die richtigen Ausgaben stattfinden. Dabei wird am Ende jeder Iteration mit Hilfe von quadratischen Kostenfunktionen eine Differenz zwischen der Ist- und Soll-Ausgabe gebildet. Der Wert der Kostenfunktion sagt direkt aus wie gut ein Modell ist. Alle verwendeten Gewichte der Neuronen dieser Iteration werden als eine Gewichtseinstellung zusammengefasst. Das Ziel ist es die Gewichtseinstellung so anzupassen, dass für die Kostenfunktion ein globales Minimum gefunden wird (siehe Abb. 12) und somit die Differenz entfernt wird.

Die Gleichung der Kostenfunktion kann je nach Anwendung variieren. So setzt man zum Beispiel für lineare Regressionsprobleme eine Kostenfunktion des Mean Square Errors (MSE) und für Klassifikationsanwendungen die Cross-Entropy Kostenfunktion ein. Eine typische Kostenfunktion E des MSE sieht wie folgt aus:

$$E = \frac{1}{2m}\sum_{i=1}^{m}(Y_i - O_i)^2 \qquad (18)$$

Wobei:

- E: Fehler
- m: Anzahl der Trainingsbeispiele
- Y_i: Ist-Ausgabe des Modells im Trainingsbeispiel i
- O_i: gewünschte Soll-Ausgabe im Trainingsbeispiel i

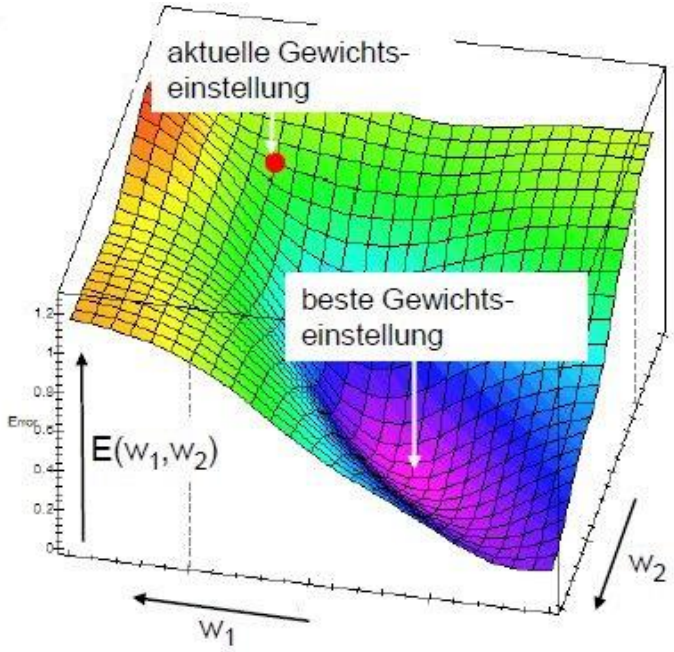

Abbildung 12 Gewichtseinstellungen in einer Kostenfunktion Meisel (2012)

Da die Kostenfunktion die Gewichtseinstellung repräsentiert, ließe sich diese durch das simple Ausprobieren von anderen Gewichten verändern. Dieser Weg ist aber schon bei einer kleinen Neuronenanzahl sehr langsam und nicht zielorientiert. Zudem wird ein Gewicht am Anfang des Netzes verändert, so verändert dieser auch alle nachfolgenden Gewichte. Als Folge verändert sich die Ausgabe und die Kostenfunktion. Alternativ verwendet man eine Technik namens „Gradientenabstieg“ (Gradient Descent). Dabei wird für die Kostenfunktion ein Gradient bestimmt, dem es zu folgen gilt. Der Gradient wird anschließend mit jeder Iteration neu berechnet und aktualisiert. Dies gilt solange bis die Kostenfunktion sich mit einer vordefinierten Schrittweite in die richtige Richtung begibt und anschließend das globale Minimum erreicht. Die Schrittweite mit dem der Gradient aktualisiert wird, wird durch die Angabe der Lernrate η definiert. Auswirkung von verschiedenen Werten einer Lernrate an der Kostenfunktion kann durch folgende Abbildung (siehe Abb. 13) verdeutlich werden.

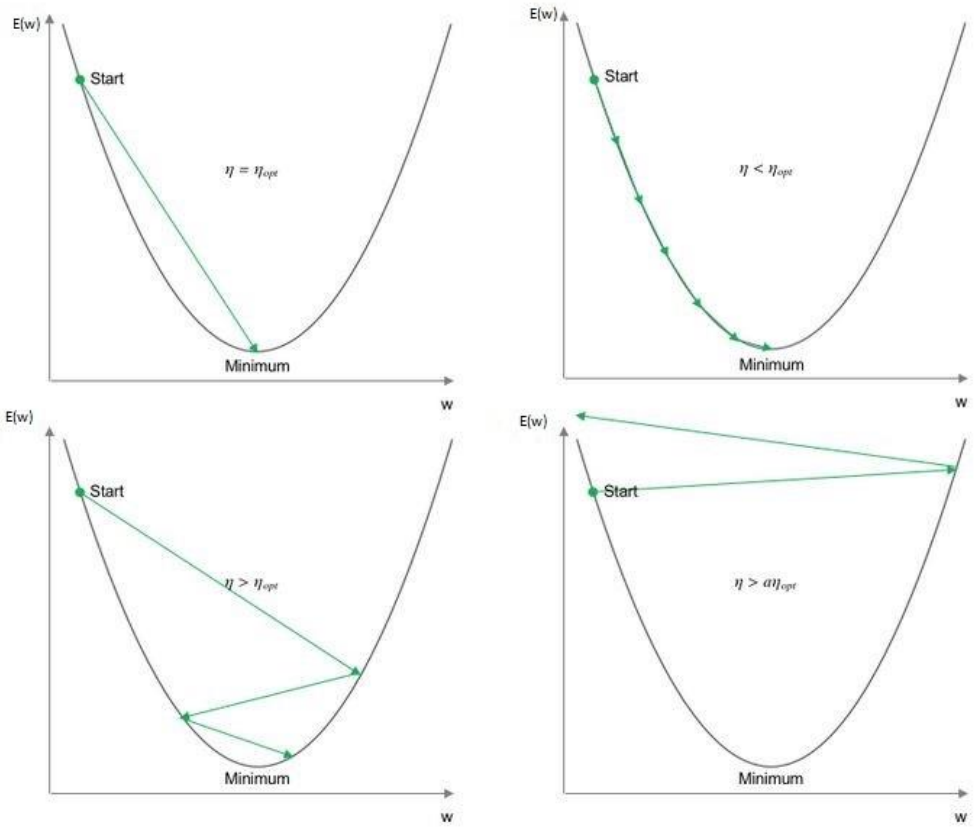

Abbildung 13 Einfluss der Lernrate auf die Kostenfunktion Heinz (2018)

Idealerweise lässt sich bei der richtigen Richtung das Minimum mit nur einer Iteration erreichen. Da die Richtung aber am Anfang des Prozesses nicht bekannt ist, muss die Lernrate niedrig ausfallen, um nicht den Pfad zu destabilisieren. Dadurch steigt jedoch die Anzahl an Iteration zum Bestimmen des Minimums der Kostenfunktion. Des Weiteren besteht die Gefahr, dass man in einem lokalen Minimum endet und das Lernen gestoppt wird. Die Lernrate wird dabei von dem Anwender selbst angegeben. Zudem ist es möglich die Lernrate adaptiv zu gestalten. Dadurch lassen sich am Anfang des Prozesses große Lernraten einsetzen, um weite Strecken zurückzulegen und im späteren Verlauf diese zu reduzieren, um sicherer zum Optimum zu gelangen. Auch die adaptive Lernrate ist in Deep Learning wie auch viele andere Parameter datenabhängig und muss experimentell ermittelt werden.

Um die richtige Richtung einzuschlagen, muss dem negativen Gradienten der Kostenfunktion gefolgt werden. Der Gradient wird durch eine partielle Ableitung der

Kostenfunktion nach dem entsprechenden Gewichtungsparameter ermittelt. Wegen der erwähnten Abhängigkeit von Neuronen der höheren und niedrigeren Schichten wird für die Ableitung der Kostenfunktion die Kettenregel eingesetzt.

Die Gleichung zum Aktualisieren der Gewichtsparameter wird wie folgt definiert:

$$w_{njt} = w_{njt-1} - \eta \frac{\partial E}{\partial w_{nj}} \qquad (19)$$

Diese Gleichung zeigt, dass die Anpassung der Gewichtseinstellung in Richtung des Minimums lediglich von dem Gewicht des Neurons der vorherigen Iteration, der partiellen Ableitung von E nach w_{nj} und der Lernrate η abhängt.

Wird das Verfahren richtig parametrisiert und angewendet, sollte sich die Kostenfunktion mit jeder Iteration verringern und letztendlich gegen null konvergieren (siehe Abb. 14).

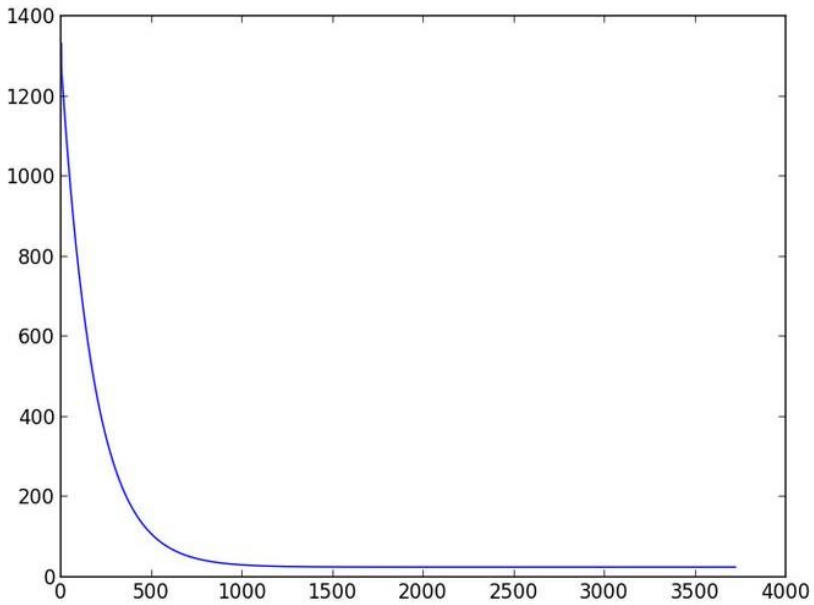

Abbildung 14 Fehler der Kostenfunktion über aller Iterationen Heinz (2018)

2.3.3 Backpropagation

Mit Hilfe des Gradient Descent kann die Kostenfunktion minimiert werden. Die dafür nötige Berechnung des Gradienten in komplexen Netzarchitekturen ist in der Praxis jedoch sehr schwierig. Grund dafür ist die Abhängigkeit jedes Neurons von der vorherigen Eingabe. Dadurch ergibt sich eine Verkettung der Wirkungen von Neuronen. Um das Problem zu lösen, verwendet man ergänzend zum Gradient Descent das Backpropagation Algorithmus. Bei diesem Verfahren wird die Ist- zu Solldifferenz, also der Wert der Kostenfunktion E über die Ausgabe- bis zur ersten verdeckten Schicht in das Netzwerk zurückgespeist. Um den Einfluss eines einzelnen Gewichts auf den resultierenden Fehler zu ermitteln und anschließend dieses Gewicht anzupassen, wird bei Backpropagation die partielle Ableitung der Kostenfunktion nach dem entsprechenden Gewichtsparameter aufgeteilt.

Um das Verfahren zu beschreiben wird die Ausgabe eines Neurons aus Schicht j in einem künstlichen Netzwerk nochmal definiert:

$$O_j = f(net_j) = f\left(\sum_n O_{j-1} w_{nj} + \theta_j\right) \qquad (20)$$

Wobei:

- w_{nj}: Gewichtparameter des Neurons der Schicht j und $j-1$
- O_{j-1}: Ausgabe des vorherigen Neurons
- θ_j: Bias des Neurons aus Schicht j, wobei diese variable auf einen konstanten Wert von 1 gesetzt wird
- f: Aktivierungsfunktion
- O_j: Ausgabe des Netzes

Die partielle Ableitung der Kostenfunktion E nach dem entsprechendem Gewichtsparameter wird durch den Einsatz der Kettenregel ermittelt:

$$\frac{\partial E}{\partial w_{nj}} = \frac{\partial E}{\partial O_j} \frac{\partial O_j}{\partial net_j} \frac{\partial net_j}{\partial w_{nj}} \qquad (21)$$

Löst man alle Faktoren der Gleichung auf, so entsteht eine neue Formel. Dabei wird berücksichtigt, ob sich das Neuron in der Ausgabeschicht oder in einer der verdeckten Schichten befindet. Die resultierende Formel wird wie folgt definiert:

$$\frac{\partial E}{\partial w_{nj}} = \delta_j O_n \quad (22)$$

Wobei:

- sich das Neuron j in der Ausgabeschicht befindet:

$$\delta_j = (O_j - O_i) f'(net_j) \quad (23)$$

- sich das Neuron j in einer verdeckten Schicht befindet:

$$\delta_j = f'(net_j) \sum_k \delta_k w_{nk} \quad (24)$$

Berücksichtigt man nun die Lernrate wie in Gleichung 19 und stellt die Gleichung nach der Änderung des Gewichtes um, so ergibt sich folgende Form:

$$\Delta w_{nj} = -\eta \frac{\partial E}{\partial w_{nj}} = -\eta\, \delta_j O_n \quad (25)$$

Wobei:

- Δw_{nj}: Änderung des Gewichtes w_{nj}
- η: Lernrate
- δ_j: Fehlersignal des Neurons j
- O_j: Ausgabe des Neurons j
- O_i: Soll-Ausgabe
- k: Index des nachfolgenden Neurons

Die Berechnung der Gradienten muss hierbei von der Ausgabeschicht bis zu den ersten verdeckten Schichten stattfinden, um diese anschließend iterativ in den

vorhergehenden Layer zu propagieren (siehe Abb. 15). Mit Hilfe des Fehlersignals δ_j der Ausgabe- und der verdeckten Schichten lassen sich nun alle Gradienten im Netzwerk berechnen. Anschließend wird die Aktualisierung der Gewichte mittels Gradient Descent durchgeführt und die Kostenfunktion reduziert.

$$\frac{\partial E_{total}}{\partial w_1} = \frac{\partial E_{total}}{\partial out_{h1}} * \frac{\partial out_{h1}}{\partial net_{h1}} * \frac{\partial net_{h1}}{\partial w_1}$$

$$\frac{\partial E_{total}}{\partial out_{h1}} = \frac{\partial E_{o1}}{\partial out_{h1}} + \frac{\partial E_{o2}}{\partial out_{h1}}$$

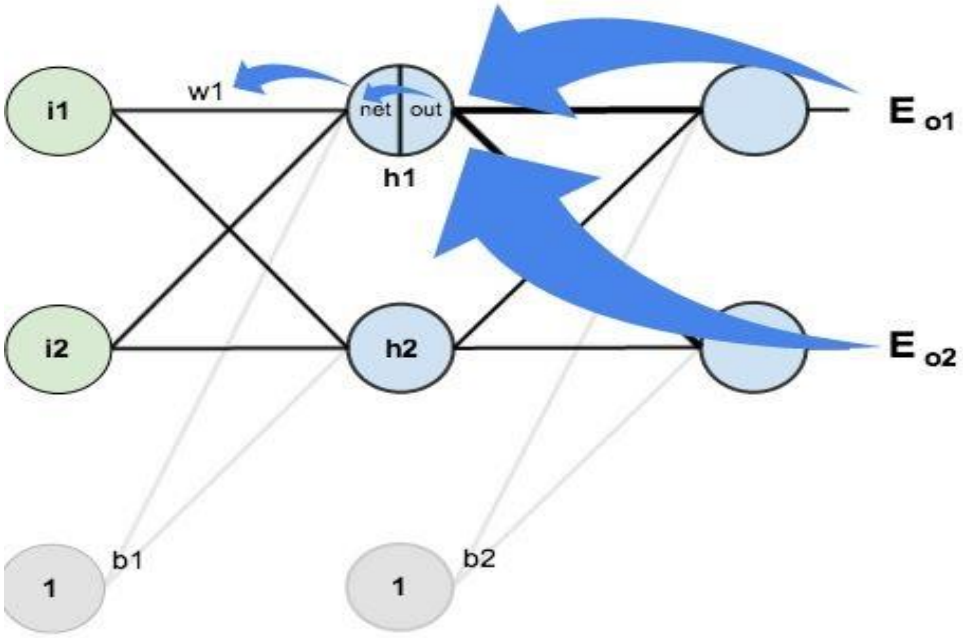

Abbildung 15 Visualisierung des Backpropagation-Verfahrens Mazur (2015)

2.4 ROS

Robot Operating System (ROS) ist ein Framework zur Entwicklung von Robotern. ROS bietet eine Abstraktionsebene der Hardware, sodass Entwickler Roboteranwendungen erstellen und testen können, ohne sich um die tatsächliche Hardware kümmern zu müssen. Es beinhaltet verschiedene Anwendungen zum Visualisieren und Debuggen von Roboterdaten. Mittels des ROS-Nachrichtensystems ist es möglich verschiedene Prozesse miteinander kommunizieren zu lassen. Dies ist ebenso möglich, wenn diese Prozesse auf verschiedenen Maschinen laufen. Weitergabe der ROS-Nachrichten kann synchron wie auch asynchron sein.

Die Software in ROS ist in Form von Paketen enthalten. Das bietet eine hohe Wiederverwendbarkeit und eine gute Modularität der Software. In Kombination der ROS-Nachrichtensystems und der Hardware-Abstraktionsebene ist es möglich eine Vielzahl an Roboteranwendungen wie Navigation und Mapping zu erstellen. ROS eignet sich für die Arbeit mit allen Robotern, da die meisten Funktionen roboterunabhängig sind. Dadurch können die Funktionspakete direkt an Robotern angewendet werden, ohne den Code innerhalb des Paketes zu ändern.

Innerhalb dieser Arbeit wird die ROS-Version „Kinetic Kame“ verwendet. Diese Version zeichnet sich durch eine hohe Stabilität unter Verwendung von Ubuntu 16.04 aus.

Die Struktur von ROS lässt sich in drei Bereiche aufteilen. Diese sind das Filesystem, das Computation Graph und die ROS-Community.

2.4.1 ROS-Filesystem Level

Das Konzept der Dateisystemebene (Filesystem) enthält die Strukturierung der Daten auf einer Festplatte.

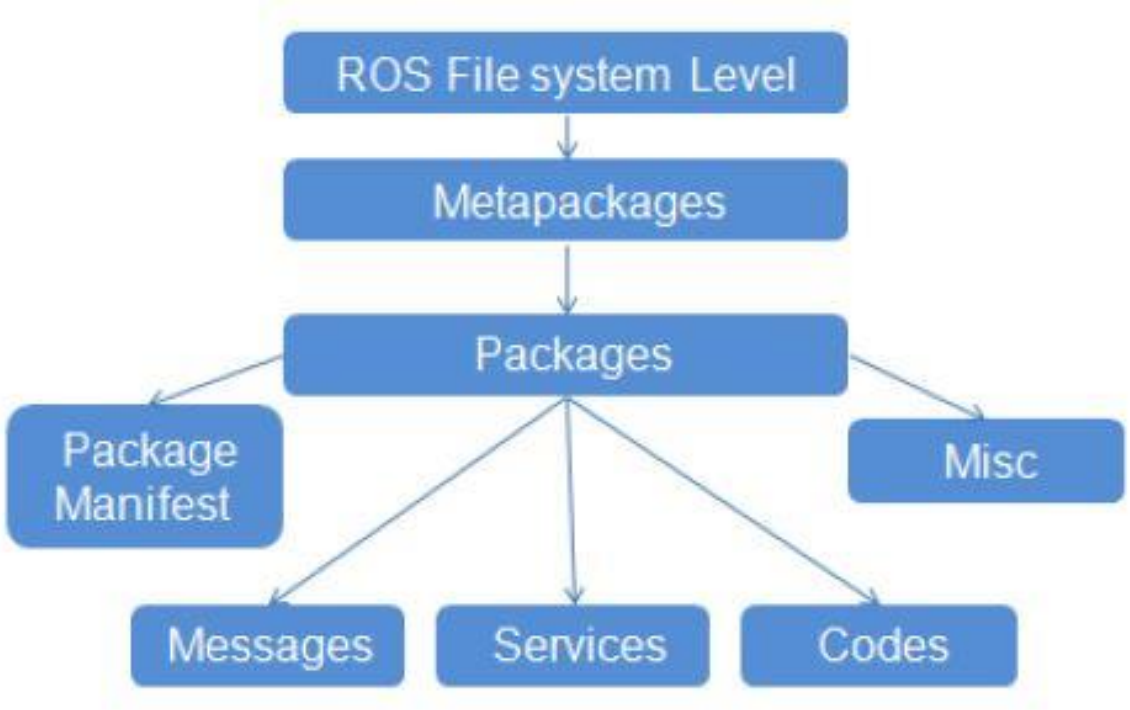

Abbildung 16 Veranschaulichung des ROS-Filesystem Levels Lentin (2017)

Metapackages: Die Metapackages sind spezielle Pakete zum Umfassen einer bestimmten Anzahl an einzelnen Paketen (Packages) für eine bestimmte Anwendung, wie z. B. die Navigation.

