Deep Reinforcement Learning-Ansatz in der Robotik

Anwendung des DRL-Ansatzes in einer Robotersimulation im Unity-Editor

Bachelor Arbeit

Eingereicht in teilweiser Erfüllung der Anforderungen zur Erlangung des akademischen Grades:

Bachelor of Science in Engineering

an der FH Campus Wien

Studienfach: Computer Science and Digital Communications

Autor:

Darius Vasiu

Matrikelnummer:

C1810475056

Betreuer:

DI Dr. techn. Mugdim Bublin

Datum:

03.06.2021

Erklärung der Urheberschaft:

Ich erkläre hiermit diese Bachelor Arbeit eigenständig verfasst zu haben. Ich habe keine anderen Quellen, als die in der Arbeit gelisteten verwendet, noch habe ich jegliche unerlaubte Hilfe in Anspruch genommen

Ich versichere diese Bachelor Arbeit in keinerlei Form jemandem Anderen oder einer anderen Institution zur Verfügung gestellt zu haben, weder in Österreich noch im Ausland.

Weiters versichere ich, dass jegliche Kopie (gedruckt oder digital) identisch ist.

Datum: 03.06.2021 Unterschrift:

Abstract

This paper examines the results of different parameters applied to a simulated robot in a DRL network. The results of four different categories and nine different training passes are considered. In addition, the best training are compared, interpreted and evaluated to the result of a training with default parameters set by Unity. Three different approaches to robotics are explained with a focus on the AI approach, more specifically the DRL approach. To bridge the gap to the topic of artificial intelligence, this subject area and the most important methods included in it, such as biological and artificial neural networks, the three types of learning Supervised, Unsupervised and Reinforcement Learning, are taken up and described.

Kurzfassung

Diese Arbeit untersucht die Ergebnisse verschiedener Parameter, die in einem DRL-Netz an einem simulierten Roboter angewendet werden. Die Ergebnisse von vier verschiedenen Kategorien und neun verschiedenen Trainingsdurchgängen werden betrachtet. Zusätzlich werden die besten Trainingsdurchgänge mit dem Ergebnis eines Trainingsdurchgangs mit von Unity festgelegten Default-Parametern verglichen, interpretiert und ausgewertet. Drei verschiedene Ansätze der Robotik werden mit Fokus auf den KI-Ansatz, genauer dem DRL-Ansatz, erläutert. Um die Brücke zum Thema der künstlichen Intelligenz zu schlagen, wird dieses Themengebiet und die wichtigsten darin enthaltenen Verfahren wie etwa biologische und künstliche neuronale Netze, die drei Arten des Lernens Supervised, Unsupervised und Reinforcement Learning aufgegriffen und beschrieben.

Abkürzungen

DRL	Deep Reinforcement Learning
DQL	Deep Q-Learning
KI	Künstliche Intelligenz
MDP	Markov Decision Process
ML	Machine Learning

Schlüsselbegriffe

Künstliche Intelligenz
Robotik
Industrie 4.0
ML-Agents
Reinforcement Learning
Unity

Inhaltsverzeichnis

1 Einführung **1**
1.1 Motivation . . . 1
1.2 Ziel . . . 2
1.3 Gliederung . . . 2

2 Künstliche Intelligenz **3**
2.1 Schwache und starke KI . . . 5
2.2 Machine Learning . . . 6
2.3 Deep Learning . . . 9
2.4 Neuronale Netze . . . 10

3 Robotik **14**
3.1 Klassischer Ansatz . . . 15
3.2 Deep Reinforcement Learning Ansatz . . . 16
3.3 Regelbasierter Ansatz . . . 20

4 Unityprojekt: Roboter mit Reinforcement Learning Modell **22**
4.1 Ziel des Projekts . . . 22
4.2 Tools . . . 22
4.3 Parameter, Belohnungssystem und Roboter . . . 25
4.4 Ergebnisse . . . 29

5 Zusammenfassung & Ausblick **35**

Bibliographie **36**

Abbildungen **38**

Tabellen **39**

1 Einführung

Die industriellen Revolutionen, die die Welt miterlebte, zwangen alle Beteiligten zum Handeln. In der ersten Revolution hat das bis dahin vorhandene Wissen vom Felde für viele nicht mehr gereicht, um mit der modernen industriellen Gesellschaft mithalten zu können. Es wurden neue Fähigkeiten in den Fabriken benötigt, sowohl manuelle, technische als auch in der Managementebene. Die Konkurrenz konnte fast über Nacht das Vielfache produzieren und somit anderen Fabriken in finanzieller Hinsicht davonlaufen. Alles ging schneller vonstatten, vor allem in der zweiten industriellen Revolution. Durch die Elektrotechnik konnte vieles modernisiert und teilweise automatisiert werden, was wiederum dazu führte, dass sich Arbeiter weiterbilden mussten um ihren Job zu behalten oder sich anderswo bewerben zu können. Massenproduktionen überschwemmten das Land und drückten die Preise der Waren. Der Aufschwung der westlichen Welt war im Gange. Mit der dritten Revolution ist vieles einfacher geworden, vieles aber umso komplexer. Das Verstehen der Materie und den erfolgreichen Einsatz der Digitaltechnik bedeutete auch, dass Experten und Spezialisten benötigt wurden. Wiederum mussten viele um den Job bangen, wenn das vorhandene Wissen nicht mehr ausreichte. Unternehmen konnten ihre Waren besser und schneller produzieren, vermarkten und verkaufen. Dies alles führt uns zu unserer Zeit, in der wir warten, dass die vierte Revolution beginnt, falls wir denn nicht schon mittendrin sind - am Anfang der Revolution. Roboter, die in Fabriken vereinsamt vor sich hin produzieren gibt es bereits, weitere solcher dunklen Fabriken sollen folgen. Arbeiter und Angestellte müssen sich weiter spezialisieren und die 'richtige' Ausbildung für das neue Zeitalter wählen, wenn sie sicher sein wollen, dass auch in Zukunft ihr Job gesichert ist. Ein großer Teil dieser Gefahr oder Chance, je nach Betrachtungsweise des Themas, wurde durch die Forschung und Entwicklung der künstlichen Intelligenz angestoßen. Viele der heutigen Jobs werden in einigen Jahren nicht mehr sein, viele neue werden entstehen. Heutige Roboter die im Einsatz sind, welche taktgenau Teile zusammenschrauben oder -schweißen, lackieren oder auseinander schneiden, wurden von Programmierern für eine bestimmte Tätigkeit programmiert und werden immer wieder neu programmiert werden müssen, zum Beispiel im Bereich des CNC-Fräsens. Hier ist bereits der Punkt, an dem diese Arbeit ansetzt. Roboter können bereits viele Arbeiten selber erlernen, optimale Lösungen finden um ein Ziel zu erreichen. Strategien wurden entwickelt um diese Technologie zu optimieren, denn ein Roboter oder eine CNC-Maschine ist ein teures Werkzeug und muss behutsam bedient werden. Wenn es autonom agieren soll, muss es sich selber gefahrlos bedienen können. Diese 'intelligenten' Roboter, also die Verbindung der Robotik mit der künstlichen Intelligenz, sind ein Teil der vierten industriellen Revolution, welche der Menschheit noch ungewisse Chancen und Risiken mitbringen werden.

1.1 Motivation

Die Möglichkeit, ein Projekt mit den nach eigener Meinung wichtigsten zwei Teilbereichen der vierten industriellen Revolution, der KI und der Robotik, ist die größte treibende Kraft der Motivation. Das Verständnis für diese Technologien aufzubauen und die Risiken und Gefahren zu verstehen, machen das Projekt umso spannender. Die heutige Industrie wird sich in gewis-

ser Hinsicht ändern, auf kurz oder lang. Dieses Projekt soll auch das persönliche Verständnis der Materie vertiefen, um die zukünftigen Umwälzungen zu verstehen und nachvollziehen zu können. Ebenfalls in Hinsicht auf die eigene Weiterbildung und auf die Möglichkeit der Wissensweitergabe zur Unterstützung jüngerer Generationen, ist das hier angeeignete vernetzte Wissen hilfreich.

1.2 Ziel

Ziel dieser Arbeit ist es, verschiedene Strategien zu untersuchen, die einen optimalen Weg zur Lösung finden. Ein Roboter soll durch Training lernen, ein Objekt autonom erreichen zu können. Durch verschiedene Strategien die durch Parameter angepasst werden, sollen Ergebnisse der Trainingsdurchläufe verglichen und interpretiert werden. Zum besseren Verständnis der Materie, werden zusätzlich drei Ansätze der Robotik erläutert und zusammengefasst.

1.3 Gliederung

Die Arbeit gliedert sich in vier große Bereiche. Im Kapitel 'Künstliche Intelligenz' werden die Begriffe und die Teilbereiche der KI erleuchtet. Ebenfalls wird in diesem Kapitel die Brücke geschlagen zwischen biologischen und neuronalen Neuronen und Netzen. Im Kapitel 'Robotik' werden drei Ansätze der Robotik erläutert, um einen Überblick über dieses Themengebiet zu verschaffen. Im Kapitel 4 werden das Projektziel, der Roboter und die Werkzeuge, die für das Projekt notwendig sind, die Konfigurationen des DRL-Modells und das Belohnungssystem beschrieben sowie die Ergebnisse gezeigt und interpretiert.