Packages: Die Packages (Pakete) sind die Hauptelemente für das Organisieren der Software in ROS. Sie können verschiedene Laufzeitprozesse, Bibliotheken oder Konfigurationsdateien enthalten.

Manifest: Die Package Manifest ist die sogenannte „package.xml“ Datei innerhalb eines Packages. Diese Datei enthält die Metadaten über das jeweilige Package. Diese sind der Name, die Versionsnummer, die Beschreibung, die Information über die Lizenz und weitere Informationen über mögliche exportierte Packages.

Repositories: Eine Gruppe von Packages die ein gemeinsames System der Versionsverwaltung (VCS) verwenden. Solche Pakete haben die gleiche Version und können zusammen mittels der Catkin Software freigegeben werden.

Message(msg) types: Die Kommunikation findet in ROS über die ROS-Messages statt. Die dazugehörige Konfigurationsdatei kann in “my_package/msg/MyMessageType.msg“ abgelegt werden. Diese Datei definiert die Strukturen der Nachrichten, die in ROS gesendet werden.

Service(srv) types: Service ist eines der Dienste der ROS-Computation Graph. Ähnlich zu den ROS-Messages können die Services unter „my_package/srv/service_files.srv“ definiert werden.

2.4.2 ROS-Computation Graph

ROS- Computation Graph ist ein Peer-to-Peer Netzwerk von ROS-Prozessen, in dem Daten verarbeitet werden.

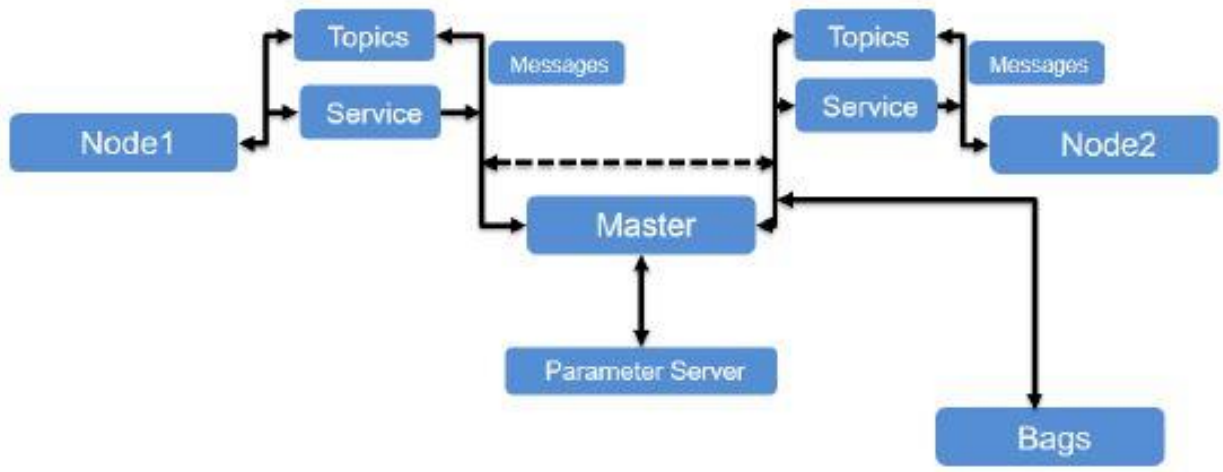

Abbildung 17 Veranschaulichung des ROS-Computation Graphs Lentin (2017)

Nodes: ROS Nodes sind Prozesse, welche von verschiedenen ROS-API Programmen zur gegenseitigen Kommunikation verwendet werden. Ein Robotersystem kann mehrere Nodes enthalten. Nodes steuern z.B. verschiedene Sensoren oder Motoren.

ROS-Nodes werden unter Verwendung von Bibliotheken wie roscpp oder rospy erstellt und benutzt.

Master: Der ROS-Master agiert als ein Zwischenknoten und verwaltet auf diese Weise die Kommunikation zwischen verschiedenen ROS-Nodes. Dieser besitzt alle nötigen Informationen zu allen Nodes in der ROS-Umgebung. Ohne den Master können sich die Nodes nicht miteinander verbinden und somit keine Nachrichten austauschen oder Dienste aufrufen.

Parameter Server: Der Parameterserver ist ein Teil des Masters und dient als ein Ablageort von Variablen, die durch Nodes oder Konfigurationsdateien angelegt werden können.

Messages: Die Kommunikation zwischen den Nodes kann auf mehrere Arten erfolgen. In allen Varianten werden die Nachrichten zwischen den Nodes in Form von ROS-Messages ausgetauscht. Es ist eine Datenstruktur mit einem Format wie z. B. Integer, Float oder String. Zudem unterstützen die ROS-Messages auch Arrays und ähnliche Formate wie der „Strukt“ in C.

Topics: Ein Topic ist das Transportsystem mittels dem die ROS-Messages zwischen zwei Nodes ausgetauscht werden. Jedes Topic besitzt einem Namen über den ein bestimmter Node die Daten veröffentlichen (publish) kann. Weitere Nodes können auf diese Weise das jeweilige Topic abonnieren (subscribe) und anschließend die Daten auslesen.

Services: Neben den Topics stellen die Services eine weitere Art der Kommunikation zwischen den Nodes dar. Bei den Topics findet die Interaktion über das Veröffentlichen und das Abonnieren der Nachrichten statt, wogegen bei den Services dies über das Anfordern oder Antworten realisiert wird. Dabei bietet ein Node ein Service unter einem Namen an, welches von anderen Nodes unter Verwendung einer Anforderungsnachricht aufgerufen werden kann.

Bags: Bags ist ein Dienstprogramm zum Ablagern und Wiedergeben von Nachrichten. Auf diese Weise können schwer erfassbare Sensordaten archiviert werden und auf anderen Computern unter anderen Bedingungen wiedergegeben werden.

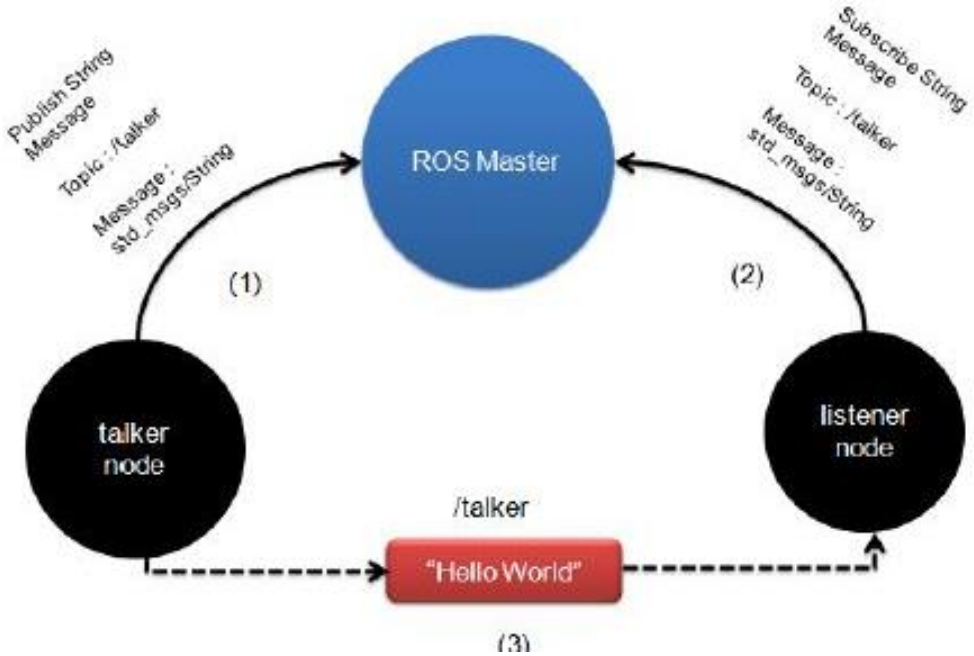

Abbildung 18 Kommunikation zwischen zwei Nodes Lentin (2017)

2.4.3 ROS-Community

Unter dem ROS-Community Bereich versteht man ROS-Ressourcen, mit denen verschiedene Communities Programme und Wissen austauschen können.

Distributions: ROS Distributions sind versionierte Sammlungen von ROS Paketen.

Repositories: Ein Repository ist ein Dateiverzeichnis in einem zentralen Versionsverwaltungssystem wie Git, SVN und Merscurial. Mit Hilfe der Repositories können Entwickler aus der ganzen Welt zu den ROS-Paketen beitragen.

ROS Wiki: Die Seite des ROS-Wiki „http://wiki.ros.org“ ist das Hauptforum von Dokumentation von Informationen zur ROS Paketen.

Mailing List: Die ROS-Mailingliste ist der primäre Kommunikationskanal für neue Aktualisierungen von ROS.

2.5 Gazebo

Gazebo ist eine in ROS integrierte, als auch eigenständige Open Source 3D Simulationssoftware. Gazebo bietet eine dynamische Simulation der Roboter mit einer großen bereits vorhandenen Bibliothek an fertigen Modellen. Es bietet zudem Unterstützung von einer Vielzahl an Sensoren, die in der Simulationsumgebung eingesetzt werden können. Das Zugreifen auf die Sensorwerte kann bereits über ROS-Elemente wie Topics und Services realisiert werden. Weitere Funktionalitäten können über verschiedene Plugins hinzugefügt werden. Diese können auch selbst für eigene Anforderungen erstellt werden.

Innerhalb dieser Arbeit wird die Gazebo Version 7 verwendet. Diese ist in ROS völlig integriert und ist mit ROS Version „Kinetic Kame“ eine der stabilsten Kombinationen. Als Physik-Engine wird „ODE“ verwendet, die auf Anwendungen in der Robotik spezialisiert ist.

2.6 Zusatzpakete

Die folgenden Unterkapitel beschreiben die wichtigsten Frameworks zur Erstellung der Reinforcement Learning Umgebung.

2.6.1 OpenAI Gym

OpenAI Gym ist eine Sammlung an Funktionen, die zum Entwickeln und Vergleichen von Reinforcement Algorithmen eingesetzt werden. Es löst ein lang vorhandenes Problem beim Erstellen von RL-Umgebungen. Durch das Standardisieren von Konzepten erhöht sich die Reproduzierbarkeit im Bereich der künstlichen Intelligenz. Ziel der Entwicklung des OpenAI Gym war es Umgebungen zu schaffen, in welchen die Entwickler ihre Lernalgorithmen testen und die Ergebnisse vergleichen können. Die Umgebung, also das eigentliche Problem wie z. B. ein Spiel ist dabei von dem Lernalgorithmus unabhängig.

OpenAI Gym ist eine in Python erstellte Open Source Bibliothek. Diese beinhaltet mehrere bereits vordefinierte Umgebungen und Agenten, die verschiedene Aufgaben

erlernen können. Es bietet somit einige Grundlagen, womit neue Umgebungen implementiert werden können. So wird eine Reihe an Funktionen vorgegeben, die zwar für neue Aufgaben angepasst werden müssen, aber dennoch ein gemeinsames Konzept teilen. Dabei werden folgende Funktionen präsentiert:

- Reset: Setzt die Umgebung auf ihren Normalzustand zurück und liefert den ersten Zustandsvektor der Umgebung.
- Step: Über diese Funktion interagiert der Agent mit der Umgebung. Dabei übermittelt der Agent eine Aktion an die Umgebung und startet somit den Prozess der Durchführung der Aktion sowie die Beobachtung und Bewertung des neuen Zustandes.
- Observe: Diese Funktion beobachtet die Umgebung und erstellt anhand dessen einen Zustandsvektor.
- Reward: Bewertet den neuen Zustand durch ausgeführte Aktion nach aufgestellten Kriterien.
- Done: Detektiert einen terminierenden Zustand oder das Lösen der Aufgabe.
- Info: Hilfsobjekt zum Debuggen.

Innerhalb dieser Arbeit werden diese Funktionen in Form von einer in Python erstellten Klasse implementiert und für unsere Aufgabe angepasst.

2.6.2 Open_ai ROS

Open_ai ROS ist ein Zusatzpaket von den Herstellern des Robot Development Studios (RDS) und bietet weitere RL-Umgebungen, die konkret in Gazebo erstellt wurden. Dieses Zusatzpaket unterteilt die RL-Trainingsumgebung in drei folgenden Bereichen:

- Aufgaben Umgebung: Dies ist eine Klasse oder Reihe an Funktionen, mit denen man die zu lernende Aufgabe des Agenten beschreiben kann.
- Roboter Umgebung: Dies sind Funktionen, die das Robotermodell beschreiben.
- Gazebo Umgebung: Dies sind Funktionen zum Verbinden mit der Simulationsumgebung. Beinhaltet die Programme gazebo_connection.py und controllers_connection.py.

Diese Unterteilung bietet den Vorteil einzelne Teile der Umgebung ändern zu können, ohne dass andere davon betroffen sind. So lässt sich einfach der Lernalgorithmus ändern und als einzelne Funktion in die gesamte Umgebung implementieren.

Wie erwähnt wird die Gazebo Umgebungsklasse dazu verwendet, um die Trainingsumgebung mit dem Gazebo Simulationsprogramm zu verbinden. Es beinhaltet spezifische Funktionen und ROS-Services zum zurücksetzen der gesamten Gazebo-Simulationsumgebung und der Controller des Robotermodells.

Die Roboter-Umgebungsklasse beinhaltet alle Roboterspezifischen Funktionen und Services. Diese erlauben das Steuern des Roboters mittels dem in Python erstellten Code. Es prüft zudem alle definierten Topics und ROS-Services auf ihre Einsatzbereitschaft.

Die Aufgaben-Umgebungsklasse definiert die Aufgabe des Agenten und beinhaltet folgende bereits vorhandene OpenAI Gym sowie weitere Funktionen:

- Step:
- Observe
- Reward
- Done
- init_env_variables: Initialisiert alle nötigen Variablen und Einstellungen.
- set_init_pose: Stellt eine Startpose des Roboters ein.

Die aufgezählten Funktionen sind eine Reihe an Standartfunktionen, welche auf die Aufgabe dieser Arbeit angepasst und durch weitere aufgabenspezifische Funktionen erweitert werden.

2.6.3 Tensorflow

Tensorflow ist ein Open Source Machine Learning Framework von Google. In Rahmen dieses Projektes wird dieses Framework zum Erstellen des neuronalen Netzwerkes verwendet. Tensorflow unterstützt die Programmiersprache Python. Aufgrund seiner flexiblen Architektur ist es möglich zur Berechnung und Erstellung der Netze neben der CPU eine GPU einzusetzen. Die trainierten Modelle lassen sich anschließend auf leistungsschwächere Plattformen wie mobile Geräte übertragen. Das Erstellen von neuronalen Netzen wurde hierbei sehr modular gestaltet. Dadurch lassen sich die neuronalen Netze aus einer Reihe verfügbaren Schichtarten und verschiedenen Kostenfunktionen zusammenstellen.

Tensorflow ist aufgrund seiner guten Dokumentation und vereinfachten Bedienung eins der beliebtesten Frameworks im Bereich Deep Learning. Die vielen Vorteile machen dieses Framework auch zum Einsatz für RL effektiv.

2.6.4 Keras

Keras ist eine in Python erstellte High-Level API für neuronale Netzwerke. Es bietet vereinfachte Schnittstellen zu Deep Learning Frameworks wie Tensorflow, Theano und CNTK an. Das Ziel von Keras ist ein schnelles Experimentieren mit neuronalen Netzen zu ermöglichen und mit möglichst geringer Verzögerung von der Idee zum Ergebnis zu gelangen. Es vereinfacht das Erstellen von Modellen, deren Parametrisierung und Auswertung.

2.6.5 Virtualenv

Virtualenv ist eine Software zum Abkoppeln von Python-Umgebungen. Das erlaubt eine virtuelle Umgebung zu erstellen, um installierte Pakete vom Rest des Systems zu isolieren. So wird auch für dieses Projekt eine eigene Python-Umgebung namens „venv“ erstellt. Dazu werden folgende zusätzliche Pakete innerhalb dieser Umgebung installiert:Tensorflow, Rospkg, Catkin_pkg, Exception, Empy, Numpy, OpenAI Gym

3 Umsetzung

Das folgende Kapitel behandelt die Entwicklung der RL-Umgebung mit anschließendem Erstellen und Trainieren des Netzwerkes. Als Erstes folgt der Aufbau des Projektes und die Beschreibung der wichtigsten Komponenten. Als Nächstes wird der Vierbein-Roboter modelliert. Dabei wird der Kriechgang als Bewegungsart untersucht und für das Projekt angepasst. Weiterhin wird anhand einer exemplarischen Konstruktion des Roboters der Aktions- und Bewegungsraum erstellt. Des Weiteren werden zwei Projektpakete für ROS aufgebaut. Das erste Projektpaket beschreibt das Modell des Roboters und die Simulationsumgebung. Dabei wird die Konvertierung von CAD zum Xacro-Format und das Aufsetzen der Simulation beschrieben. Im zweiten Projektpaket wird die RL-Umgebung definiert. Es werden der Zweck und die Implementierung alle programmierten Funktionen erläutert. Abschließend erfolgt die Erstellung des neuronalen Netzwerkes für den DQN-Algorithmus und deren Training.

3.1 Aufbau der Umgebung

Das gesamte Projekt besteht aus drei miteinander agierenden Komponenten. Der Lernalgorithmus und die Lernumgebung werden innerhalb des Projektpakets „stepper_training“ modelliert. Die Gazebo-Simulationsumgebung wird dabei im Projektpaket „stepper_mini“ erstellt. Abbildung 19 beschreibt die übergeordneten Bestandteile des Codes.