2 Künstliche Intelligenz

Um den Begriff der künstlichen Intelligenz zu verstehen, muss erst verstanden werden, was das Wort 'Intelligenz' im Grunde bedeutet. Was ist 'Intelligenz'? Wie kommt diese zustande und ab wann gilt ein Subjekt als intelligent? Es stellt sich weiter die Frage, ob die Intelligenz automatisch auch ein Bewusstsein mit sich bringt. Diese Frage kann ausgeweitet werden auf das Bewusstsein. Was ist das Bewusstsein? Ist ein Wesen, welches selbstständig agieren kann sich selbst bewusst? Diese Frage wiederum dringt bereits tief in die Psychologie ein. Ein Hund, ein Schwarm Vögel oder auch ein Ameisenvolk kann in gewisser Hinsicht als intelligent bezeichnet werden, doch haben diese Wesen auch ein Bewusstsein? Was ist mit einem Käfer, einer Gelse oder gar einem Bakterium? Auch hier könnte man sagen, dass es eine gewisse Intelligenz gibt, wenn auch das Verhalten mit elektrochemischen Abläufen erklärt werden könnte. Ist ein Virus intelligent, wenn man bedenkt, dass dieses die ganze Welt im Zaum halten kann? Wo kann die Grenze gezogen werden in Bezug auf Intelligenz und Bewusstsein? Wenn nun die Wissenschaft es schafft, eine Superintelligenz zu erschaffen, hat dieses bereits ein Bewusstsein?

Es zeigt sich, die Frage der Intelligenz und des Bewusstseins ist nicht so einfach zu beantworten. Die Betrachtung der Kombination dieser zwei Begriffe ist daher wichtig, da es zumindest beim Menschen Hand in Hand geht, auch wenn nicht unbedingt gekoppelt. Ob dies für das gesamte Tierreich oder nur für einen Teil gilt, ist zurzeit noch nicht geklärt und das Sammeln von wissenschaftlichen Fakten wird womöglich noch einige Zeit in Anspruch nehmen. In der wissenschaftlichen Literatur ist die Antwort auf diese Frage umstritten. Denn, Intelligenz ist nicht messbar. Oft wird ein IQ-Test als Indikator für Intelligenz gewertet, was aus wissenschaftlicher Sicht kritisch zu betrachten ist. Messbar ist die Intelligenz definitiv nicht, zumindest nicht die biologische. Dass sich das Bewusstsein im Hirn entwickelt ist wissenschaftlich erforscht, wenn auch noch nicht ganz klar ist, wo es genau entsteht, Hirnscans können aus dem Hirntomografen können als Beweis betrachtet werden [RN12].

Der Begriff 'künstliche Intelligenz', wie John McCarthy diese Technologie als erster nannte, wurde während eines Workshops im Sommer 1956 am Dartmouth College das erste Mal offiziell erwähnt, als einige Wissenschaftler sich mit dem Thema befasst haben. Russell und Norvig hätten hierfür jedoch lieber 'berechnende Rationalität' [RN12] gewählt, da dieser Begriff die tatsächliche Arbeitsweise dieser Technologie genauer bezeichnet.

Eine Intelligenz löst ein gestelltes Problem. Um zur Lösung zu gelangen bedarf es ein Ziel. Wie kann ein Ziel erreicht werden, wenn es viele Wege gibt, die zu diesem führen? Man wählt den einfachsten und schnellsten Weg. Hier entwickelt sich bereits ein Algorithmus. In gewisser Hinsicht gab es die Idee der künstlichen Intelligenz bereits vor mehr als 2.000 Jahren, nur wurde der Begriff Jahrhunderte später erst erfunden. In der Nikomachischen Ethik beschreibt Aristoteles bereits eine Art Algorithmus. Der erste mathematische Algorithmus jedoch (der Begriff geht zurück auf den Namen des arabischen Mathematikers al-Chwarizmi), wurde von Euklid von Alexandria beschrieben - den euklidischen Algorithmus, der den größten gemeinsamen Teiler zweier natürlichen Zahlen berechnet.

Turing Test

Um nun eine künstliche Intelligenz auf die tatsächliche Intelligenz prüfen zu können, wurde von Alan Turing 1950 ein Test entwickelt. Dieser Test dient dazu, die Intelligenz eines Computers zu belegen. Hierfür führt Computer A und Person B mit Person C eine Konversation, eine Frage-Antwort-Konversation. Der Computer hat die Aufgabe, der Person C die Unterscheidung zwischen Mensch (Person B) und Maschine (Computer A) unmöglich zu machen. Um dem Computer eine faire Chance zu geben, darf die Kommunikation ausschließlich elektronisch erfolgen, über Maus und Tastatur. Wenn Person C nicht entscheiden kann, welcher dieser zwei Gesprächspartner die Maschine ist, ist der Test bestanden. [Tur50] Dieser Test war so ausgeklügelt und überlegt, dass er heute noch aktuell ist und in gewisser Hinsicht als der ultimative Test angesehen wird, um eine künstliche Intelligenz zu prüfen, ob diese tatsächlich eine Intelligenz entwickelt hat. Natürlich gibt es heute bereits in gewissen Bereichen KIs, die in Konversationen den Kommunikationspartner täuschen können. Ein Beispiel dafür ist Google Duplex [O'L19]. Nichtsdestotrotz gibt es kritische Stimmen, die diesen Test als veraltet oder als nicht ausreichend bezeichnen, daher gibt es bereits verbesserte Tests, wie den Lovelace-Test (KI muss Kreativität beweisen [BBF00]) oder den Metzinger-Test (KI muss in einer Diskussion argumentieren können [Met01]).