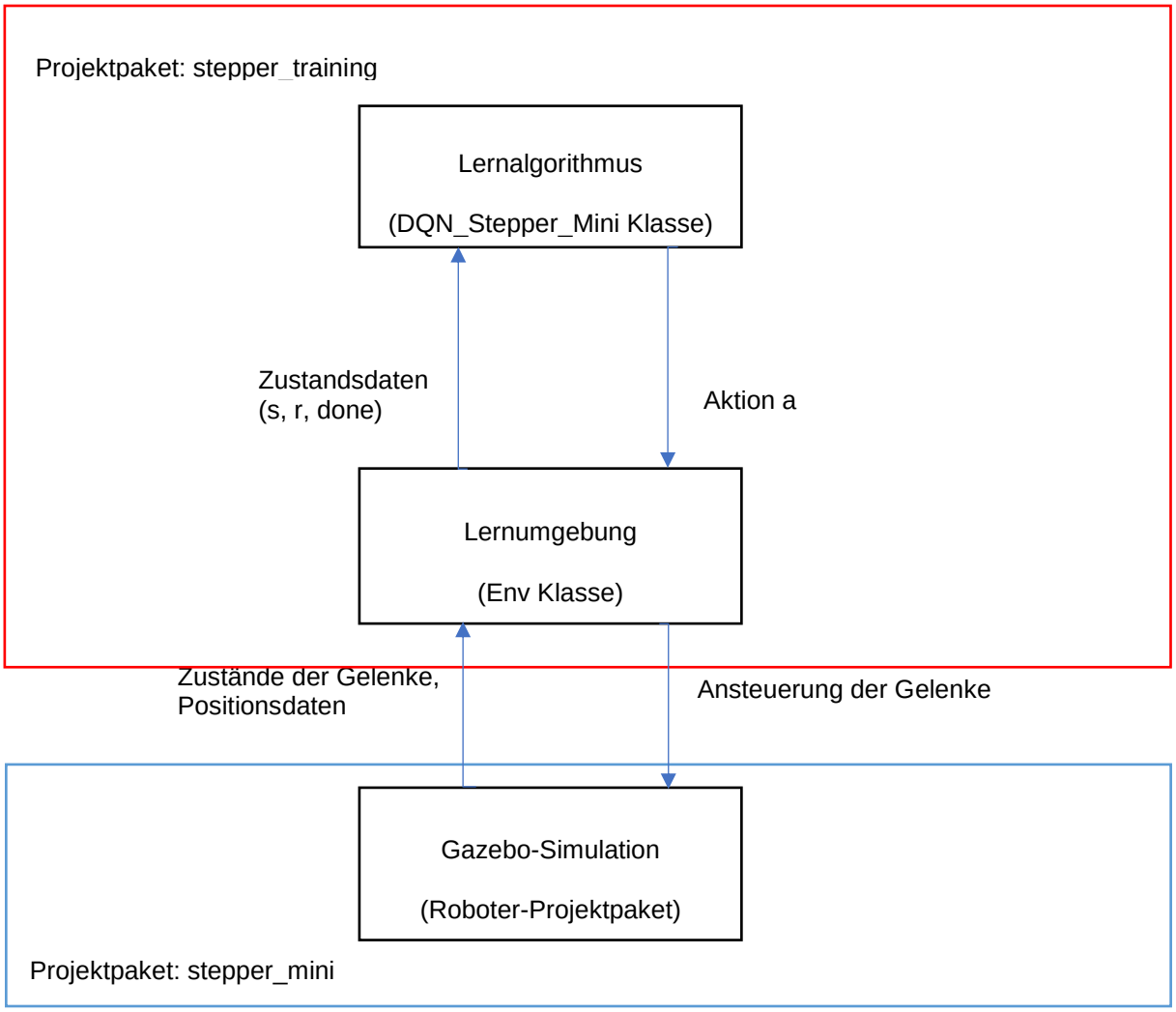

Abbildung 19 Architektur des Projektes

- Der Lernalgorithmus ist für das Lernen des Agenten zuständig. Dazu gehört das Erstellen des Target- und DQN-Netzwerkes, Berechnung des Q-Wertes, sowie sammeln und abspeichern der Lerndaten und des Modells. Es fragt schrittweise die Zustandsdaten der RL-Umgebung ab. Anhand dessen wird die nächste Aktion bestimmt, die wiederum an die RL-Umgebung übermittelt wird.
- Die Lernumgebung besteht aus der Env-Klasse, die zum Beschreiben der RL-Umgebung benutzt wird. Diese Klasse beschreibt neben der Roboter-Umgebung auch die Aufgabenumgebung. Dazu gehören alle Funktionen die zum Ausführen und bewerten des resultierenden Zustandes notwendig sind.

- Gazebo-Simulation beinhaltet das Roboter-Projektpaket. Dazu gehören das Modell des Roboters, weitere Umgebungsobjekte und die Config-, Launch und Worlddatei. Während der laufenden Simulation werden mittels Nodes und Topics die Positionen der Gelenke übergeben sowie die Daten des Odometers und der neuen Gelenk-Positionen abgefragt.

Die Kommunikation zwischen der in Python erstellten Lernumgebung und der Gazebo-Simulationsumgebung erfolgt mit Hilfe der ROS-Nodes und Topics. Die Implementierung der Nodes erfolgt innerhalb der Launchdateien und wird in Kapitel 3.3.4 näher erklärt.

3.2 Modellierung des Roboters

In diesem Kapitel wird das Modell des Vierbein-Roboters definiert und erstellt. Dazu erfolgt zunächst die Analyse des Kriechgangs. Anschließend folgt die Konstruktion des Roboters. Darauf aufbauend wird der Bewegungsraum und das Bewegungsmuster der resultierenden Bewegungsart definiert. Abschließend werden die Aktionen und die dazugehörige Bewegungsabfolge der einzelnen Beinelemente festgelegt.

Zur Fortbewegung wird eine vereinfachte Variante des Kriechgangs verwendet. Aus der Analyse des vollen Kriechgangs ergibt sich die Anzahl an benötigten Gelenken und wie diese angesteuert werden müssen, um eine Aktion auszuführen. Bei dieser Art der Fortbewegung wird zu einem Zeitpunkt nur ein Bein bewegt, die restlichen bilden ein Stützpolygon (siehe Abb. 20). Der Schwerpunkt des Roboters darf das Stützpolygon zu keiner Zeit verlassen. Verlässt der Schwerpunkt das Stützpolygon, wird der Roboter umkippen, sofern dieser nicht zeitlich durch das Absetzen des vierten Beines stabilisiert wird.

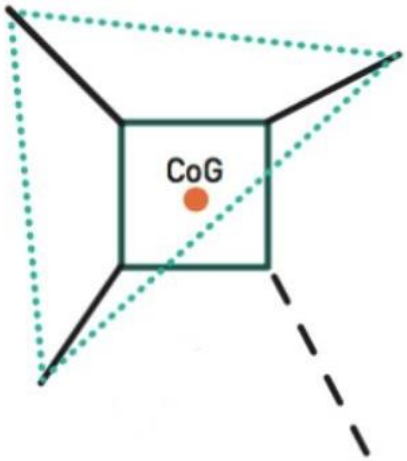

Abbildung 20 Resultierendes Stützpolygon Josh (2016)

Viele Roboter basieren auf dieser Fortbewegungsweise. Sie können alle unterschiedlich komplex sein, dennoch folgen sie alle den gleichen Grundregeln.

1. Die Bewegungen beginnen alle aus einem stationären Zustand mit stets gleicher Geschwindigkeit in gerader Linie über eine horizontale Auflagefläche.

2. Ein typischer Vierbeinroboter ist in Längs- und Querrichtung symmetrisch. Dies reduziert die Komplexität der Mechanik und des Designs.

3. Die Masse des Körpers ist um ein Vielfaches größer ist als die der Beinglieder. Deshalb wird der Schwerpunkt in der geometrischen Mitte des Roboters angenommen.

Das am weitesten verbreitete Beispiel eines Vierbein-Roboters für den Einsatz des Kriechgangs kann Abbildung 21 entnommen werden. Aus diesem Beispiel kann eine exemplarische Konstruktion mittels einem CAD-Programm entworfen werden. Es wird zunächst nur ein Bein konstruiert und anschließend vervielfacht. Bei der Konstruktion ist dabei zu beachten, dass die Achsen des Koordinatensystems mit den gewünschten Drehachsen übereinstimmen, den diese werden innerhalb der STL-Datei abgespeichert und müssen mit der Definition des Simulationsmodells übereinstimmen. Zusätzlich werden die Ursprungskoordinatensysteme jedes einzelnen Bauteiles visualisiert.

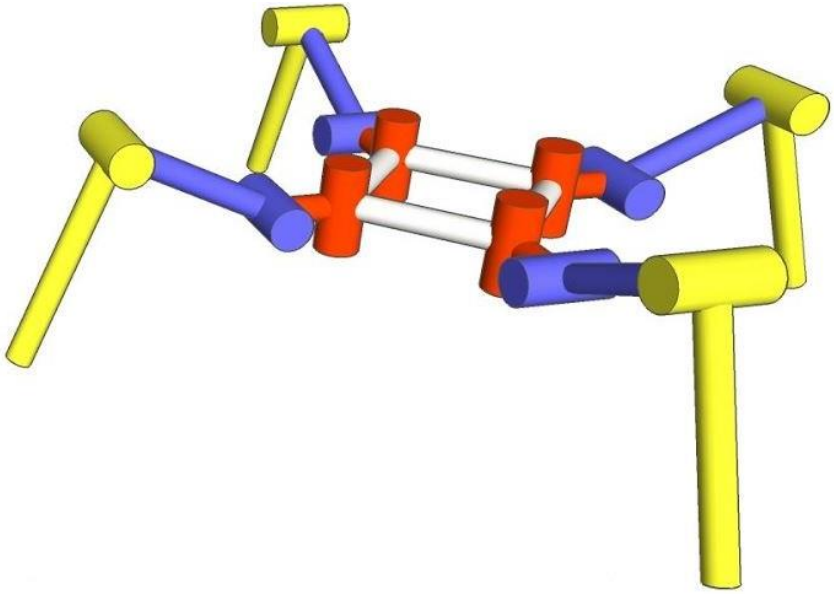

Abbildung 21 Beispiel eines Vierbein-Roboters Ritesh (2018)

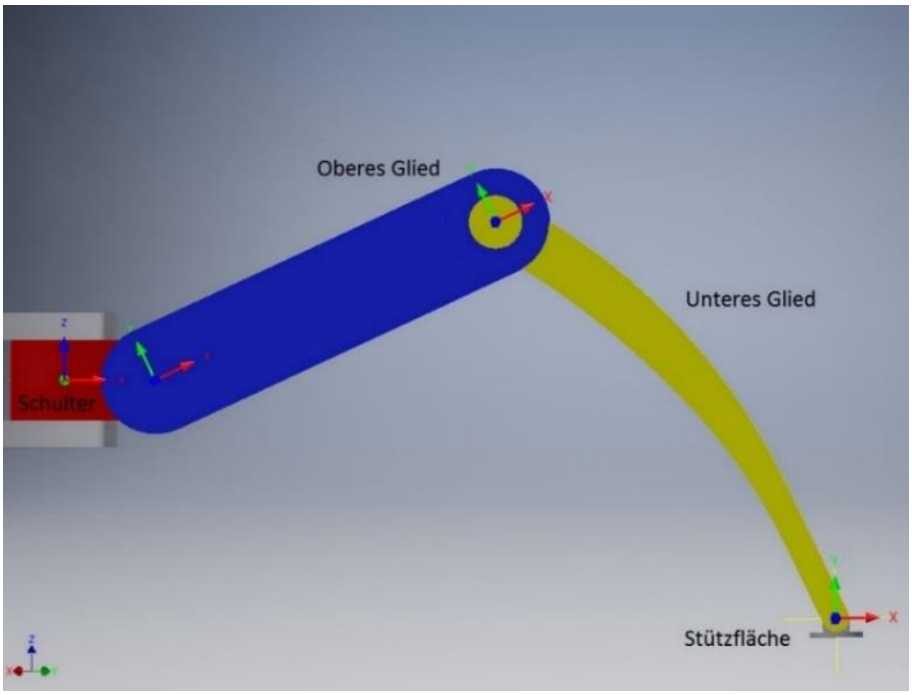

Abbildung 22 Exemplarische Konstruktion eines Beines

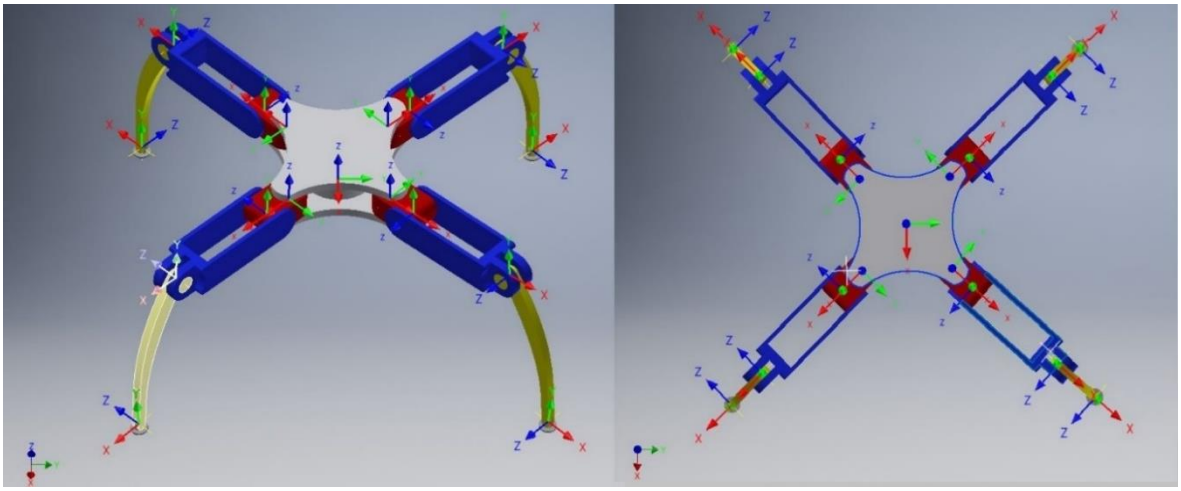

Abbildung 23 Ansicht der Roboterkonstruktion

Anhand der Konstruktion des Roboters und der visualisierten Ursprungskoordinatensysteme wird eine Denavit-Hartenberg-Transformationstabelle erstellt. Die Erstellung erfolgt jeweils vom Körperteil (weiß) bis zu jedem Schulterbauteil (rot). Da die Beinkonstruktion für alle Beine identisch ist, werden die Transformationen vom Schultergelenk bis zur Stützfläche nur einmal beschrieben. Die Verschiebung wird dabei in Metern und zyx-Notation sowie die Verdrehung in Rad angegeben.

Verbindung	z	y	x	ϕ	θ	ψ
Körper-Schulter 1	0.0125	-0.0275	0.0275	0	0	-0.8
Körper-Schulter 2	0.0125	-0.0275	-0.0275	0	0	-2.37
Körper-Schulter 3	0.0125	0.0275	-0.0275	0	0	2.37
Körper-Schulter 4	0.0125	0.0275	0.0275	0	0	0.8
Schulter - Oberes Glied	0	0	0.017	1.57	-0.3	0
Oberes Glied - Unteres Glied	0	0	0.7	0	0	0
Unteres Glied - Stützfläche	0	-0.1	0.025	1.57	0	-0.3

Tabelle 1 DH-Transformationstabelle des Robotermodells

3.2.1 Bewegungsraum

Mit der Erstellung eines Roboters kann nun der Bewegungsraum der einzelnen Beine definiert werden (siehe Abb. 24). Die roten Linien stellen dabei die Startposition der Beinglieder dar. Diese Position wird als „ausgestreckt“ definiert. Die graue Linie zeigt die zweite Position an, welche als Position beim Ausführen eines Rückwärtsschritts des Beines oder Verschiebungsschritts des Körpers definiert wird. Beim Bein gilt diese Position als eingezogen. Der Abstand zwischen diesen beiden Positionen beträgt 0.8 Rad, was ca. 45.8 Grad entsprechen würde. Die blauen Linien zeigen die maximalen Grenzen der Schritte nach vorne (positive Drehrichtung) bzw. zurück plus den Abstand zum Einleiten der Wendung des Körpers aus den maximalen Positionen. Die ganze Bogenlänge zwischen den blauen Grenzen dient im späteren Verlauf als Begrenzung beim Modellieren des Modells in ROS und Gazebo. Der maximale Bewegungsraum der Oberen Beinglieder wird auf + 0.5 Rad und -0.5 Rad hinsichtlich ihrer Startposition begrenzt. Das Folgegelenk zum unteren Beinelement sowie zu der Stützfläche wird dabei festgesetzt und kann somit nicht beweget werden. Die Startpositionen der Gelenke werden bereits innerhalb der DH-Tabelle festgesetzt.

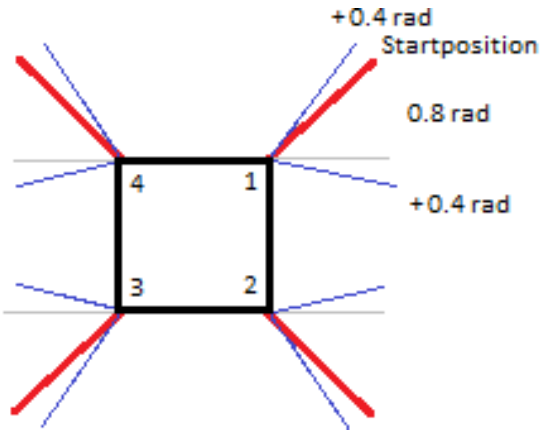

Abbildung 24 Grenzen des Bewegungsraumes der einzelnen Beine

Die Winkelpositionen der Beinglieder werden dabei alle in Rad angegeben. Dies ist die standardisierte Einheitengröße in ROS. Die Aufrundung der Winkelposition auf eine Nachkommastelle hilft beim Trainieren des DQN-Netzwerkes, da das Training innerhalb einer physikalischen Simulation stattfindet, haben die Aktuatoren stets eine Streuung bei Einnehmen der Positionen. Die 0.8 Rad Winkelposition können

gleichzeitig dazu führen, dass der Schwerpunkt bei einigen Anordnungen der Stützbeine das Stützpolygon verlässt. Als Abhilfe dazu werden an die Spitzen des letzten Gliedes der Beine rundförmige Stützflächen hinzugefügt, um das Stützpolygon zu erweitern. Dies führt auch dazu, dass unter geeigneter Anordnung der Beine der Körper von nur zwei Stützbeinen gehalten werden kann.

3.2.2 Bewegungsmuster

Das Bewegungsmuster unter Verwendung des ausführlichen Kriechgangs wird anhand Abbildung 25 verdeutlicht. Jedes Bein hat dabei drei Gelenke und besteht aus drei Gliedern. Die Zahlen geben dabei die Nummer des jeweiligen Beines an.

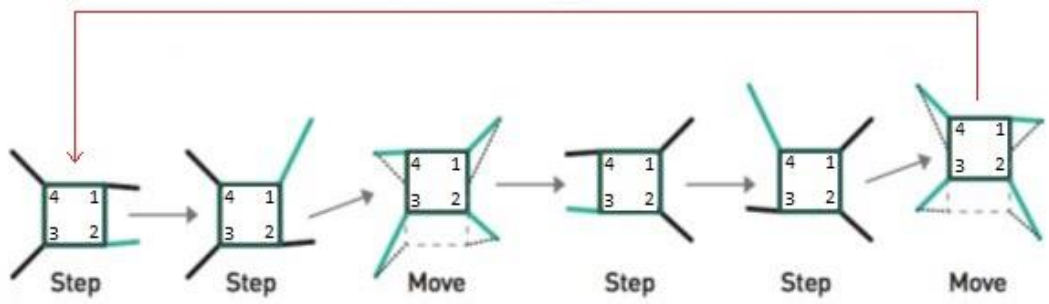

Abbildung 25 Bewegungsmuster des Kriechgangs Ritesh (2018)

Im Folgenden werden die einzelnen Bewegungsschritte der Abbildung 25 erklärt.

- Die Abfolge dieser Bewegungsart läuft von links nach rechts ab. Beginnend mit der ersten linken Position startet der Roboter seinen Bewegungsprozess. Dabei sind zwei Beine ausgestreckt und zwei gegenüberliegenden eingezogen.
- Bein eins wird angehoben und ausgestreckt. Dabei wird auch das letzte Beinglied ausgestreckt.
- Bein zwei und vier Verschieben sich um 45 Grad nach hinten. Die letzten Gelenke der Beine eins und drei werden wieder in ihre Normalposition eingezogen. Körper bewegt sich nach vorne.
- Bein 3 wird angehoben und ausgestreckt.
- Bein vier wird angehoben und ausgestreckt. Letztes Beinglied wird ebenfalls ausgestreckt.

- Beine eins und drei drehen sich aus dem Stand nach hinten und rufen somit eine Verschiebung des Körpers nach vorne hervor.
- Bein zwei wird angehoben und ausgestreckt. Letztes Beinglied wird ebenfalls ausgestreckt. Die Sequenz beginnt von vorne.

Um den Zustandsraum, welcher als Input für das neuronale Netzwerk dient und somit den Lernprozess zu vereinfachen wird das Einziehen und Ausstrecken des unteren Beingliedes (siehe Abb. 22 gelbes Beinelement) ausgelassen. Die Bewegung des Körpers wird lediglich durch das Zurückdrehen des jeweiligen Beinpaars 1/3 oder 2/4 hervorgerufen.

Durch diese angepasste Bewegungsart ergibt sich ein neues Bewegungsmuster. Berücksichtigt man die Tatsache, dass der Roboter in einer vordefinierten Startposition startet, ergeben sich zwei Möglichkeiten wie die Anfangsbewegung theoretisch ablaufen kann. Die erste richtige Aktion wäre in dem Fall der Vorwärtsschritt des zweiten oder des dritten Beines. Danach kann aus dieser Lage eines der Verschiebungsaktionen durchgeführt werden. Nach der Verschiebungsaktion müssen wieder mindestens zwei Beine bewegt werden, um eine weitere Verschiebungsaktion richtig ausführen zu können. Durch diese neue Bewegungsart ist es möglich, dass der Agent sich nur auf eine Verschiebungsaktion fixiert. Dies führt dazu, dass zwei der vier Beine bei der Verschiebung und Umstellen der Beine nicht eingesetzt werden, sondern lediglich als Stützen eingesetzt werden. Im Kapitel 3.4.3 wird durch ein passendes Belohnungssystem versucht diesen Effekt auszugleichen.