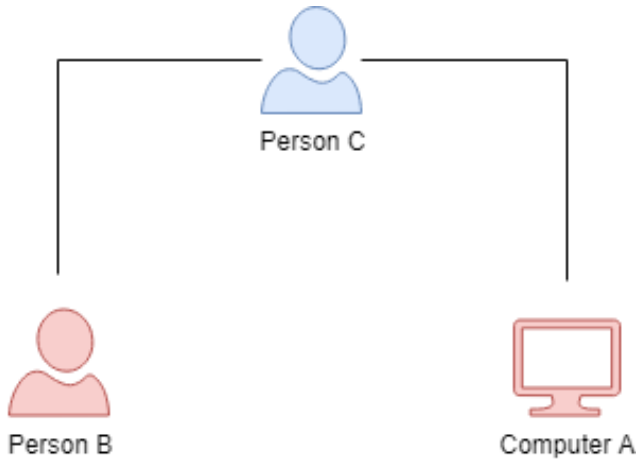

Abbildung 2.1: Veranschaulichung Turing Test [Tur50]

Technologische Singularität

Der Begriff der technologischen Singularität beschreibt den Zeitpunkt, an dem die Menschheit die letzte und wichtigste Erfindung entwickeln wird, nämlich die Superintelligenz. Diese Superintelligenz würde dann das Erfinden übernehmen und uns Menschen, je nach Betrachtungsweise entweder vom Intelligenzthron stoßen, oder aber sie würde dem Menschen den Frieden auf Erden bringen. So oder so, würde es einen gewaltigen Umbruch mit sich bringen. Der Umstand, dass sich die Rechenleistung der Computer in immer kürzeren Zeitabständen verdoppelt, wie das Mooresche Gesetz von Gordon Moore prophezeite [Moo06], führt zu der Annahme, dass dieser Zeitpunkt in naher Zukunft eintreten wird. Nach Raymond Kurzweil, wäre dieser Zeitpunkt im Jahre 2045 [Kur06]. Das wäre demnach auch der letzte durchgeführte Turingtest - der auch ohne Zweifel bestanden werden wird.

2.1 Schwache und starke KI

Künstliche Intelligenz kann unterschieden werden in schwacher und starker KI. Infolgedessen ist die Unterscheidung von Bedeutung, da es uns zurzeit eine klare Linie vorgibt, was im möglichen und im (noch) unmöglichen Bereich liegt. Auch wenn hier von einer klaren Linie die Rede ist, ist sie in der theoretischen Forschung und in der sich damit befassenden Philosophie je nach Betrachtungsweise teils verschwommen. Denn: es gibt sehr wohl bereits Modelle, die einen Turingtest erfolgreich absolvieren können, jedoch ist hier nicht die Rede von einer starken KI. Laut Turing beziehungsweise seinem Test aber, wäre es sehr wohl der Beweis einer tatsächlichen Intelligenz. Beispiele für eine schwache KI, die einen Turingtest bestehen könnte, wäre Googles Duplex [O'L19] und die Reservierung eines Tischs in einem Restaurant [LM18].

Ein weiterer erfolgreicher Test wäre auch das chinesische Zimmer: eine Person, die kein Chinesisch beherrscht befindet sich in einem Zimmer. In diesem Zimmer gibt es ein Buch mit Anweisungen und leere Stücke Papier. Durch einen Schlitz werden von außen Stücke Papier mit Symbolen welche die Person nicht versteht in das Zimmer hineingeworfen. Nun sucht die Person das dazu passende Symbol aus dem Buch mit Anweisungen, schreibt dieses auf ein leeres Stück Papier und gibt es zurück durch den Schlitz. Ein Außenstehender würde dieses System wahrscheinlich als intelligent bezeichnen, denn die Idee liegt nahe, dass auf dem hineingeworfenen Papier eine Frage geschrieben steht und auf dem zurückgegebenen Papier die Antwort darauf. Jedoch ist es in Wahrheit weder intelligent (wenn auch ausgeklügelt) noch kann die Person im Zimmer Chinesisch verstehen, was sie selbst aufschreibt [RN12].

2.1.1 Schwache KI

Der erste Teil der künstlichen Intelligenz ist die schwache KI. Diese Art von KI treffen wir heutzutage an verschiedenen Stellen an. Schwache KI-Systeme sind alle Systeme, die sich zurzeit auf dem Markt befinden, auch diese die nicht für die Allgemeinheit gedacht sind, zum Beispiel das autonome Navigationssytem, welches das US-Militär in ihren Drohnen verwendet. Aber auch der Empfehlungsalgorithmus von Google, Spotify oder Netflix, oder Algorithmen die Krebs frühzeitig erkennen können oder Alexa, Cortana oder Siri - alle fallen unter dem Schirm der ßchwachen KI". Diese Systeme haben alle eines gemeinsam: die Fähigkeiten beschränken sich auf wenige bis einzelne Anwendungsbereiche. Eine Spracherkennungs-KI kann keinen Saugroboter steuern oder ein Fahrzeug autonom fahren. Die Kombination dieser Systeme würde eine stärkere schwache KI erzeugen, jedoch keine starke KI.

Zusammenfassend kann man sagen, dass eine schwache KI für eine konkrete Aufgabe erstellt, programmiert oder trainiert wurde.

2.1.2 Starke KI

Das Thema der starken KI kann nicht ohne philosophische Gedanken auskommen. Wie am Anfang der Arbeit erwähnt, ist die Intelligenz gekoppelt an das Gehirn, das Gehirn gekoppelt an das Bewusstsein und das Bewusstsein wiederum an der Intelligenz. Ob diese drei Begriffe nur gemeinsam genannt werden können, zumal alles Eins ist oder ob es sich um drei verschiedene Aspekte eines Wesens handelt ist eine tiefe philosophische Frage und kann nicht mit Gewissheit beantwortet werden, daher ist diese Frage auch nur subjektiv zu beantworten. Tatsache ist: wenn die Menschheit es schafft, eine starke KI zu erzeugen, könnte es bedeuten, dass wir auch Bewusstsein schaffen können. Weiters würde auch die Frage nach dem "Woher"wieder auflodern, denn: wenn wir es schaffen, ein Bewusstsein zu erschaffen, dann

könnten wir ebenfalls von einem anderen Bewusstsein erschaffen worden sein. Hier würde auch die Simulationstheorie einige Fragen beantworten können. Auch der Descarte'sche Dämon [Des08], der noch ein Gedankenkonstrukt war, hätte durch unser Handeln die Realität kennengelernt.

Zusammenfassend kann man sagen, dass eine starke KI eine Intelligenz ist, die mit der menschlichen gleichzusetzen ist, oder diese sogar übertrifft. Siehe auch Kapitel der technologischen Singularität.

2.2 Machine Learning

Machine Learning ist eine Teilmenge der künstlichen Intelligenz. Die Teilbereiche 'supervised learning', 'unsupervised learning' und 'reinforcement learning' werden in den Unterkapiteln näher beschrieben. Es gibt viel mehr Arten von Machine-Learning-Systemen, die hier nicht näher betrachtet werden, da es sonst den Rahmen bei weitem sprengen würde, nur als Beispiel: die Unterschiede von instanzbasiertem und modellbasiertem Lernen oder den Unterschied zwischen Batch- und Online-Learning [Ger17].

Die Technologie des maschinellem Lernens zielt darauf ab, einer Maschine oder genauer, einem Computer das Lernen beizubringen, ohne dass dieser darauf programmiert wurde zu lernen. Was noch vor ein paar Jahrzehnten nur in der Science Fiction Platz fand, findet man heutzutage in jedem erdenklichen Programm. Ob es der Suchalgorithmus von Google ist, der Spamfilter von Microsoft im E-Mail-Programm Outlook oder dem Vorschlagalgorithmus von Spotify oder Netflix - Machine Learning treffen wir jeden Tag an, bewusst oder unbewusst. Die Möglichkeiten, die sich hier anbieten sind schier unendlich. Doch auch die gewagtesten und optimistischsten Ideen brauchen Zeit, denn, je komplexer das Problem, desto länger dauert das Training. Auch das Training muss immer wieder durchlaufen und angepasst werden, wenn es das Ziel aus den Augen verliert, zum Beispiel durch das Problem des Over- oder Underfitting. Das bedeutet nichts anderes, als das ein System zu sehr oder zu wenig auf die Trainingsdaten fokussiert und trainiert wurde. Mit den Trainingsdaten erzielt es herausragende Ergebnisse, aber es kann sich nicht an andere Daten anpassen (Overfitting). Andersherum wird das Training vorzeitig beendet um das Overfitting zu unterbinden, doch diese Aktion löst das Problem aus, dass unser System unzureichend trainiert wurde und zu schlechteren Ergebnissen führt (Underfitting) [Ger17]. Ein gutes System soll generalisiert werden können, was Overfitting unterbindet, daher sollte man das unbedingt vermeiden. Hier ist eine Balance notwendig, die nur durch mehrere Testdurchläufe erreicht werden kann, um die Parameter so zu wählen, damit das System generalisierbar wird, was jedoch Zeit und Ressourcen verbraucht.