3.2.3 Beschreibung der Aktionen

In diesem Unterkapitel werden die einzelnen Aktionen des Agenten definiert. Diese sollen den nötigen Bewegungsbereich zum Erfüllen der RL-Aufgabe abdecken. Für die einzelnen Aktionen werden Bewegungsabfolgen erstellt, welche im späteren Verlauf der Arbeit in Form von Programmfunktionen realisiert werden.

Um eine Aktion wie die Verschiebung des Körpers auszuführen, werden die Gelenke des Roboters angesteuert. Das Bewegen der Gelenke wird über eine Eingabe einer

Position von π bis $-\pi$ hinsichtlich der Achsenausrichtung des Schulterbauteils erfolgen. Die einzelnen Gelenke werden sowohl einzeln nacheinander als auch gleichzeitig in Gruppen bewegt. Gedreht wird dabei um die z-Achse mit Beachtung der Rechte-Hand-Regel. Zunächst werden alle Aktionen aufgelistet und nummeriert.

Der Aktionsbereich des Roboters begrenzt sich auf 12 Aktionen.

	Ausstrecken	Einziehen
Bein 1	0	1
Bein 2	2	3
Bein 3	4	5
Bein 4	6	7

Tabelle 2 Aktionstabelle Teil.1

	Verschieben des Körpers	Drehen des Körpers
Beinpaar 2/4	8	-
Beinpaar 1/3	9	-
Rechts	-	10
Links	-	11

Tabelle 3 Aktionstabelle Teil. 2

Anhand der Konstruktion und des Bewegungsraumes der einzelnen Schulterbauteile und der Oberen Beinglieder können die Abläufe der einzelnen Beinelemente beim Ausführen einer Aktion definiert werden. Als Erstes werden das Ausstrecken und das Einziehen des Beines definiert. Diese beiden Aktionen werden in einer einzigen Funktion implementiert, wobei das Vorzeichen bei der einzunehmenden Winkelposition der Funktion übergeben wird.

- Anheben des oberen Beinelementes des jeweiligen Beines um 0.3 Rad.
- Verdrehen des Schulterbauteils je nach Bein um $\pm$ 0.8 Rad
- Wartepause von t Sekunden
- Absetzen des oberen Beinelementes wieder auf 0 Rad Position

Die Verschiebung des Körpers wird paarweise ausgeführt, wobei je nach Positionen der Beine die Beinpaare 1 und 3 oder 2 und 4 eingesetzt werden. Die Bewegung des Beinpaares 2/4 läuft folgend ab:

- Anheben der oberen Beinelemente des Beinpaars 1/3 um 0.1 Rad.
- Wartepause von t Sekunden
- Einziehen der Beine 2/4 durch Verdrehen der Schultergelenke um -0.8 Rad beim Bein 2 und +0.8 Rad bei Bein 4.
- Wartepause von t Sekunden
- Absetzen der oberen Beinelemente des Beinpaars 1/3 auf die 0 Rad Position.

Analog wird das Verschieben des Körpers mittels des Beinpaars 1/3 durchgeführt.

- Anheben der oberen Beinelemente des Beinpaars 2/4 um 0.1 Rad.
- Einziehen des Beinpaares 1/3 durch Verdrehen der Schultergelenke um -0.8 Rad beim Bein 1 und +0.8 Rad bei Bein 3.
- Wartepause von t Sekunden
- Absetzen der oberen Beinelemente des Beinpaars 2/4 auf die 0 Rad Position.

Des Weiteren wird das Drehen des Körpers implementiert. Beim Drehen des Körpers wird die aktuelle Lage der Beine nicht berücksichtigt. Darüber hinaus wird lediglich eine Drehfunktion erstellt, wobei die Drehrichtung durch eine übergebene Variable definiert wird. Da die Aktionsabfolge fest definiert ist, ist das Drehen des Körpers nicht aus jeder Beilage vorteilhaft. Bei Erfolgreichen Training würde der Agent lernen, aus welcher Konstellation der Beine die Drehbewegung sicher durchgeführt werden kann. Der Ablauf der Drehbewegung läuft wie folgt ab:

- Anheben der oberen Beinelemente des Beinpaars 1/3 um 0.1 Rad.
- Umstellen des Beinpaars 1/3 um 0.4 Rad in ausgewählte Drehrichtung
- Wartepause von t Sekunden
- Absetzen der oberen Beinelemente des Beinpaars 1/3 auf die 0 Rad Position
- Wartepause von t Sekunden
- Anheben der oberen Beinelemente des Beinpaars 2/4 um 0.1 Rad.

- Umstellen des Beinpaars 2/4 um 0.4 Rad in ausgewählte Drehrichtung
- Wartepause von t Sekunden
- Absetzen der oberen Beinelemente des Beinpaars 2/4 auf die 0 Rad Position

Die Wartepausen sind nötig, damit die an Gazebo übergebenen Befehle nicht gleichzeitig ausgeführt werden. Die Dauer der Pausen wird so eingestellt, dass diese möglichst minimal ausfallen und zugleich den Controllern genügend Zeit geben die Bewegung auszuführen.

3.3 Erstellen des Roboter-Projektpakets

3.3.1 Modellpaket

Einzelne Projekte in ROS werden als Pakete (Packages) gehandhabt. So wird auch für dieses Projekt jeweils ein Paket für das Modell und ein weiteres für die RL-Umgebung inklusive der dazugehörigen Algorithmen erstellt. Das Erstellen der Pakete erfolgt durch Catkin Tools. Das Paket wird dabei innerhalb des Ordners *catkin_ws/src* erstellt, welches die Arbeitsumgebung darstellt. Anschließend wird in dem erstellten Paket die folgende Ordnerstruktur aufgebaut:

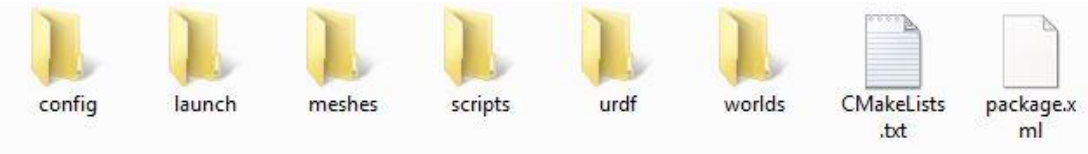

Abbildung 26 Ordnerstruktur eines ROS-Projektes

1. Config: Beinhaltet die Datei Joints.yaml und beschreibt den Controller der Joints sowie deren PID Werte. Diese Daten werden beim Aufrufen des Packages in den Parameter-Server geladen.
2. Launch: Beinhaltet die main- und world-Launchdatei. Diese sind für das Erzeugen und starten der Modelle, der Welt, diverser Plugins und der im Projekt verwendeten Nodes zuständig.

3. Meshes: Beinhaltet den Ordner CollisionSTL und VisualSTL, welche die STL Dateien der einzelnen Bauteile beinhalten.
4. Scripts: Beinhaltet Python-Code für den Odometer-Plugin, welche die XYZ-Koordinaten des Körpers im Raum liefert.
5. URDF: Beinhaltet das parametrisierte URDF-Modell des Ziels und das Xacro-Modell des Roboters.
6. Worlds: Beinhaltet die Beschreibung der eingesetzten Welt und der darin statischen Objekte. Zudem werden innerhalb der Worlddatei die Parameter der Physik-Engine sowie die physikalischen Parameter der Umgebung wie die Anziehungskraft beschrieben.
7. CMakeLists.txt und package.xml: Automatisch erstellte Metadaten mit Informationen und Einstellungen zu dem Projektpaket.

3.3.2 Erstellen des Modells

Um den Roboter innerhalb einer Simulation in Gazebo einzusetzen, muss dieser in URDF (Unified Robot Description Format) oder Xacro-Format umgewandelt werden. Zudem müssen die CAD-Daten der Konstruktion in STL-Format vorliegen. Diese sind vorzugsweise in STL binärem Format zu erstellen. Als Einheitengröße müssen Meter eingestellt werden. Des Weiteren müssen für jedes einzelne Bauteil jeweils zwei STL-Dateien erstellt werden. Dies ist notwendig, da eine Datei die Kollisionseigenschaften des Bauteils und die weitere die visuellen Eigenschaften beschreibt. Die STL der Kollisionsbox ist vorzugsweise in sehr niedriger Qualität zu erstellen, um die Physik-Engine bei der Berechnung zu entlasten. Die STL für die visuellen Eigenschaften kann in hoher Qualität erstellt werden, da dies die Leistung des Computers nur bedingt beeinträchtigt.

URDF (Universal Robotic Description Format) ist ein standardisiertes XML-Dateiformat. Dieses wird genutzt, um in ROS Elemente eines Robotermodells zu beschreiben. Dazu gehören die kinematischen und dynamischen Eigenschaften eines Modells. Das Beschreiben der Eigenschaften wird dabei in Form von Tags (Elementen) realisiert. Eine der Schwächen des URDF-Formates ist es, dass hierbei keine parallelen Kinematiken eingesetzt werden können. Zudem lässt sich schwer eine

reale Reibung zwischen Bauteilen und der Welt realisieren. Elemente außerhalb des Modells wie das Licht oder Parameter der Welt lassen sich hiermit nicht definieren und müssen durch ein zusätzliches Format beschrieben werden.

Neben URDF gibt es das sogenannte Xacro-Format, welches die gleiche Syntax verwendet. Vorteil des Xacro-Formats ist es Makros und Variablen verwenden zu können. Auf dieser Weise verringert sich drastisch der Aufwand beim Beschreiben der einzelnen Beine. Darüber hinaus ist es möglich simple mathematische Funktionen einzusetzen sowie das Verweisen und Einbinden weiterer Xacro-Dateien.

Innerhalb der URDF/Xacro existiert das“<gazebo>“ Element. Es ist eine Erweiterung zum URDF und bietet weitere Einstellmöglichkeiten für die Simulation. Im Grunde ist es ein Verweis auf die Elemente des SDF-Formats, welche im Normalfall in URDF nicht enthalten bzw. beim Starten von Gazebo automatisch mit Default Werten initialisiert werden. Es existieren drei verschiedene Typen des <gazebo> Elementes. Diese sind das „<robot>“, „<link>“ und „<joint>“ Element. Dieses Element bezieht sich nach wie vor nur auf das Modell und nicht die Welt.

Die einzelnen Gliedmaßen des Roboters gelten als sogenannte Links und werden mittels einem Link-Element beschrieben. Um den Link eine annähernd reale physikalische Bewegungsform zu verleihen, werden folgende Parameter eingestellt: Davon wurden die meisten mit Hilfe des Konstruktionsprogramms automatisch berechnet:

- Name
- Gewicht
- Abstand zum Schwerpunkt
- Ausrichtung des Trägheitsfeldes gegenüber den Achsen
- Trägheit
- Steifigkeit
- Dämpfung
- Reibkoeffizient

Eine Linkbeschreibung sieht folgendermaßen aus:

```
41  <link name="Base">
42    <visual>
43      <geometry>
44        <mesh filename="${Base_path_vis}" />
45      </geometry>
46    </visual>
47    <collision>
48      <geometry>
49        <mesh filename="${Base_path_col}" />
50      </geometry>
51    </collision>
52    <inertial>
53      <mass value="${Base_mass}" />
54      <origin rpy="0 0 0" xyz="0.0  0.0 0.0125"/>
55      <inertia ixx="0.00019" ixy="0.0" ixz="0.0" iyy="0.00019" iyz="0.0" izz="0.00031" />
56    </inertial>
57  </link>
```

Abbildung 27 Beschreibung eines Links in Xacro-Format

Über den <gazebo> Element werden weitere Parameter wie die Steifigkeit, Dämpfung und Reibkoeffizienten eingestellt.

```
59  <gazebo reference="Base">
60    <kp>1000000.0</kp>
61    <kd>1.0</kd>
62    <mu1>0.21</mu1>
63    <mu2>0.21</mu2>
64    <minDepth>0.001</minDepth>
65    <maxVel>0</maxVel>
66  </gazebo>
```

Abbildung 28 Zusätzliche Parametrisierung durch den <gazebo> Element

Um die beschriebenen Links miteinander zu verbinden, wird in URDF/Xacro-Format der Joint-Element benutzt. Dieser beschreibt den Typ der Verbindung und deren dynamischen Eigenschaften. Die dynamischen Parameter können nicht direkt abgelesen werden, sondern müssen in den meisten Fällen durch Erfahrungswerte ermittelt werden. Zum Beschreiben eines Joints werden folgende Parameter verwendet:

- Name des Joints
- Typ des Joints
- Name des Parentlinks
- Name des Childlinks
- Vierschiebung in x, y, z von vom letzten zum nächsten Frame (Koordinatensystem)
- Verdrehung in r, p, y von vom letzten zum nächsten Frame (Koordinatensystem)
- Angabe um die zu drehende Achse
- Maximale einsetzbare Kraft für diesen Joint
- Maximale Drehgeschwindigkeit für diesen Joint
- Begrenzungen der Bewegungsfreiheit wie den Drehwinkel für diesen Joint (lower, upper)
- Haft- und Gleitreibung

Eine Beschreibung eines Joints-Elementes sieht folgenderweise aus:

```
70  <joint name="Base_Shoulder_${prefix}" type="revolute">
71    <parent link="Base"/>
72    <child link="Shoulder_${prefix}" />
73    <origin rpy="${rpy}" xyz="${xyz}"  />
74    <axis xyz="0 0 1" />
75    <limit effort="${effort}" velocity="${velocity}" lower="${lower}" upper="${upper}" />
76    <dynamics damping="${damping}" friction="${friction}" />
77  </joint>
```

Abbildung 29 Beschreibung eines Joints in Xacro-Format

Neben dem Namen und dem Typen des Joints wird die Verbindung der Links durch den <parent link> und <child link> Element beschrieben. Das <origin> Element beschreibt die Transformation von dem Parent-Frame zu dem Child-Frame bezüglich der Verdrehung in Rad und Verschiebung in Metern. Der Parent-Frame bezieht sich auf den Punkt des Koordinatenursprungs. Die Achsendrehrichtung wird innerhalb des <axis> Elementes beschrieben. Begrenzungen werden im <limit> Element definiert. Begrenzungen beim Drehen werden über „lower“ und „upper“ Elementes bestimmt,

wobei dies ebenfalls in Rad angegeben wird. Die Begrenzung in „effort“ gibt die maximale einsetzbare Kraft auf das Gelenk an.

Auch für das Joint-Element kann eine <gazebo> Referenz erstellt werden. So wird hierbei die Federeigenschaft eines Joints durch einen Dämpfer minimiert.

```
79  <gazebo reference="Base_Shoulder_${prefix}">
80    <implicitSpringDamper>1</implicitSpringDamper>
81  </gazebo>
```

Da eine reale Beschreibung der physikalischen Werte in einer simulierten Umgebung meist nicht möglich bzw. sehr schwierig ist, werden für einige Parameter Pseudowerte benutzt. Dies hat den Grund, da die Engine nicht alle Einflüsse der realen Umgebung optimal beschreiben kann.

Um die Verbindung zwischen einem Joint und einem Aktuator zu beschreiben, muss für jeden Joint ein „Transmission“ Element definiert werden. Dieser sieht wie folgt aus:

```
230  <transmission name="Base_Shoulder_${prefix}_trans">
231    <type>transmission_interface/SimpleTransmission</type>
232    <joint name="Base_Shoulder_${prefix}">
233      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
234    </joint>
235    <actuator name="Base_Shoulder_${prefix}_motor">
236      <mechanicalReduction>1</mechanicalReduction>
237      <hardwareInterface>hardware_interface/EffortJointInterface</hardwareInterface>
238    </actuator>
239  </transmission>
```

Abbildung 30 Beschreibung eines Transmission-Elementes

Darüber hinaus muss innerhalb der URDF ein Gazebo Plug-In hinzugefügt werden. Dieser liest alle Transmission Elemente ein und lädt Schnittstellen für den Control Manager. Der Standardaufruf sieht wie folgt aus:

```
<gazebo>
    <plugin name="gazebo_ros_control" filename="libgazebo_ros_control.so">
     <robotNamespace>/stepper_mini</robotNamespace>
    </plugin>
</gazebo>
```

Abbildung 31 Aufruf des ROS-Control Plugins

Als letztes wird ein Odometer-Plugin implementiert. Mit dessen Hilfe können die xyz-Koordinaten eines ausgewählten Links mittels einer Node ausgegeben werden. Die dafür nötige Node wird in der Launchdatei erstellt.

```
<gazebo>
    <plugin name="p3d_base_controller" filename="libgazebo_ros_p3d.so">
        <robotNamespace>/stepper_mini</robotNamespace>
        <alwaysOn>true</alwaysOn>
        <updateRate>50.0</updateRate>
        <bodyName>Base</bodyName>
        <topicName>odom</topicName>
        <gaussianNoise>0.01</gaussianNoise>
        <frameName>world</frameName>
        <xyzOffsets>0.0 0.0 0.0</xyzOffsets>
        <rpyOffsets>0.0 0.0 0.0</rpyOffsets>
    </plugin>
</gazebo>
```

Abbildung 32 Odometer Plugin

Innerhalb der Datei „goal.urdf“ wird das Zielfeld erstellt. Es dient lediglich dazu den Zielpunkt innerhalb der Simulationsumgebung zu visualisieren. Erstellt wird dieser mittels eines Link-Elementes in Form eines flachen roten Zylinders mit einer visuellen Länge von 0.001 Metern und einem Radius von 0.025 Metern. Der Radius des Ziels wird zudem für die Belohnungsfunktion verwendet. Alle weiteren physikalischen Werte des Modells wie die Kollisionsbox oder Trägheit werden auf einen sehr geringen Wert eingestellt.

3.3.3 Erstellen der Welt

Die Erstellung der Welt in Gazebo wird innerhalb der Datei „quadruped.world“ ausgeführt. Beschrieben wird die Welt dabei im SDF-Format. Als Grundgerüst wird innerhalb der Main- und world_launcher-Launchdatei eine Standardwelt von Gazebo geladen. Es lädt eine leere Umgebung mit Default-Werten. Mittels der extra erstellten Worlddatei können einzelne physikalische Eigenschaften manuell parametrisiert und weitere Objekte der Welt hinzugefügt werden. Ähnlich wie bei der Erstellung der Modelle wird auch die Welt mittels verschiedener Elemente beschrieben. Die wichtigsten Elemente sind die world, physics und das model-Element. In dem World-Element wird die Bodenebene hinzugefügt. Zudem werden die Anziehungskräfte und das Magnetfeld der Welt parametrisiert. Wie in Abbildung 33 dargestellt, können mit Hilfe des physics-Elementes verschiedene Einstellungen bezüglich der mathematischen Berechnung der Physik-Engine angepasst werden.

```
13  <physics name='default_physics' default='0' type='ode'>
14    <ode>
15      <solver>
16        <type>quick</type>
17        <iters>100</iters>
18        <precon_iters>0</precon_iters>
19        <sor>1</sor>
20        <use_dynamic_moi_rescaling>0</use_dynamic_moi_rescaling>
21      </solver>
22      <constraints>
23        <cfm>0</cfm>
24        <erp>0.1</erp>
25        <contact_max_correcting_vel>10</contact_max_correcting_vel>
26        <contact_surface_layer>0.001</contact_surface_layer>
27      </constraints>
28    </ode>
29    <max_step_size>0.0001</max_step_size>
30    <real_time_factor>1</real_time_factor>
31    <real_time_update_rate>1000</real_time_update_rate>
32  </physics>
```

Abbildung 33 Physics-Element der Worlddatei

Des Weiteren wird für die Lernumgebung ein Spielfeld erstellt. Dieser gilt dabei als ein statisches Objekt. Dadurch werden die Trägheit und die Masse dieses Objektes nicht berücksichtigt. Die Größe des Feldes wird auf 4x4x0.05 Meter eingestellt. Weiterhin wird dieser Fläche ein sehr hoher Reibungskoeffizient sowie eine maximale Anzahl an

Kontakten vorgegeben. Abschließend werden einige visuelle Eigenschaften des Modells, Lichteffekte und die Position der Kamera eingestellt. Zu beachten ist, dass hierbei die meisten Parameter den Standardwerten gleichen, dennoch deklariert wurden, um im späteren Verlauf und weiteren Arbeiten angepasst zu werden.

Die fertige Welt inklusive des Roboters und eines beliebig gesetzten Ziels wird anhand Abbildung 34 dargestellt.