2.2.1 Supervised Learning

Das Prinzip des überwachten Lernens bzw. des supervised learning funktioniert mit Informationen über den Input und den Output. Hierfür werden Input/Output-Paare erstellt und diese werden dem System zugeführt um aus diesen zu lernen. Durch das Erkennen von Mustern und Beispielen wird das System trainiert und kann später auf neue Daten angewandt werden und dafür Prognosen abgeben. Beispiele für die Anwendung wäre ein Spamfilter. Der Filter wird durch das automatische oder händische Markieren (labelling) trainiert, indem es lernt, Mails von einem bestimmten Absender, welche mehrmals als Spam markiert wurden, automatisch in den Spamordner zu verschieben, sobald eine neue Mail von diesem Absender eintrifft [Ger17]. Die Möglichkeiten oder Arten des supervised learning, sind die Klassifikation und

die Regression. Für jede Art werden Daten benötigt die im Voraus vorbereitet und selektiert werden müssen. Der Erfolg des Trainings ist abhängig von der Qualität des Trainingssatzes und daher von großer Bedeutung, mehr dazu im Unterkapitel 'Trainingsdaten'.

Klassifikation

Das oben genannte Spamfilter-Beispiel ist ein Klassifikationsbeispiel. Es wird versucht zu klassifizieren um so zukünftige Ereignisse in Klassen oder Kategorien packen zu können. Ein klassisches Beispiel ist der MNIST-Datensatz, hier sind handgeschriebene Ziffern von 0 - 9 klassifiziert. Somit kann damit ein System auf das Erkennen von Ziffern trainiert werden.

Regression

Das Ergebnis einer Regression ist eine Zahl, eine Wahrscheinlichkeit für das Auftreten eines bestimmten Problems. Häufig wird diese Art im Bereich der Finanzen eingesetzt, sei es das Bewerten von Autos oder Häusern, die Preisvorhersage des Grundstücks in ein paar Jahren oder auch das Bestimmen des Aktienwertes. Es sollte noch erwähnt werden, dass die Regression mit der Klassifikation kombiniert werden kann, zum Beispiel um die Spam-Wahrscheinlichkeit einer Mail zu beziffern.

Trainingsdaten

Für einen ML-Algorithmus sind gute Daten das A und O! Ohne genug Daten, die auch qualitativ hoch sind, kann das System nicht gut genug und nicht genau genug (wenn überhaupt) trainiert werden. Es müssen ausreichend Daten zur Verfügung gestellt werden, damit auch viele verschiedene Variationen gelernt werden können. Je weniger Daten man hat, desto weniger aussagekräftig wird das Ergebnis. Als Vergleich: würde man ein Kübel Meerwasser aus einem Ozean ziehen, könnte man nicht auf die tausenden Arten der Meeresbewohner schließen. Dafür wären viel mehr Kübel notwendig, an verschiedenen Stellen und Tiefen des Meeres, um einen bestimmten Durchschnitt errechnen zu können. Ebenfalls muss darauf Acht gegeben werden, ob die Daten auch repräsentativ und relevant sind für das spätere, fertig trainierte Modell. Ein Modell, welches Preise für Grundstücke voraussagen soll, kann nicht mit Bildern trainiert werden. Dieses Beispiel ist womöglich weit hergeholt, dennoch ist hier klar, was gemeint ist - Daten müssen repräsentativ und relevant sein. Wenn ein Regressionsmodell erstellt werden soll, muss es auch mit Daten gefüttert werden, die dieses Modell trainiert und am Ende entstehen lassen können. Auch minderwertige Daten müssen vermieden werden, da es sonst zu Unschärfen kommen kann. Die Testdaten müssen auf Hochwertigkeit geprüft werden und gegebenenfalls manuell aufgewertet werden, wie zum Beispiel Ausreißer in den Daten zu entfernen oder fehlende Daten in einer Tabelle zu ergänzen [Ger17].

2.2.2 Unsupervised learning

Der große Unterschied zum supervised learning, ist es, dass es beim unsupervised learning keine Zielvariable gibt. Es werden Muster erkannt, jedoch ohne dass Muster durch einen Input und dessen Output gelernt wurden. Dem ML-Modell werden Daten eingespielt, ohne zu wissen, was diese Daten repräsentieren. Die Aufgabe ist nun, die Daten zu klassifizieren, in Clustern zusammenzufassen. Unsupervised-Modelle werden oft für visuelle Ergebnisse genutzt, da diese in einer Grafik einfach in Gruppen zusammengefasst werden können und so