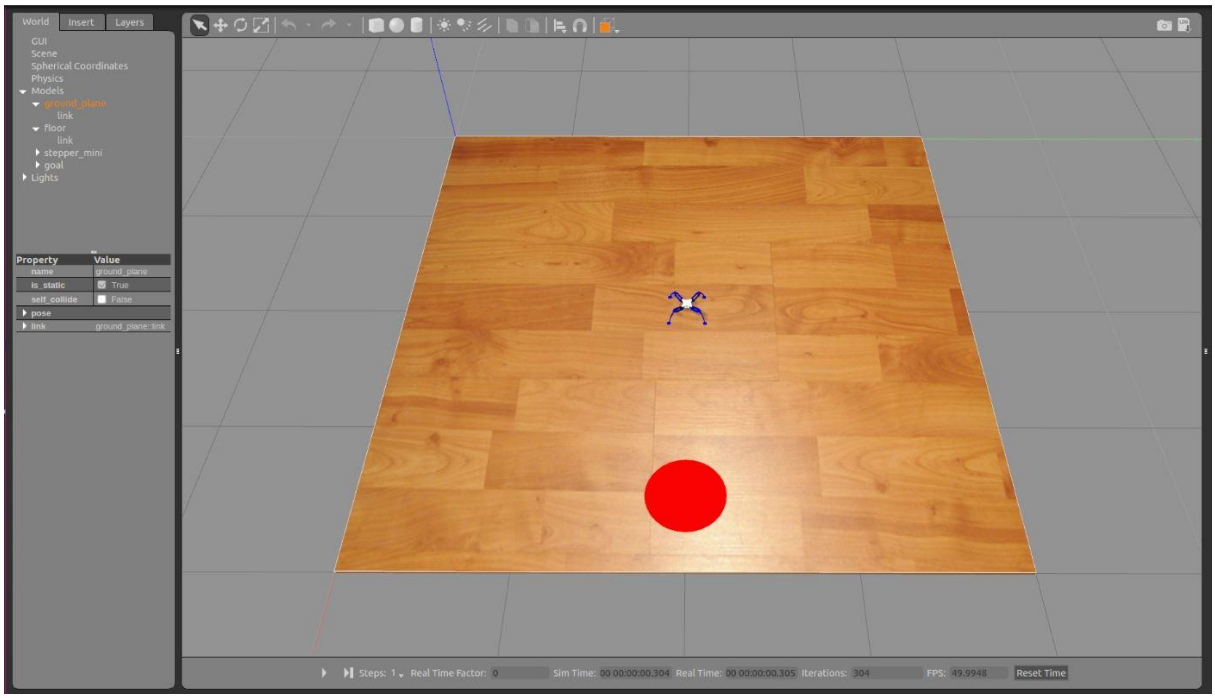

Abbildung 34 Komplette Simulationsumgebung

3.3.4 Erstellen der Launchdateien

Launchdateien gelten in ROS als Startdateien. Diese bieten eine bequeme Möglichkeit zum Starten von Nodes, des Masters und weiterer Initialisierungen wie das Laden von Konfigurationen. Das Starten der Launchdtateien wird mittels dem ROS-Konsolenbefehl *roslaunch,* gefolgt mit der Angabe des Projektpakets und dem Namen der Launchdatei durchgeführt. Launchdateien haben das Format launch und verwenden zur Beschreibung das XML-Format.

Um diese Datei mit Inhalt zu füllen, wird zunächst das Launch-Element erstellt. Innerhalb dieses Elementes werden nun weitere Elemente beschrieben. So werden nun mittels des arg-Elementes globale Variablen wie der Startpunkt des Roboters innerhalb der Simulationswelt erstellt.

```
12  <!-- Robot pose -->
13    <arg name="x" default="2"/>
14    <arg name="y" default="2"/>
15    <arg name="z" default="0.09"/>
16    <arg name="roll" default="0"/>
17    <arg name="pitch" default="0"/>
18    <arg name="yaw" default="0"/>
```

Abbildung 35 Benutzung des arg-Elementes zum Erstellen globaler Variablen

Diese Parameter können nun bei der Erstellung neuer, sowie Initialisierung vorhandener Nodes verwendet werden. Eine Node zum Erzeugen des Roboters in der Simulationswelt wird wie folgt initialisiert.

```
22    <node name="robot_spawner" pkg="gazebo_ros" type="spawn_model" output="screen"
23          args="-urdf -model stepper_mini -param robot_description
24                -x $(arg x) -y $(arg y) -z $(arg z)
25                 -R $(arg roll) -P $(arg pitch) -Y $(arg yaw)"/>
```

Abbildung 36 Erstellen einer Node

So werden weitere Nodes aus Systeminternen sowie eigenen Projektpaketen gestartet, um verschiedene Dienste zu nutzen:

- Controller_manager: Mit Hilfe dieser Node erfolgt die Steuerung eines Robotermechanismus, welches mittels der Transmission-Elemente beschrieben wurden. Es bietet zudem eine Schnittstelle zum Laden, Starten und Zurücksetzen der Controller.
- Robot_state_publisher: Eine Node dieses Pakets erlaubt es den Zustand eines Roboters zu veröffentlichen. Als Eingabe nimmt das Paket die Gelenkwinkel des Roboters und setzt diese mit Hilfe des kinematischen Baummodells in 3-D Posen um.

Des Weiteren wird über das „<include>“ Element eine weitere Laucnhdatei zum Laden einer standardisierten Gazebowelt gestartet. Die Yaml-Konfigurationsdateien aus dem Config-Ordner werden mittels dem „rosparam“ Element geladen. Diese beiden Elemente werden im folgendem dargestellt.

```
5    <include file="$(find stepper_mini)/launch/world_launcher.launch">
6    </include>
```

Abbildung 37 Beschreibung des include-Elementes

```
28   <rosparam command="load" file="$(find stepper_mini)/config/joints.yaml"/>
```

Abbildung 38 Beschreibung des rosparam-Elementes

Abschließend wird für den Odometer-Plug in eine weitere Node initialisiert. Diese ruft den im Skript enthaltenen Code namens „get_gazebo_model_odometry.py“ und liefert bei Abfrage dieser Node die aktuelle Position des Roboters.

3.3.5 Einstellen des Position-Controllers

Wie in der realen Umgebung müssen auch in der Simulation die Regler der Gelenkmotoren eingestellt werden. Es muss jedoch vorher sichergestellt werden, dass die eingestellten Reibungswerte der Bodenfläche und der einzelnen Links des Robotermodells sich innerhalb der Simulation richtig verhalten. So kann eine falsche Parametrisierung der physikalischen Werte zu einer Reihe von Fehlern führen. Dabei kann unter anderem eine nicht ausreichende Dämpfung oder Reibung zur Oszillation der inneren Kräfte und somit zum Absturz der Physik-Engine führen. Oberflächlich kann dies bereits mit Gazebo geprüft werden. Es bietet eine Möglichkeit Kräfte und Momente auf einzelne Links des Modells zu wirken und deren Rückwirkung zu beobachten. Sind diese passend eingestellt, können die PID-Regler angepasst werden.

Als Regler wird ein Position-Controller der Obergruppe Kraftregler eingesetzt mit einstellbaren P, I und D-Glied als Parametern. Zum Einstellen der PID-Werte des Controllers wird das Programm *rqt_gui* verwendet.

Innerhalb des Programms werden folgende drei Plugins gestartet.

- Plugins/Topics/Message Publisher: Plugin zum Zugreifen auf die Nodes zum Steuern der Gelenke
- Plugins/Visualization/Plot: Ausgabe eines Plotfensters mit Auswahl von Nodes und Topics
- Plugins/Configuration/Dynamic Reconfigure: Plugin zum schrittweisen verstellen der PID-Werte bei einer gleichzeitig laufenden Simulation

Zunächst müssen im Fenster “Message Publisher” alle nötigen Joints hinzugefügt werden. Das Hinzufügen erfolgt über die *Topic* Auswahlbox. Die Joints des Roboters sind mit dem Paketnamen des Projektpaketes gegenzeichnet. Wichtig dabei ist die Zeile des jeweiligen Joints zu nehmen, die *command* als letzten Element besitzt. Über dieses Topic werden dem Roboter die Positionen der Winkel übermittelt.

Als Typ muss std_msgs/Float64 ausgewählt werden, sofern das nicht als Default steht. Die Frequenz wird auf 50 Herz eingestellt. Über das *expression* Element lassen sich nun je nach Controller Typ Winkel oder Positionen der Joints einstellen. Um die vorgegebenen Werte mit tatsächlichen zu vergleichen, müssen beide Werte im MatPlot Fenster geplottet werden. Um die eingestellten Positionswerte zu plotten, muss das Topic des jeweiligen Joints in der Topic Eingabebox eingefügt werden

Um die PID-Werte einzustellen, wird nacheinander jedem Gelenk eines einzelnen Beines eine Sinusfunktion sowie ein fester Wert von 0.8 Rad eingegeben. Mittels dieser Eingaben wird der Regler für das Umstellen der Beine eingestellt. Die aktuelle und die vorgegebene Position der Gelenke werden geplottet und miteinander verglichen. Neben dem Plotfenster können die Veränderungen auch in der Simulationsumgebung betrachtet werden. So wird innerhalb der Simulation auf die Präzision der Bewegungen, Reaktionszeit des Reglers und möglichst sanftes Absetzen der Beine geachtet. Beim Bewegen der Gelenke dürfen keine zu großen Kräfte hervorzurufen werden, welche das Modell umstoßen oder verschieben. Die jeweiligen PID-Werte werden in dem Dynamic Reconfigure Fenster schrittweise angepasst.

Da es keine automatischen Programme zum Einstellen der Regler gibt, wird das Anpassen der PID-Werte rein manuell und anhand der Erfahrungswerte durchgeführt.

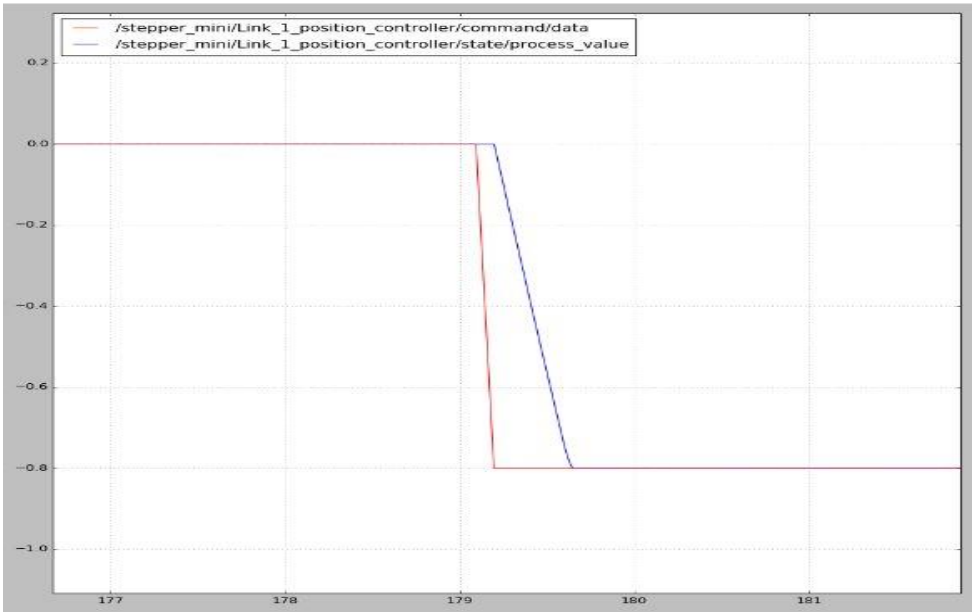

Abbildung 39 Impulsantwort des Schultergelenkes auf Eingabe von 0.8 Rad

Analog dazu wird das Gelenk von dem Schulterbauteil zu dem oberen Glied durchgeführt. Die resultierenden Werte werden innerhalb der Configdatei hinterlegt und mittels der Launchdatei in den Parameter-Server geladen.

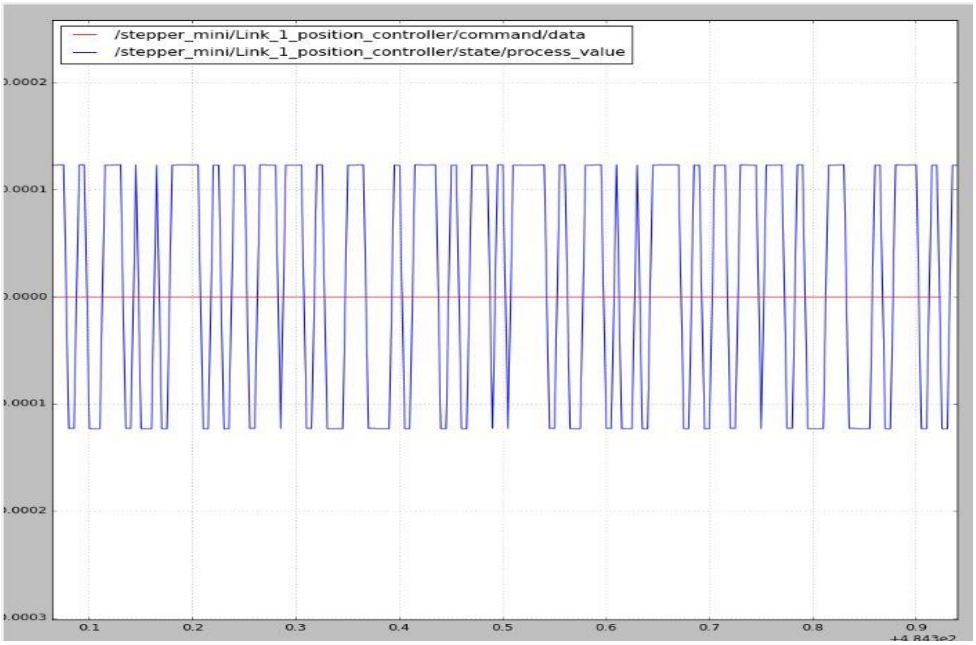

Abbildung 40 Restliche Streuung des Schultergelenkes

Die restliche Streuung von ca. 0.0001 Rad lässt sich auch nach Einstellen der Regler der Aktuatoren nicht komplett wegregeln. Wegen dem Aufbau der Move-Funktionen, in den zu der aktuellen Position der Gelenke die 0.8 Rad addiert werden, führt es zu den Oszillationen der Position nach einigen Hundert Schritten. Deshalb werden Funktionen implementiert, welche bei Detektion einer Streuung den Wert des Winkels zurücksetzen. Dies entfernt Oszillationen der Werte und hält somit an den Lernalgorithmus übergebene Zustandswerte stets gleich.

3.4 Erstellen der Reinforcement Learning Umgebung

Alle Bestandteile der RL-Umgebung werden innerhalb eines weiteren ROS-Projektpaketes namens „Stepper_training“ erstellt. Dabei wird ähnlich dem Roboter-Projektpaket eine Ordnerstruktur geschaffen. Diese beinhaltet den erstellten Pythoncode sowie eine Konfigurations- und Launchdatei befindet.

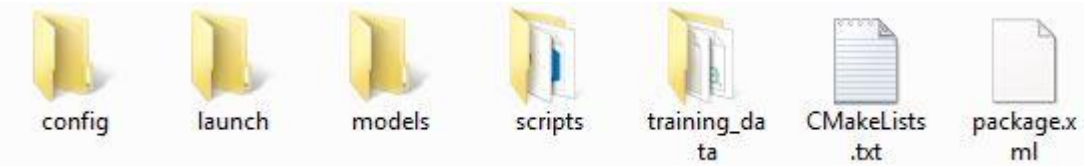

Abbildung 41 Ordnerstruktur des Trainingspaketes

- Config: Der Config Ordner enthält die Datei „openai_params.yaml“. In dieser Datei werden alle einzustellenden Parameter für den gesamten Code dieses Trainingspakets hinterlegt. Dazu gehören die Parameter zum Erstellen des Target- und DQN-Netzwerkes, Lerndauer in Episoden und Lernschritten sowie weitere globale Parameter für die Lernumgebung.
- Launch: Mittels der Launchdateien werden Pythonskripte gestartet.
- Models: In diesem Ordner wird das zuletzt trainierte Model in h5- und yaml-Format abgespeichert.
- Scripts: Dieser Ordner beinhaltet die erstellten Pythonskripte. Dazu gehört die Klasse „Env“ der Lernumgebung und die Klasse „DQN_Stepper_Mini“ der Lernmethode. Außerdem befinden sich darin zwei weitere Hilfsklassen aus dem Paket Openai_ROS. Diese sind die Klassen „GazeboConnection“ zum Starten,

Pausieren und Zurücksetzen der Simulation in Gazebo und „ControllersConnection“ für das Zurücksetzen der Controller der Aktuatoren.

- Training_data: Dieser Ordner enthält eine CSV-Datei mit den gesammelten Trainingsdaten in (s, a, r, s‘, d) Format. Zudem dient es als Ablageort für die während des Trainings erstellten Diagramme.

3.4.1 Lernalgorithmus

Der Lernalgorithmus wird innerhalb der DQN_Stepper_Mini Klasse implementiert. Es stellt den Startpunkt des gesamten Lernprozesses dar. So werden innerhalb dieser Klasse alle für den Lernalgorithmus nötigen Funktionen erstellt. Diese beinhalten das Erstellen der Netzwerke, deren Training und Auswertung. Unabhängig von dem Lernablauf wird bei jeder Initialisierung dieser Klasse entweder das vorhandene DQN-Netzwerk geladen oder neu erstellt. Das Target-Netzwerk wird jedes Mal neu erstellt und übernimmt zu Beginn die Gewichte des DQN-Netzwerkes. Der Lernalgorithmus zum Trainieren der Netzwerke wird innerhalb der run-Funktion implementiert. Anhand des Pseudocodes in Abbildung 42 soll der wesentliche Ablauf verdeutlich werden.

Algorithm 1 DQN-Algorithm

```
for jede Episode do
    St ← Env.reset()
    done ← False
    for jeden Schritt in einer Episode do
        At ← choose_action(St)
        St + 1, r, d ← Env.step(At)
        Speichere Datensatz im Exp. Replay Speicher
        if Exp. Replay Speicher >= Batchsize then
            if gesamte Schrittanzahl%1000 == 0 then
                model_train(True)
            else
                model_train(False)
        if done == False then
            St ← St+1
        else
            Beende die Episode
    Epsilon neuberechnen
    Abspeichern des Modells
```

Abbildung 42 DQN-Lernalgorithmus

Das Erstellen der Netzwerke und Initialisieren der Variablen erfolgt dabei vor dem Beginn des Lernablaufs.

Anhand Abbildung 43 soll die Architektur der DQN_Stepper_Mini Klasse veranschaulicht dargestellt werden. Die blauen Kästchen stellen dabei die in dieser Klasse enthaltenen Funktionen dar. Das Aufrufen der nächsten Aktionen wird anhand der Pfeile dargestellt. Diese Darstellung enthält nur selbst erstellte Funktionen. Funktionen eingebundener Bibliotheken werden hierbei nicht dargestellt.

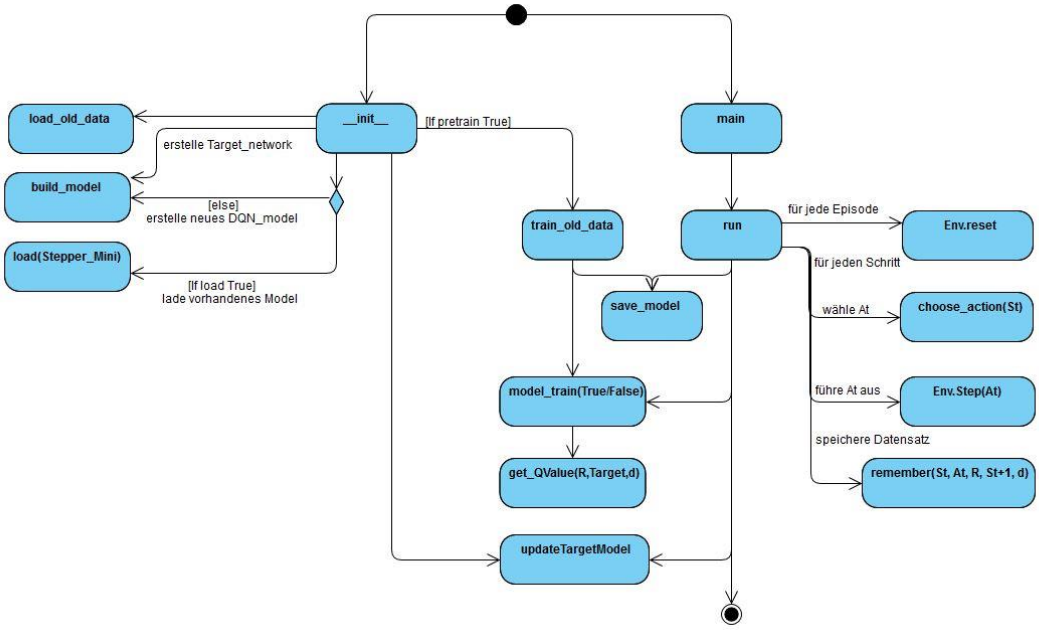

Abbildung 43 Grobe Architektur der DQN_Stepper_Mini Klasse

Im folgenden Abschnitt werden alle erstellten Funktionen beschrieben:

- Main: Diese Funktion initialisiert die Klasse und startet den Lernvorgang mittels der run-Funktion.
- __init__: In dieser Funktion werden die wichtigsten globalen Variablen deklariert. Die Werte der Variablen werden aus der Konfigurationsdatei „openai_params.yaml“ extrahiert. Des Weiteren wird bei Anweisung ein vorhandenes Modell geladen oder neue erstellt.
- build_model: Diese Funktion erstellt und konfiguriert das Target- und das DQN-Netzwerk.
- run: In dieser Funktion wird episoden- und schrittweise die durch das DQN-Netzwerk ausgewählten Aktionen ausgeführt und der Lernumgebungsklasse „Env“ übergeben. Des Weiteren wird aus dieser Funktion das DQN- und das Target-Netzwerk trainiert.
- load_old_data: Diese Funktion lädt vorhandene Datensätze und fügt diese dem Experience Replay Speicher hinzu.

- train_old_data: Verwendet den Experience Replay Speicher, um den Target- und DQN-Netzwerk zu trainieren. Das Target-Netzwerk wird alle tausend Schritte trainiert. Das erstellte Modell wird am Ende des Trainings abgespeichert.
- load: Lädt das im Ordner models erstellte Modell.
- updateTargetModel: Kopiert bei Aufruf die Gewichtsparamater des DQN-Modells in das Target-Modell
- remember: Diese Funktion speichert den Datensatz der Form (*s, a, r, s‘, d)* in dem Experience Replay Speicher. Enthält sowohl geladene als auch neue Datensätze.
- Env.reset: Startet die reset-Funktion der Umgebungsklasse Env.und liefert den ersten Zustand S_t der Umgebung
- Choose_action (S_t): Wählt abhängig von Epsilon, ob die nächste Aktion zufällig oder vorhergesagt wird. Das Vorhersagen der Aktion mittels des DQN-Netzwerks erfolgt anhand des Zustandes. Diese Funktion bekommt den Zustand als Eingabe und liefert eine zufällige oder vorbestimmte Aktion zurück.
- Get_QValue(r, Target, d): Innerhalb dieser Funktion wird der Q-Wert einer Aktion berechnet. Es wird dabei zwischen einem positiven und negativen terminierenden Zustand unterschieden. Der positive terminierende Zustand ist dabei das Erreichen des Ziels. Bei dem negativen terminierenden Zustand nimmt der Q-Wert den Wert der Belohnung dieser Aktion an, wobei diese immer negativ ausfällt.
- Env.Step(At): Mittels dieser Funktion wird eine ausgewählte Aktion an die Env-Klasse übermittelt und ausgeführt. Als Rückgabe wird der Datensatz der Form *(s, a, r, s‘, d)* ausgegeben.
- Model_train(True/False): Nimmt aus dem Experience Replay Speicher ein Sample und trainiert das Model. Zudem bestimmt bei True die nächsten Q-Targets und übermittelt diesen an die Funktion Get_QValue().
- save_model: Speichert das erstellte DQN-Netzwerk im Ordner „models“.

Experience Replay Speicher

Im Programm wird der Experience Replay Speicher in Form einer Deque() Funktion mit einer festen Größe von 500000 Datensätzen implementiert. Es speichert die Datensätze in Form von *(s, a, r, s‘, d)* und beinhaltet sowohl geladene als auch neue Datensätze. Im Laufe des Trainings wird bei jeder neuen Trainingsepisode mittels der Funktion random.sample(Speicher, Batchsize) batchweise zufällige Datensätze aus diesem Speicher rausgenommen, wobei die eingestellte Batchsize 64 Datensätze beträgt. Diese Datensätze werden anschließend zum Trainieren des Netzwerkes eingesetzt. Der Experience Replay Speicher wird dabei in der Variable Memory gespeichert.

Target Network

Wie im Kapitel 2.2.1 beschrieben wird das Target-Netzwerk in Form eines weiteren neuronalen Netzwerkes implementiert. Das DQN- und das Target-Netzwerk haben dabei den identischen Aufbau. Aufgebaut werden die beiden Netzwerke mittels der Funktion „build_model“ (siehe Abb. 43). Des Weiteren wird die Funktion „updateTargetModel“ erstellt. Bei Aufruf dieser Funktion werden mittels der Befehle get_weights und set_weights die Gewichtsparameter des DQN-Netzwerkes in das Target-Netzwerk kopiert. Alle tausend Schritte werden die Parameter kopiert und das Target-Netzwerk neu trainiert.

Netzwerkarchitektur

Das DQN- und das Target-Netzwerk wird mittels der Bibliothek Keras schichtweise aufgebaut. Das Erstellen des Netzwerkes wird dabei mit folgenden Funktionen durchgeführt:

- Sequential: Mit dieser Funktion wird das Model initialisiert.
- Add: Diese Funktion fügt dem Netzwerk je nach Aufruf eine Schicht hinzu, wobei bei der ersten verdeckten Schicht die Eingabeschicht miterstellt wird. Zusätzlich muss ab der Eingabeschicht jede weitere Schicht parametrisiert werden. Die

wichtigsten Parameter sind die Anzahl an Neuronen und die Aktivierungsfunktion.

- Compile: Mit dieser Funktion wird der Lernprozess konfiguriert. Dabei wird eine Optimierungsmethode, Kostenfunktion sowie eine Metrik, die zum Auswerten der Genauigkeit eines Modells genutzt wird, eingestellt.
- Summary: Gibt eine zusammenfassende Darstellung des Netzwerkes aus.

Der Aufbau dieser beiden Netzwerke ist dabei identisch und wird folgendermaßen implementiert:

```
60      def build_model(self):
61
62          model = Sequential()
63          model.add(Dense(64, input_dim=(self.state_size), activation='relu', kernel_initializer='lecun_uniform'))
64          model.add(Dense(64, activation='relu', kernel_initializer='lecun_uniform'))
65          model.add(Dense(self.action_size, activation='linear', kernel_initializer='lecun_uniform'))
66          model.compile(loss='mse', optimizer=RMSprop(lr=self.alpha, rho=0.9, epsilon=1e-06), metrics=["acc"])
67          model.summary()
68
69          return model
```

Abbildung 44 Funktion zum Aufbauen der Netzwerke

Das Netzwerk besteht neben der Ein- und Ausgabeschicht aus zwei verdeckten Schichten. Die verdeckten Schichten werden wie folgt konfiguriert:

- Neuronenanzhal: 64
- Aktivierungsfunktion: relu
- Initialisierer: lecun_uniform
- Lernrate 0.0025
- Kostenfunktion: MSE

Als Eingabe dient der Zustand des Roboters und der Umgebung. Dieser ist sechs Werte lang und besteht aus den vier Winkelpositionen des Schultergelenkes, der Distanz zum Ziel und der Ausrichtung zum Ziel. Als Ausgabe dienen die Q-Werte der zwölf möglichen Aktionen.

3.4.2 Lernumgebung

Die Lernumgebung des Agenten wird innerhalb der Klasse „Env“ erstellt. Es beinhaltet alle nötige Funktion zum Erstellen und Einsetzen der Lernumgebung. Dazu gehört das verarbeiten der übermittelten Aktion, Ansteuern der Gelenke, Auswertung der Sensordaten und abschließende Berechnung der Belohnung. Zudem implementiert diese Klasse die von OpenAI Gym-Infrastruktur benötigten Funktionen step, init und reset. Zu Beginn wird die Umgebung mittels des folgenden Befehles initialisiert:

```
25  reg = register(
26      id='World-v0',
27      entry_point='world_env:Env',
28      timestep_limit= 100,
29      )
```

Abbildung 45 Initialisierung der Umgebung

Dabei wird der Name der Umgebung, dessen Eingangspunkt und die Anzahl der Schritte pro Episode deklariert. Zudem werden bei dem ersten Aufruf dieser Klasse durch die DQN_Stepper_Mini-Klasse alle in der Konfigurationsdatei definierten Variablen geladen. Wie im vorherigen Kapitel beschrieben, kommuniziert der Lernalgorithmus über die reset- und step-Funktion mit der Lernumgebung.

Algorithm 2 Ablauf der reset-Funktion

1: *delete_model(goal)* ▷ lösche das Zielmodel
2: *gazebo.unpauseSim()* ▷ Starten der Simulation
3: *controllers _object.reset_controllers()* ▷ Zurücksetzen der Controller
4: *check_all_sensors_ready()* ▷ prüfe alle Sensoren
5: *gazebo.pauseSim()* ▷ Simulation pausieren
6: *gazebo.resetSim()* ▷ Zurücksetzen der Simulation
7: *gazebo.unpauseSim()* ▷ Starten der Simulation
8: *controllers_object.reset_controllers()* ▷ Zurücksetzen der Controller
9: *check_all_sensors_ready()* ▷ prüfe alle Sensoren
10: *gazebo.pauseSim()* ▷ Simulation pausieren
11: *init_env_variables()* ▷ initialisiere Umgebungsvariablen
12: *init_goal()* ▷ erzeuge das Zielmodell
13: *obs* ← *get_obs()* ▷ sammle Umgebungsdaten
14: *joint_pos* ← *get_joint_states()* ▷ sammle Jointdaten (Winkelpositionen)
15: *states* ← *convert_obs_to_state(obs, joint_pos)* ▷ erstelle Zustandsvektor
16: *return(states)* ▷ rückgabe des Zustandsvektors an den Lernalgorithmus

Abbildung 46 Aufbau der reset-Funktion

Algorithm 3 Ablauf der step-Funktion

```
1: gazebo.unpauseSim()                                  ▷ Starten der Simulation
2: set_action(At)                                       ▷ führe übermittelte Aktion aus
3: gazebo.pauseSim()                                    ▷ Simulation pausieren
4: obs ← get_obs()                                      ▷ sammle Umgebungsdaten
5: joint_pos ← get_joint_states()                       ▷ sammle Jointdaten (Winkelpositionen)
6: done ← is_done(obs, joint_states)                    ▷ prüfe done-Kriterien
7: reward ← compute_reward(obs, joint_pos)              ▷ berechne Belohnung
8: states ← convert_obs_to_state(obs, joint_pos)        ▷ erstelle Zustandsvektor
9: delete_goal(done))                                   ▷ lösche das Zielmodell wenn done ist True
10: return(states, reward, done)                        ▷ Rückgabe des Datensatzes (St+1, r ,d)
```

Abbildung 47 Aufbau der step-Funktion

Alle in der Env-Klasse implementierten Funktionen werden wie folgt beschrieben:

- __init__: Initialisiert alle vordefinierte Variablen, Controller und Publisher. Die Publisher werden zum Übermitteln der Winkelpositionen an die jeweiligen Gelenke benutzt.
- Reset: Die Reset-Funktion ist eine Ansammlung von weiteren Funktionen (siehe Abb. 46). Diese setzen die Simulationsumgebung zurück und liefern den ersten Zustand St. Diese Funktion wird zu Beginn jeder Episode ausgeführt.
- Step: Die Step-Funktion wird innerhalb jedes Lernschrittes ausgeführt und führt die aus DQN_Stepepr_Mini-Klasse übermittelte Aktion (siehe Abb. 47). Über weitere Aktionen wird der resultierende Zustand beobachtet, ausgewertet und in Form von *(r, s‘, d)* an die DQN_Stepper_Mini-Klasse übermittelt.
- Init_env_variables: Initialisiert alle globalen Variablen. Im Gegensatz zu den Variablen aus der __ini__ Funktion werden diese Variablen bei jedem Reset zurückgesetzt.
- Is_done: Funktion zum Detektieren eines terminierenden Zustandes.
- get_obs: Diese Funktion ermittelt anhand weiterer Funktionen die Distanz zum Ziel, Ausrichtung zum Ziel sowie die nach einer Aktion resultierenden Neigungswinkel um xyz-Achsen des Roboters. Die ausgelesenen Werte werden zurück an die Step- oder Reset-Funktion übermittelt.

- Get_orientation_euler: Liest die Odometerdaten aus und bestimmt die xyz-Neigungswinkel des Körpers.
- Get_dist_to_goal: Liest die Odometerdaten aus und bestimmt die Distanz zum Ziel.
- Compute_reward: Belohnungsfunktion der Lernumgebung.
- Set_action: Dies ist eine übergeordnete Funktion zum Ausführen der Aktionen. Neben dem Ausführen der jeweiligen Funktionen der Aktionen werden hierbei die Drehrichtung und die Nummer des gewählten Beines übermittelt, sodass die Anzahl an weitere Funktion minimiert werden kann.
- move_Leg: Funktion zum Einziehen und Ausstrecken des Beinelementes.
- Shift_base_2_4: Verschiebungsaktion des Beinpaares 2/4.
- Shift_base_1_3: Verschiebungsaktion des Beinpaares 1/3.
- Turn: Funktion zum Drehen des Roboters.
- Get_joint_states: Liest die Winkelpositionen der Schultergelenke aus und rundet diese auf eine Nachkommastelle.
- Normalise: Setzt die Winkelpositionen der Gelenke periodisch auf ihre Startwerte zurück, um Oszillation zu vermeiden.
- Convert_obs_to_state: Konvertiert die Winkelpostionen der Gelenke, den Abstand und die Ausrichtung zum Ziel zur einer Zustandliste.
- Init_goal: Setzt je nach dem in der Konfigurations-Datei definiertem Szenario die Position des Ziels in der Simulationsumgebung. Erzeugt zudem das Objekt des Ziels innerhalb der Simulation.
- Delete_goal: Löscht das Zielobjekt aus der Simulation.
- Check_publishers_connection: Überprüft alle Publisher-Topics auf ihre Einsetzbarkeit.
- Joints_callback: Funktion zum Zugreifen auf die Gelenkdaten.
- Odom_callback: Funktion zum Zugreifen auf die Odometerdaten.
- check_joints_states_ready: Prüft die Bereitschaft des joint_states-Topics.
- check_odom_ready: Prüft die Bereitschaft des Odom-Topics.
- Check_all_sensors_ready: Startet die Funktionen check_joints_states_ready und check_odom_ready.

3.4.3 Belohnungssystem

Das Belohnungssystem der Lernumgebung beschreibt die Aufgabe des Agenten. Die positiven und negativen Belohnungen müssen dabei in einem richtigen Verhältnis zueinander und zu den Aktionen modelliert werden. Eine ungeeignete Modellierung kann zu unvorhergesehenen Fehlern führen. Zudem wird die Belohnungsfunktion meist aus Erfahrungswerten und anhand langer Testreihen erstellt.

Das gesamte Belohnungssystem des Agenten wird innerhalb von Funktion „reward“ und „done“ implementiert und lässt sich somit bequem anpassen oder austauschen. Diese beiden Funktionen werden erst nach dem Ausführen einer Aktion und dem Sammeln der Sensordaten ausgeführt. Die done-Funktion wird dabei als Erstes ausgeführt und übermittelt an die reward-Funktion eine Boolean-Variable, die einen terminierenden Zustand angibt.

Done-Funktion

Zunächst wird die done-Funktion behandelt, denn diese ist zum Detektieren von Kriterien zuständig, die ein Zurücksetzen der Umgebung und des Roboters herbeirufen. Es berechnet somit selbst keine Belohnungen. Sollte eine dieser Kriterien zutreffen, wird ein done-Marker auf True gesetzt. Dieser wird in weiteren Funktionen zusätzlich zur Berechnung der Belohnung eingesetzt und leitet das Zurücksetzten der Lernumgebung ein. Bei nicht zutreffen einer dieser Kriterien bleibt der done-Marker auf False und erlaubt das Ausführen weiterer Aktionen. Als Eingabe dieser Funktion fungieren folgende Daten:

- Observations: Dies ist ein Tupel mit aus der Simulationsumgebung ausgelesenen Daten des Roboters und des Ziels. Dazu gehören die ermittelte Distanz zum Ziel, Ausrichtung zum Ziel sowie die Neigungswinkel der xyz-Achsen des Körpers.
- Joint_states: Dies ist ein Tupel mit Winkelpositionen der vier Schultergelenke und der Gelenke zwischen der Schulter und des oberen Beinelementes.

Anhand dieser Daten werden folgende Kriterien aufgestellt:

- Neigungsbegrenzung: Durch eine ausgeführte Aktion resultierende Neigungswinkel um die x- und y-Achse des Körpers überschreitet den Maximalwert von 0.1 Rad.
- Bewegungsbegrenzung: Durch eine ausgeführte Aktion resultierende Winkelposition der Beine überschreiten die Begrenzungen des erlaubten Bewegungsraumes der einzelnen Beine. Die erlaubte Toleranz des Bewegungsraumes beträgt + 0.1 Rad im Falle eines ausgestreckten Beines und – 0.9 im Falle eines eingezogenen Beines. Diese Abfrage gilt für alle vier Beine, wobei die Vorzeichen der Toleranzwerte sich je nach Koordinatensystem ändern.
- Zielfeld: Die aktuelle Distanz zum Ziel unterschreitet einen Minimalwert von 0.25 Metern. Trifft diese Anweisung zu, so wird ein zusätzlicher Marker „goal_reached“ gesetzt. Dieser bedeutet das Erreichen des Ziels und wird innerhalb der reward-Funktion weiter ausgewertet. Die Distanz wird dabei von der Mitte des Körpers bis zur Mitte des Zielfeldes berechnet.

Reward-Funktion

Die reward-Funktion ist für die Berechnung der direkten Belohnungen zuständig, die anhand von mehreren Kriterien zustande kommen. Zur Auswertung des Zustandes und Berechnung der Belohnung werden dieser Funktion die Variablen Observations und Done übermittelt. Mit Hilfe dieser Werte wird ein Modell der Belohnungsfunktion erstellt. Zu Beginn der Auswertung wird automatisch eine negative Belohnung von 30 initialisiert. Da der Q-Wert anhand der Belohnungen errechnet wird, senk dies die Wahrscheinlichkeit des Ausführens sich nicht lohnender Aktionen.

Analog zu der done-Funktion werden hier einige Kriterien aufgestellt. Der anfängliche Wert bei jeder Abfrage der Kriterien kann in positive oder negative Richtung weiter angepasst werden. Zusätzlich zu den Kriterien werden Abfragen erstellt, die erfüllt werden müssen. Die Größe der Distanz und der Ausrichtung wird dabei mit einem weiteren Faktor multipliziert. Die Größe dieser Faktoren wird so gewählt, dass der

Agent die größten Belohnungen durch das Ausführen der Verschiebungsaktionen sammelt. Da für eine Verschiebungsaktion meistens mindestens zwei Aktionen ausgeführt werden, muss die resultierende Belohnung dies ausgleichen.

Verringerung der Distanz durch Verschiebungsaktion

Bei diesem Kriterium wird die Differenz der aktuellen Distanz mit der vorherigen verglichen. Dabei gibt es folgende Unterpunkte, wobei alle davon zutreffen müssen:

- Aktuelle Distanz < vorherige Distanz
- Ausgeführte Aktion ist 8 oder 9
- done ist False

Dieses Kriterium wird nur abgefragt, solange der done-Marker auf False gesetzt ist und diese nur durch die dafür vorgesehenen Aktionen hervorgerufen wurden. So erhält der Agent keine Belohnung für die Verringerung der Distanz zum Ziel, die durch mögliche Simulationsfehler wie das Gleiten der Beine bei einer fehlerhaften Reibung hervorgerufen werden.

Die maximal zurückgelegte Distanz pro Verschiebungsaktion beträgt dabei ca. 0.1 Meter. Der Faktor dieser Aktion beträgt 1200. Daraus lässt sich für diese Aktionsart eine maximale Belohnung von 120. Dieser Wert unterliegt dabei den anfänglichen Abzug von 30, sodass die resultierende Belohnung den Wert 90 ergibt.