einen schnellen Überblick gewähren lässt. Für Prognosen lässt sich dieses System nicht einsetzen, da es nur bestehende Daten klassifizieren kann. Dafür funktioniert es umso besser mit Echtzeitdaten, zum Beispiel mit den Echtzeitdaten der Besucher einer Webseite. Hier kann ausgewertet werden, wie viele Männer zu welcher Uhrzeit, wie viele Frauen wie lange auf der Webseite waren und so diese umgestalten um die Zielgruppe zu erreichen, die man erreichen will. Auch in der IT-Security werden unsupervised ML-Modelle eingesetzt um das Verhalten des Datenstroms zu analysieren. Sobald etwas nicht der Norm des Clusters entspricht, wird Alarm geschlagen, um einen möglichen Angreife zu entlarven. Das Clustering ist eine Methode des unsupervised learnings und wird auch in der Medizin eingesetzt um Krankheitsbilder zusammenzufassen. Zur Bildererkennung in Online-Fotoalben wird ebenfalls das Clustering eingesetzt, indem das Modell die Gesichter der Personen auf den Fotos erkennt und auseinanderhalten lernt. Zusätzlich können noch den 'gefundenen' Gesichter Namen-Labels hinzugefügt werden. Mit der Suchfunktion besteht nach dem Labelling die Möglichkeit, nach dem Namen 'Markus' zu suchen und so alle Bilder als Ergebnisse zu erhalten, auf denen die Person mit dem Label 'Markus' erscheint [RN12].

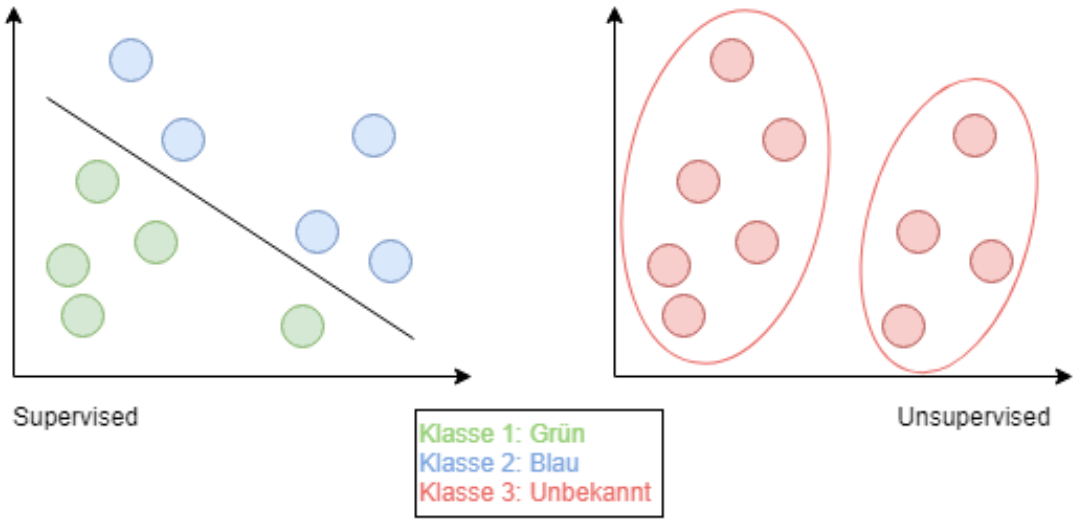

Abbildung 2.2: Unterschied Supervised und Unsupervised Learning [Ger17]

2.2.3 Reinforcement Learning

Reinforcement Learning, zu deutsch verstärkendes Lernen, wird eingesetzt, wenn das Training eines Modells zu komplex und zu zeitaufwändig ist. Um Schach zu erlernen, bedarf es natürlich das Wissen der Regeln angeeignet zu haben, jedoch muss auch jeder mögliche Zug in Betracht gezogen werden und für alle erdenklichen Fälle muss ein Entwickler den Code dafür schreiben - beim ersten Zug sind es bereits 400 mögliche Spielfigurenstellungen, da jeweils 20 Figuren für jede Farbe zur Verfügung stehen. Mit jedem neuen Zug entstehen neue Möglichkeiten des Spielausgangs - eine Programmierung der KI ist schlicht zu aufwändig. Um diese Problem lösen zu können, wird das Prinzip der Belohnung und der Bestrafung eingesetzt. So wie man einen Hund mit Leckerlis als Belohnung dressiert, so wird auch das ML-Modell mit Pluspunkten belohnt. Das Modell selbst zielt darauf ab, so viele Punkte wie möglich zu erhalten. Im Beispiel des Schachs wäre eine Belohnung oder Bestrafung nach jedem Zug nicht wünschenswert, da im Vorhinein nur schwer prognostiziert werden kann, ob dieser Zug

zum Ziel führt und den Gegner am Ende schlägt, oder ob dieser Zug der erste Zug ist, der zur Niederlage führt. Daher liegt es nahe, diesem Modell am Ende des Spiels, bei Niederlage oder Sieg, Punkte abzuziehen oder mit Punkten zu belohnen. Die Punktevergabe kann wie im Schach nach Absolvierung des Durchgangs verteilt werden oder aber nach jedem Schritt. Ein anderes Beispiel wäre ein menschenähnliches, zweibeiniges Modell. Dieses sollte nach jedem erfolgreichem Schritt belohnt und nach jeder Berührung des Oberkörpers mit dem Boden, also nach einem Fall, bestraft werden. Zwischenbelohnungen können auch zielführend sein, wenn das Modell so zu einem besseren und schnellerem Ergebnis kommt. Die Anzahl der möglichen Belohnungsvariationen ist hoch, da jede erdenkliche Möglichkeit ausprobiert werden kann für das Training des Modells. Auch hier gilt, wie bei anderen Modellen, dass Zeit und Ressourcen Hand in Hand gehen mit dem Training. Es ist möglich, dass ein Modell stunden- auch tagelang trainiert wird und kein befriedigendes Ergebnis liefert - einen Tag später jedoch fängt das Modell an, Resultate zu liefern und die Investition hat sich doch noch ausgezahlt. Näheres zum Belohnungssystem im Unterkapitel 'Belohnungssystem'.