Verringerung der Ausrichtung durch Drehaktion

Analog zu vorherigem Kriterium wird für die Ausrichtung die aktuelle Ausrichtung mit der Vorherigen verglichen. Weiterhin sind folgende Unterpunkte zu erfüllen:

- Aktuelle Ausrichtung < vorherige Ausrichtung
- Ausgeführte Aktion ist 10 oder 11
- done ist False

Die maximale Änderung der Ausrichtung durch eine Drehaktion beträgt dabei 0.4 Rad. Mit einem Faktor von 140 beträgt nun die maximal erreichbare Belohnung den Wert

56. Nach dem anfänglichen Abzug beträgt die resultierende Belohnung den Wert 26. Der durchschnittliche Belohnungswert der Drehaktion kann je nach Position der Beine variieren. Die Belohnung der Drehaktion darf zudem den anfänglichen Abzug nicht überschreiten, da sonst der Agent in eine Dauerschleife dieser Aktion verfällt. Zusätzlich soll sich die Drehaktion nur zu Beginn einer Episode und zwischenzeitigem Ausrichten lohnen.

Verringerung der Ausrichtung durch Verschiebungsaktion

Beim Verringern der Ausrichtung werden neben den Drehaktionen zehn und elf auch in einer separaten Abfrage die Verschiebungsaktionen acht und neun abgefragt. Wie in Kapitel 3.2.3 erwähnt wurde, könnte sich der Agent darauf versteifen für den gesamten Weg nur ein Beinpaar zum Verschieben des Körpers einzusetzen. Dies führt neben dem ungeeigneten Bewegungsmuster zu einer sich kontinuierlich oszillierenden Ausrichtung zum Ziel. Da vor allem die resultierenden Kräfte beim Zurückdrehen der Schultergelenke nicht komplett in die Vorwärtsbewegung übertragen werden. So entfernt sich der Roboter immer weiter von der geraden Linie. Da nun die beiden Verschiebungsaktionen neben der Differenzbelohnung auch die Belohnung durch richtige Ausrichtung zum Ziel erhalten können, müsste der Agent es bei genügend Trainingsdaten erlernen die Verschiebungsaktionen abwechselnd auszuführen. Folgend werden dafür nötigen Kriterien aufgelistet:

- Aktuelle Ausrichtung < vorherige Ausrichtung
- Ausgeführte Aktion ist 8 oder 9
- done ist False

Die maximale Änderung der Ausrichtung durch eine Verschiebungsaktion ist dabei 0.1 Rad. Die Belohnung dieses Kriteriums hat einen Faktor von 100. Dabei ergibt sich eine maximale Belohnung von 10, wobei dieser Wert dem anfänglichen Abzug nicht unterliegt. Zudem ist diese Belohnung kumulativ mit dem ersten Kriterium. Dadurch resultiert eine maximale Belohnung inkl. des Abzuges den Wert 100. Der Faktor bei Verringerung der Ausrichtung durch eine Drehaktion muss dabei größer sein als durch

eine Verschiebungsaktion. Dadurch soll der Agent die Drehaktionen gegenüber den Verschiebungsaktionen zum Ändern der Ausrichtung bevorzugen.

Stabilitätskriterium

Die Idee dabei ist, dass das Ausstrecken der Beine, die dem Roboter mehr Stabilität für die nächsten Aktionen verleihen keinen Abzug bekommen. Dabei wird für jedes Schultergelenk und die dazugehörige Aktion eine Abfrage erstellt.

- Die Winkelposition die Schultergelenke eins ist gleich null.
- Die dabei ausgeführte Aktion ist 0
- done ist False

Im gleichen Wege wird diese Abfrage für die Aktion 3, 5 und 7 erstellt. Bei Zutreffen dieses Kriteriums wird der anfängliche Abzug der Aktion verworfen. Die Aktion erhält nun eine Belohnung von 5.

Erhöhen der Ausrichtung durch eine Bewegungsaktion

Dieses Kriterium wird zusätzlich zu der Abfrage der Neigung der done-Funktion implementiert. Zudem erweitert es das vorherige Kriterium um die Abfrage, ob das Ausstrecken des Beines eine Verschiebung des Körpers hervorruft. Dieses Kriterium tritt in Kraft, wenn die beiden vorderen Beinglieder eingezogen sind. Wenn nun eines der beiden Beine ausgestreckt wird, befindet sich der Schwerpunkt des gesamten Roboters außerhalb des Stützpolygons. Dadurch fällt der Körper auf das bewegende Bein und resultierend bewegt sich der gesamte Körper in die jeweilige Richtung. Die Abfrage wird folgend aufgestellt:

- Aktuelle Ausrichtung > vorherige Ausrichtung
- Ausgeführte Aktion ist 0 bis 7

Bei Zutreffen dieser Abfrage kommt zusätzlich zu dem anfänglichen Kriterium ein weiterer Abzug von 10. Dadurch beträgt der maximale Abzug dieser Aktion den Wert 40.

Erreichen des Ziels

Bei diesem Kriterium geht es um die Belohnung für das Erreichen des Zielfeldes. Hierfür müssen folgende Kriterien zutreffen:

- done ist False
- goal_reached ist True

Die Größe der Belohnung beträgt bei diesem Kriterium einen Wert von 250. Dieser Wert ist mit anderen Kriterien nicht kumulativ. Durch diese Modellierung wird dem Agenten neben dem Bewegen in die richtige Richtung ein zusätzlicher Impuls zur Maximierung der Belohnung durch Erreichen des Ziels vorgegeben.

Terminierender Zustand

Dieses Kriterium trifft in Kraft, wenn das Ziel nicht erreicht und der Agent einen terminierenden Zustand durch die done-Funktion detektiert hat. Die Abfragen für dieses Kriterium werden folgendermaßen aufgestellt:

- done ist True
- goal_reached ist false

Die Bestrafung für dieses Kriterium beträgt den Wert -250. Der Agent soll lernen diese Zustände zu vermeiden. Zudem werden die Umgebung und der Roboter beim Zutreffen dieser Abfrage zurückgesetzt.

Für das Erreichen des Ziels und des Versagens wird kein Faktor eingestellt, sondern die Werte werden von Anfang an so eingestellt, dass sie den maximalen Gewinn erzeugen. Zudem sind die Belohnungen dieser Kriterien mit anderen nicht kumulativ.

Abschließend werden die aktuelle Distanz und Ausrichtung zum Ziel innerhalb der Klasse abgespeichert. Die abgespeicherten Werte gelten bei der Auswertung nächster Aktion als vorherige Vergleichswerte und werden erneut zum Auswerten der präsentierten Kriterien eingesetzt.

3.5 Lernvorgang

In diesem Kapitel folgt das Training des Agenten. Die zum Training benötigten Datensätze werden im Laufe mehrerer Szenarien gesammelt. Mittels den gesammelten Datensätzen wird das Netzwerk wie im Falle von Supervised Learning trainiert.

Die Szenarien unterscheiden sich in den Positionen, in denen das Ziel erzeugt wird. Anfänglich ist das Ziel nur auf eine Position begrenzt, sodass möglichst mehr Datensätze nur für die Fortbewegung mit einer festen Ausrichtung gesammelt werden. Im zweiten Szenario erweitern sich die möglichen Positionen zum Erzeugen des Ziels auf drei Punkte, um vermehrt Datensätze für das Drehen zu sammeln. Im letzten Szenario sind die möglichen Positionen nicht mehr vorgegeben und das Ziel wird zufällig innerhalb der Simulationsumgebung erzeugt.

Zu Beginn des Trainings für jedes Szenario werden mittels der Epsilon Greedy Strategie Datensätze gesammelt. Die Epsilon Greedy Parameter werden folgendermaßen eingestellt, wobei bei jedem Neustarten des Trainings die Parameter zurückgesetzt werden:

- Epsilon: 1
- Epsilon Decay: 0.9985
- Epsilon Min.: 0.3

Zu Anfang beträgt die Wahrscheinlichkeit eine zufällige Aktion auszuführen 100 Prozent. Dies erlaubt dem Agenten möglichst viele Folgezustände aus dem Startzustand zu erkunden. Nach jeder Episode wird das Epsilon neu berechnet. Zu Anfang sind die Episoden jeweils nur wenige Schritte lang, den der Agent kann bereits mit der ersten Aktion einen negativen terminierenden Zustand hervorrufen. Wurden während des Erkundens der Umgebung genug Datensätze für eine Batch gesammelt, wird in jedem weiteren Schritt das Netzwerk trainiert und abgespeichert. Mit einem stetig sinkenden Epsilon werden zunehmend Aktionen anhand des Netzwerkes bestimmt. Dadurch können Aktionen ausgeführt werden die zu erster Belohnung führen. Das minimale Epsilon wird dabei auf einen relativ hohen Wert von 30%

eingestellt, sodass auch wenn der Agent den halben Weg zum Ziel zurückgelegt hat, immer noch zufällige Aktionen zum Erkunden der Welt hervorgerufen werden.

Szenario 1

Bei dem ersten Szenario soll der Agent die richtige Reihenfolge des Bewegungsmusters erlernen. Die Ausrichtung bleibt bei diesem Szenario stets gleich, soweit der Roboter sich nicht selbst von der geraden Linie zum Ziel abwendet. Der Abstand zum Ziel wird hierbei anfänglich auf einen festen Wert voreingestellt, sodass für diese beiden Werte nicht approximiert werden muss. Der feste Zielpunkt innerhalb der Welt wird auf x = 3.5 Meter und y = 2 Meter gesetzt. Der Abstand zum Ziel wird anschließend etappenweise um 0.25 Meter verkleinert. Dadurch soll der Agent schneller die Zustände erlernen, die näher zum Ziel sind, ohne durch hervorgerufene Zufallsaktionen seinen zurückgelegten Weg zu verlieren. Der Roboter startet bei allen Szenarien immer in der Mitte der Fläche, welches in dem Punkt x = 2 Meter und y = 2 Meter liegt. Der Agent lernt zusätzlich das aus dem Zustand mit einer Ausrichtung von 0 Rad nicht gedreht werden muss, da bereits die erste Drehaktion bestraft wird. Dennoch werden Datensätze zum Drehen bereits im ersten Szenario gesammelt. Das Training in diesem Szenario endet, sobald der Roboter mehrmals nacheinander das Ziel erreicht.

Szenario 2

Das zweite Szenario erweitert die möglichen Startpunkte des Ziels auf drei Positionen (siehe Tabelle 5). Dadurch liegen die Zielpunkte jeweils zu 0, 135 und -135 Grad von der Startausrichtung des Roboters verdreht. Durch diese Positionierung soll der Agent nun effektiver das Drehen erlernen. Dies soll den Effekt haben, dass der Agent im letzten Szenario besser die vorhandenen Zustände für eine optimale Funktion approximieren kann. Das vorherige Modell und Datensätze werden geladen, um mit diesen weiter trainieren zu können.

	x	y
Positionen 1	3.5	2
Positionen 2	1	1
Positionen 3	1	3

Tabelle 4 Startpositionen des Ziels im zweiten Szenario

Szenario 3

Im dritten Szenario wird das Zielfeld in einer zufälligen Position innerhalb der Fläche erzeugt. Die Datensätze der vorherigen Szenarien werden wieder in den Experiece replay Speicher geladen. Das vorherige Modell wird ebenfalls geladen. Das im Szenario 3 trainiertes Modell gilt als finales Modell und wird hierbei 100000 Iterationen lang trainiert. Die Epsilon Greedy Parameter werden ebenfalls eingestellt, wobei Epsilon Min. nun auf den Wert 0 eingestellt wird.

4 Auswertung

Das folgende Kapitel dieser Arbeit beschäftigt sich mit der Auswertung des im vorherigen Kapitel trainierten Modells. Das trainierte Modell wird dazu verwendet das erste und zweite Szenario zu testen. Anhand erstellter Diagramme wird Leistung des Agenten besprochen. Des Weiteren werden visuelle Merkmale der erlernten Bewegung hinsichtlich der Belohnungskriterien beschrieben.

Szenario 1

Zur Auswertung des ersten Szenarios wird die über alle Schritte kumulativ gesammelte Belohnung über die Anzahl der ausgeführten Schritte geplottet. Die Testdauer beträgt dabei drei Episoden und endet nach ca. 175 Schritten. Eine Episode kann entweder mit dem Erreichen des Ziels oder dem Ausführen eines terminierenden Schrittes

enden. Das Erreichen des Ziels wird durch einen Anstieg von 250 signalisiert, wobei das Versagen einen Abfall von 250 bedeutet.

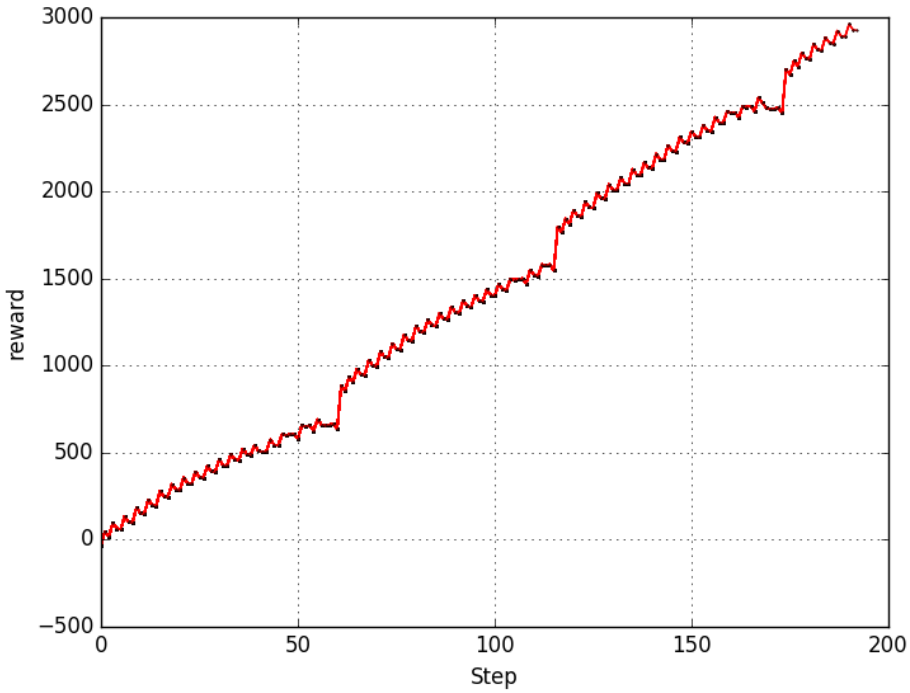

Abbildung 48 Kumulative Belohnung pro Schritt des ersten Szenarios

Anhand Abbildung 48 ist zu sehen, dass der Agent stetig passende Aktionen zum Erhöhen der Gesamtbelohnung ausgewählt hat. Im Laufe dieser drei Episoden ist der Agent stets innerhalb ihm stehender Anzahl von 100 Schritten zum Ziel gekommen. Im Schnitt waren es ca. 58 Schritte pro Episode. Die Steigung und der Verlauf sind dabei fast identisch, was eine hohe Wiederholgenauigkeit für dieses Szenario aufweist.

Der Roboter hat sich dabei immer über die gleiche Trajektorie zum Ziel bewegt. Dabei ist er jedes Mal von der Geraden abgekommen, da das Abwechseln der Schrittbewegungen nur bedingt funktioniert hat. Des Weiteren hat der Agent stets nur zwei Bewegungsaktionen zwischen den Verschiebungsaktionen durchgeführt. Als Folge hat der Agent das Ausstrecken eines Beines immer aus einer unstabilen Lage durchgeführt, wodurch der Körper auf das angehobene Bein gefallen ist und somit beim Bewegen des Beines auch den Körper verdreht hat. Dadurch hat der Agent auch

durch die Beinbewegungen seine Ausrichtung gewechselt. Er konnte dies aber nach Überschreiten einer bestimmten Ausrichtung durch eine Drehaktion ausgleichen.

Szenario 2

Die Auswertung des zweiten Szenarios läuft in gleicher Weise wie das erste Szenario ab. Dabei wird wie im Lernvorgang nur die Anzahl der möglichen Zielorte erhöht. Zur Auswertung wurden alle drei möglichen Zielorte hinzugezogen, um festzustellen, ob alle Punkte gleichermaßen trainiert worden sind. Zudem kann bei diesem Szenario der Einsatz der Drehaktion geprüft werden.

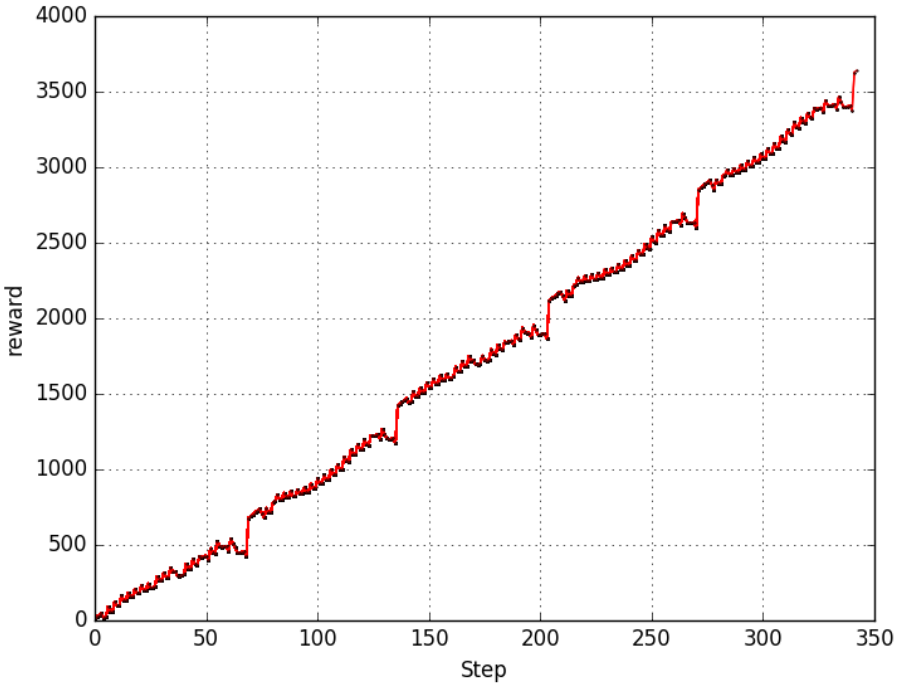

Abbildung 49 Kumulative Belohnung pro Schritt des zweiten Szenarios

Wie in Abbildung 49 zu sehen ist, führt der Agent auch hier stets passende Aktionen aus, um die Gesamtbelohnung zu maximieren. Im Laufe aller Episoden kam der Agent am Ziel an. Im Schnitt konnte der Agent innerhalb von ca, 70 Schritten eine Episode erfolgreich abschließen.

Bei der Fortbewegung zum Ziel hatte der Agent in diesem Szenario die gleichen Probleme wie im Ersten. Hinzu kommt das, der Agent zu Beginn einer Episode nie die

Ausrichtung komplett durch die Drehaktionen ausgeglichen hat. Dabei hat der Agent gelernt, dass die restliche Ausrichtung durch das Ausführen der jeweiligen Verschiebungsaktion ausgeglichen werden kann. Somit hat das Belohnungskriterium für das Abwechseln der Verschiebungsaktion eher einen negativen Effekt hervorgerufen.

Szenario 3

Die Auswertung des dritten Szenarios wird analog zu den vorherigen durchgeführt. Dabei wurde die gesamte Belohnung über zehn Episoden aufgezeichnet, bei den verschiedene Zielpunkte angesteuert wurden.

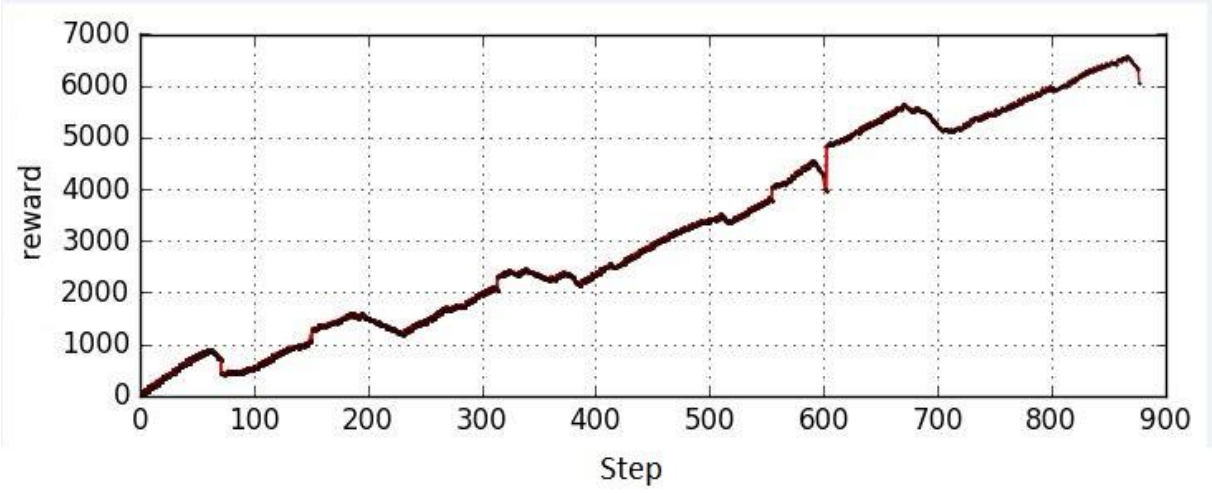

Abbildung 50 Kumulative Belohnung pro Schritt des dritten Szenarios

Wie in Abbildung 50 zu sehen ist, wird die gesamte Belohnung des Agenten stets erhöht. Dennoch kann der Agent in diesem Szenario nicht jede Episode erfolgreich abschließen. Von zehn durchgeführten Episoden hat der Agent nur zwei Episoden erfolgreich beendet. Des Weiteren wurden wie bei Schritt 600 zu sehen ist zwei Zielpunkte in unmittelbarer Nähe zum Roboter erzeugt. Dadurch hat der Agent diese Episoden sofort erfolgreich beendet. Manche der Episoden konnte der Agent nicht abschließen. Dies geschah, da der Agent kurz vor dem Ziel in einer Dauerschleife des Drehens gefangen war. Der Grund dafür ist, dass aus einer unvorteilhaften Position der Beine, die richtige Drehbewegung eine viel zu kleine Differenz der Ausrichtung

ergeben hat. So hat der Agent durch eine richtige Aktion eine negative Belohnung bekommen. Des Weiteren hat sich der Agent in einem hohen Bogen zum Ziel bewegt.

Da das Training des Models mehrere Etappen durchlief, ist es schwer zu sagen mit wie vielen Lernschritten trainiert wurde. Zudem hängt es sehr stark von der Qualität der Daten ab. Sich immer wiederholende Datensätze bringen nur wenig Mehrwert für das Netzwerk. Im Laufe der Tests der Belohnungsfunktionen wurden jedoch über 100000 Datensätze gesammelt.

Vorherige Trainingsversuche

Durch einen vorherigen Trainingsversuch mit einem vereinfachten Belohnungssystem konnte der Agent auch im dritten Szenario immer das Ziel erreichen. Der Grund dafür war vor allem ein längeres Training von ca. 500000 Schritten und über 220000 Datensätzen. Des Weiteren hatte das Belohnungssystem neben der done-Funktion und der Abfrage bei Erreichen des Ziels und Zutreffen der done-Kriterie nur die ersten beiden Belohnungskriterien. Diese sind das Belohnen bei Verminderung der Distanz und der Ausrichtung. Anhand des Bewegungsmusters hat der Agent gelernt nur eine der beiden Verschiebungsaktionen zu benutzen. Die restlichen Beine waren somit stets ausgestreckt und dienten nur als Stütze. Beim Trainieren dieses Modells ergab sich folgendes Diagramm:

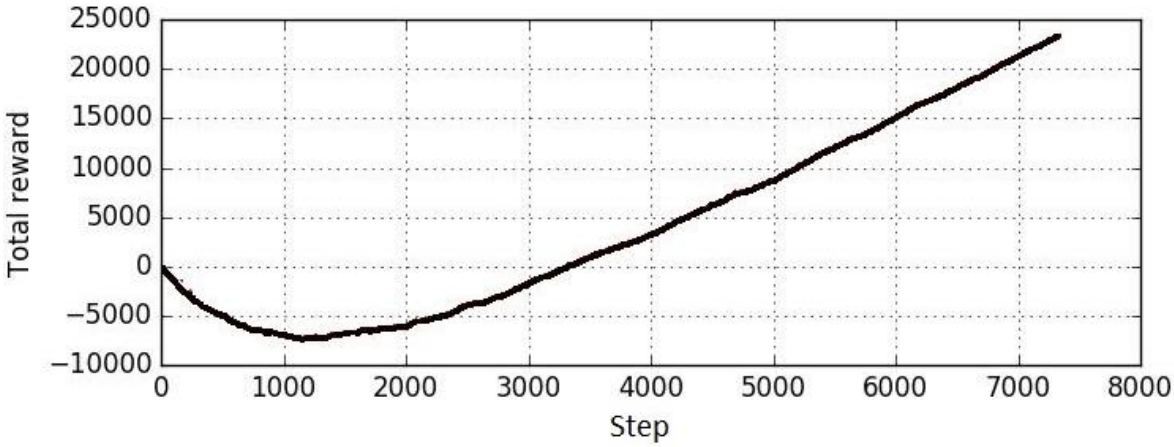

Abbildung 51 Trainingsverlauf aus vorherigem Trainingsversuch

Hierbei wurde anhand von gesammelten Datensätzen 100-mal pro Schritt trainiert. Der Agent hat bereits nach ca. 120000 Trainingsdurchläufen das Bewegungsmuster so weit erlernt, sodass im weiteren Verlauf stets die Gesamtbelohnung erhöht wurde. Bei Erstellung dieses Modells wurde die Epsilon Greedy Strategie nicht implementiert, da dies bereits beim Sammeln der Daten geschah.

5 Fazit

In Rahmen dieser Bachelorarbeit wurde eine Reinforcement Learning Umgebung für einen Quadruped-Roboter entwickelt. Der Roboter hatte dabei die Aufgabe ein Bewegungsmuster zum Navigieren innerhalb einer physikalisch simulierten Umgebung zu erlernen. Die physikalische Umgebung wurde dabei mittels Simulationssoftware namens „Gazebo“ erstellt. In dieser Umgebung fand zudem das Training des Agenten statt. Das Modellieren des Vierbein-Roboters, seiner Bewegungen und physikalischen Eigenschaften wurde mit Hilfe des Robotik-Frameworks ROS durchgeführt. Die Reinforcement Learning Umgebung wurde mittels dem Framework OpenAI Gym und dem Zusatzpaket Openai_ros erstellt. Zum Erlernen des Bewegungsmusters wurde das DQN-Algorithmus inklusive der Verbesserungsmethoden Experience Replay Speicher und Target Network verwendet. Dazu wurde ein Belohnungssystem entwickelt anhand dessen der Roboter seine Aktionen ausgewählt hat. Abschließend wurde ein Modell erstellt und getestet.

Die Hauptaufgabe war es eine Reinforcement Learning Umgebung zu einwickeln, in der ein Vierbein-Roboter eine bestimmte Aufgabe erlernen kann. Die Einzelkomponenten dieser Umgebung wurden in einzelnen Funktionen und Klassen erstellt, sodass diese modular angepasst oder ausgetauscht werden können. Innerhalb der Umgebung ist es nun möglich, mit Hilfe der Belohnungsfunktion dem Agenten eine Aufgabe vorzugeben. Durch die Funktionen der Lernumgebung können die Bewegungsbereiche des Roboters verändert werden, um den Zustandsbereich zu erhöhen oder eine andere Bewegungsart umzusetzen.

Die Aufgabe des Agenten war es ein Bewegungsmuster zum Bewegen und Navigieren zu einem beliebigen Zielpunkt innerhalb einer Simulationsumgebung zu erlernen. Für die ersten beiden Szenarien konnte der Agent stets jedes Ziel anvisieren und sich durch das erlernte Bewegungsmuster innerhalb der vorgegebenen Schrittzahl bewegen. Das Bewegungsmuster wurde jedoch nur bedingt richtig umgesetzt. Wie im vorherigen Kapitel beschrieben, wurden einige Belohnungskriterien nicht richtig umgesetzt. Zum einen kann das an zu wenigen Datensätzen und zu kurzem Training liegen. Zum anderen müssen die Kriterien der Belohnungsfunktion vor allem bei den Verschiebungsaktionen weiter verfeinert werden. Im dritten Szenario konnte der Agent zwar alle Ziele anvisieren, konnte aber nicht alle Episoden erfolgreich beenden. Hierbei haben deutlich mehr Datensätze gefehlt. Aufgrund des Zeitfaktors innerhalb der Simulation dauert das Ausführen von Aktionen relativ lange. Dadurch erstreckt sich das Sammeln von Daten schon für wenige Zehntausende Datensätze über mehrere Tage. Zusammenfassen lässt sich sagen, dass der Agent nur die wesentlichen Kriterien zum Erreichen des Ziels erlernt hat. Diese sind das Verringern der Distanz und der Ausrichtung. Zusätzliche Kriterien zum Verbessern der Bewegungsart wurden nicht richtig umgesetzt. Diese benötigen eine Weiterentwicklung der Belohnungsfunktion und das optimale Einstellen der Simulationsumgebung.

Die größten Schwierigkeiten im Laufe der Arbeit bereitete die Parametrisierung der physikalischen Simulationsumgebung inkl. des Robotermodells sowie das entwickeln einer passenden Belohnungsfunktion. Bei der Parametrisierung der Simulationsumgebung war es nicht möglich mittels realer Reibungswerte ein annähernd reales Reibverhalten zwischen den vielen Komponenten einzustellen. Aus diesem Grund mussten Pseudowerte eingestellt werden, um das Modell und die Korrelation der Kräfte zu stabilisieren. Das Modellieren der Belohnungsfunktion war insofern schwierig, da Veränderungen durch eingestellte Parameter erst nach langem Sammeln der Daten und anschließendem Training sichtbar waren. Dies machte es zu einem langen Prozess.

Die Simulationssoftware Gazebo und die dazugehörige Physik-Engine ODE sind bei einer optimalen Parametrisierung für Anwendungen dieser Art durchaus geeignet.

Zudem hat diese Software den großen Vorteil einer Open-Source-Lizenz und die ständige Weiterentwicklung durch die Community. Das Software-Framework ROS ist beim Erstellen von Robotermodellen ein sehr hilfreiches Werkzeug. Dadurch lassen sich wie im Falle des Atlas-Roboters von Boston Dynamics sehr komplexe Robotermodelle entwickeln.

6 Aussichten

Im letzten Kapitel werden einige Aussichten und Verbesserungsmöglichkeiten dieses Projektes präsentiert. Es wird zunächst auf mögliche Verbesserungen innerhalb der erstellten Arbeit hingewiesen. Daraufhin werden Methoden erwähnt, welche die jetzigen Methoden ergänzen könnten. Abschließend werden mögliche fortführende Arbeiten vorgeschlagen, die auf Grundlage dieses Projektes aufbauen könnten.

6.1 Physikalische Umgebung

Innerhalb der erstellten Arbeit wäre es sinnvoll die Parameter der physikalischen Simulation weiter zu optimieren. So würden bei einer optimalen Parametrisierung keine unerwünschten Nebeneffekte wie das Gleiten der Beine, das Übersteuern der Regler oder Verschieben eines Körpers durch eine Beinbewegung entstehen. Dadurch werden die Belohnungen nicht durch falsche physikalische Eigenschaften verfälscht. Des Weiteren kann ein anderer Controller zum Bewegen der Gelenke einsetzten werden. Der verwendete Controller erlaubt es nur Winkelpositionen nacheinander zu verstellen. So könnte man einen Trajektorien-Controller einsetzten, der gleichzeitig mehrere Gelenke entlang einer Trajektorie bewegt, wobei die vorgegebenen Punkte dieser Trajektorie zu bestimmten Zeiten angesteuert werden. Dadurch ließe sich eine besser gesteuerte und flüssigere Bewegung implementieren. Dies würde die Stabilität des Modells erhöhen. Des Weiteren muss das Belohnungssystem erweitert werden. So würde es sich anbieten eine zusätzliche Abfrage über den Schwerpunkt des Roboters zu erstellen. Dies ließe sich durch den vorhandenen Odometer-Plugin

erstellen. Erweitert man diesen auf die vier Stützflächen, so kann man innerhalb der Simulationsumgebung eine Fläche berechnen. Berechnet wird diese durch alle zu einem Zeitpunkt auf dem Boden stehenden Stützflächen. Auf diese Weise ließen sich zusätzliche Kriterien zum überwachen des Schwerpunktes erstellen. Um das Training zu beschleunigen, müsste ein Zeitfaktor zwischen der Simulationsumgebung und der realen Zeit ermittelt werden. Dabei muss zwischen der Anzahl an Berechnungen pro Zeiteinheit und den Zeitschritten ein passendes Gleichgewicht gefunden werden. Dadurch ließe sich die Ausführzeit der Aktionen innerhalb der Simulation beschleunigen. Der Einsatz von virtuellen Maschinen hat aus Erfahrung nur bedingt funktioniert, da Gazebo in einem virtuellen Betriebssystem oft abstürzt.

6.2 DQN Verbesserungsalgorithmen

Neben dem Verbessern und Hinzufügen weiterer Schichten zu den Netzwerken existieren für den DQN-Algorithmus weitere Verbesserungsmethoden, die in dieser Arbeit nicht implementiert worden sind. Diese Methoden verkürzen die Trainingsdauer und erhöhen gleichzeitig die Genauigkeit des Netzwerkes.

Double DQN

Diese Methode beschäftigt sich mit der Frage, ob die nächste bestmögliche Aktion, die mit dem größten Q-Wert ist. Vor allem am Anfang des Trainings ist die Auswahl anhand der größten Q-Werte meist falsch. Dazu wird bei der Berechnung des Q-Wertes die Aktionsauswahl und die Berechnung voneinander abgekoppelt. Das DQN-Netzwerk wählt anhand des berechneten Q-Wertes die beste Aktion zum Gelangen in den nächsten Zustand. Das Target-Netzwerk generiert die Q-Werte und das DQN-Modell entscheidet, welches davon verwendet werden soll. Es hilft somit die Überschätzungen von Q-Werten zu reduzieren und das Training zu beschleunigen.

Dueling DQN (DDQN)

Bei diesem Verfahren wird der Q-Wert in V(s) und A(s, a) aufgeteilt. Das V(s) beschreibt dabei den Vorteil sich in diesem Zustand zu befinden. Das A(s, a) beschreibt den Vorteil beim Ausführen einer Aktion gegenüber anderen. Dazu werden

mit Hilfe des DDQN innerhalb des Netzwerkes die beiden Werte separat voneinander ermittelt (siehe Abb. 49). Anschließend werden die beiden Werte mittels spezieller Schichten wieder zusammengeführt und ergeben somit den neuen Q-Wert. Durch diese Aufteilung kann das Netzwerk lernen, welche Zustände wertvoll sind, ohne die Auswirkung jeder Aktion in jedem Zustand lernen zu müssen. Dadurch ist es nicht mehr erforderlich jeden Wert einer Aktion zu berechnen

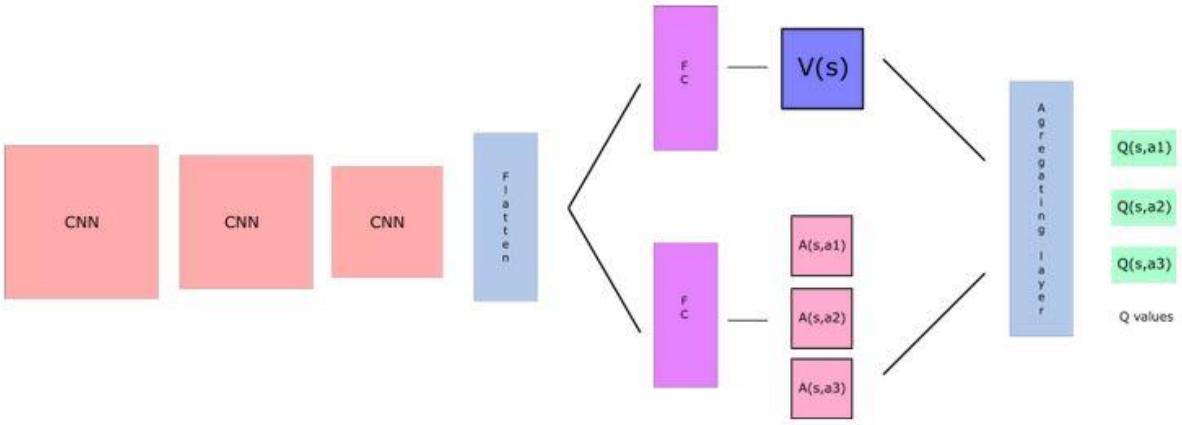

Abbildung 52 Netzarchitektur bei Dueling DQN Simonini (2018)

Prioritized Experience Replay

Diese Methode folgt der Tatsache, dass einige Datensätze im Laufe des Trainings eine höhere Wichtigkeit erlangen. Dies geschieht durch das zufällige Entnehmen der Datensätze aus dem Experience Replay Speicher. So werden einige Datensätze öfters ausgewählt als andere. Bei einer stets wachsenden Datensammlung wächst die Wahrscheinlichkeit nicht ausgewählt zu werden umso mehr. Aus diesem Grund wird versucht die Verteilung der Daten zu ändern, indem die Priorität jedes Datensatzes anhand eines Kriteriums ausgewählt wird. Dazu wird eine binäre Baumstruktur geschaffen, in welcher die Prioritätswerte der Datensätze hinterlegt werden. Bei jeder Aktualisierung des Target-Netzwerks werden die Prioritäten der Datensätze ebenfalls aktualisiert. Die Datensätze werden dann hinsichtlich ihrer Priorität hochgestuft.

6.3 Fortführende Arbeit

Des Weiteren wäre es interessant zu testen, wie sich ein in einer Simulationsumgebung trainierter Agent in einer realen Umgebung verhalten würde. Mit Hilfe von ROS lassen sich die erstellten Projektpakete auf Mikrocontroller wie Arduino und kleinere Computersysteme wie Raspberry Pi übertragen. Die realen Robotermodelle lassen sich ebenfalls über Mikrocontroller steuern. Die Kommunikation zwischen dem System und der Hardware läuft dank ROS weiterhin über ROS-Nodes und Topics ab. Dadurch können gesamte Arbeitsumgebungen wie Lagerhallen in Gazebo simuliert und als Lernumgebung zum Trainieren eines Systems eingesetzt werden. Die trainierten Systeme ließen sich anschließend direkt ohne Nacharbeit auf reale Robotermodelle übertragen. Analog zu kleineren Robotermodellen, die simple Aufgaben lernen auszuführen, wäre es möglich autonome Fahragenten für den Straßenverkehr bereits innerhalb einer Simulation hinreichend genau zu entwickeln und zu testen.

7 Quellenverzeichnis

[Barto 2017] Sutton, Richard S.; Barto, Andrew G.: Reinforcement Learning: An Introduction, A Bradford Book, 5. Nov. 2017

[Meisel 2012] Meisel, Andreas: Neuronale Netze, Vorlesung Robotvision

[Sagar 2017] Sharma, Sagar: Activation Functions in Neural Networks, 6. Sep. 2017

https://towardsdatascience.com/activation-functions-neural-networks-1cbd9f8d91d6 (10. Apr. 2019)

[Heinz 2018] Heinz, Sebastian: Wie lernen Neuronale Netze, 26. Feb. 2018

https://www.statworx.com/de/blog/wie-lernen-neuronale-netze/ (10. Apr. 2019)

[Mazur 2015] Mazur, Matt: A Step by Step Backpropagation Example , 17 März. 2015

https://mattmazur.com/2015/03/17/a-step-by-step-backpropagation-example/ (10. Apr. 2019)

[Lentin 2017] Lentin, Joseph: ROS Robotics Projects, Packt Publishing Ltd., März.2017

[Josh 2016] Elijah, Josh: How to Program a Quadruped Robot with Arduino, 22. Nov. 2016

https://makezine.com/2016/11/22/robot-quadruped-arduino-program/ (10. Apr. 2019)

[Ritesh 2018] Waghe, Ritesh G.; Bhoyar, Deepak; Ghormade, Sagar: A Real Time Design and Implementation of Walking Quadruped Robot for Environmental Monitoring, May. 2018

https://www.irjet.net/archives/V5/i5/IRJET-V5I541.pdf (10. Apr. 2019)

[Simonini 2018] Simonini, Thomas: Improvements in Deep Q Learning, 5. Jul 2018

https://medium.freecodecamp.org/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682 (10. Apr. 2019)

Erklärung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe.

Hamburg, 11. April 2019 ……
Roman Jungblut