In der Robotersteuerung ist dieses System daher wichtig, da es im Normalfall immer wieder neue Situationen gibt, die im Voraus nicht trainiert werden konnten. Das Modell selbst hat aber die Fähigkeit zu reagieren. Nehmen wir als Beispiel ein autonomes Fahrzeug. So einem Fahrzeug kann die gesamte Straßenroute Europas integriert werden, jedoch nicht alle Möglichkeiten der Verkehrssituationen. Somit muss es eigenständig auf verschiedene Wetterlagen, veränderte Kreuzungen mit Umleitungen, verschiedene Verkehrs- und Gefahrensituationen reagieren können. Die hier verwendete Simulation eines Roboters, hat ein Reinforcement Learning ML-Modell implementiert. Diese Simulation soll zeigen, wie ein Industrieroboter darauf trainiert werden kann, verschiedene Szenarien zu beherrschen und dies selbständig, mit bereits trainiertem Modell, durchzuführen.

2.3 Deep Learning

Deep Learning ist ein Bereich des maschinellen Lernens und somit eine Teilmenge dessen. Das neuronale Netz welches notwendig ist, ist hierarchisch aufgebaut. Das Netz selbst ist in Schichten aufgeteilt. Aufzählend wären das die Eingabeschicht, die verborgene(n) Schicht(en) und die Ausgabeschicht. Jedes neuronale Netz hat eine Eingabeschicht, in der die Rohdaten eingespielt, verarbeitet und an die (erste) verborgene Schicht weitergegeben werden. Es gibt mindestens eine verborgene (hidden) Schicht, diese verarbeitet die Daten aus der Eingabeschicht. Laut Jeff Heaton sind zwei verborgene Schichten ausreichend [Hea08], um gute Ergebnisse zu erzielen, jedoch reicht im Grunde bereits eine Schicht für die meisten Probleme - hier sind die Anzahl der Neuronen in dieser Schicht aussagekräftiger, als die Schicht selbst. Die Daten aus den verborgenen Schichten werden nach Bearbeitung an die Ausgabeschicht als Ergebnis geleitet, jedes neuronale Netz hat genau eine Ausgabeschicht.

Der Begriff des Deep Learning bezeichnet den Aufbau dieser Reihenfolge, welche essentiell ist für den Lerneffekt des neuronalen Netzes. Die Reihenfolge oder horizontale Hierarchie selbst würde bildlich dargestellt eine Reihe von Knoten zeigen, die verzweigt das neuronale Netz bilden. Durch die horizontale Hierarchie entsteht eine Tiefe (deep), die das Netz braucht um zu lernen und Ergebnisse liefern zu können [GBC16].

Die Vorteile, die sich durch diese Technologien ergeben sind bekannt, jedoch sind sie bei weitem nicht voll ausgeschöpft. Viele Bereiche der Wissenschaft können von dieser Technologie profitieren, manche in relativ kurzer Zeit und manche müssten noch auf einen Durchbruch warten. Zwei Beispiele für den Erfolg dieser Technologie wäre der Erfolg von AlphaGO, ein Programm welches angefangen im Jahr 2015 mehrere GO-Champions geschlagen hat, 2015

Fan Hui, 2016 Lee Sedol, 2017 Ke Jie [SHM+16]. Go wird als eines der kompliziertesten Spiele bezeichnet. Durch die vielen Möglichkeiten, wie das Spiel sich entwickeln könnte, ist es weit komplexer als Schach. Ein weiteres Beispiel ist der Durchbruch in der Forschung der Proteinfaltung. Hier konnte Google´s DeepMind mit dem System AlphaFold [SEJ+20] ein jahrelanges Problem mittels Deep Learning lösen: das Problem der Proteinfaltung. Das Problem bestand darin, den Prozess zu verstehen und zu entschlüsseln - die Wirkung der Proteine im Körper verstand man, doch der Prozess der Proteinfaltung, also die Vorhersage der Struktur aus der Aminosäuresequenz konnte bis AlphaFold nicht gelöst werden. Dank Deep Learning gibt es nun zumindest ein Werkzeug für die Entschlüsselung.

2.4 Neuronale Netze

Neuronale Netze sind der Grundstein für das biologische Nerven- und Kommunikationssystem. Der Informationsaustausch erfolgt über elektrochemische Vorgänge, die sich in unserem Hirn abspielen. Die Funktion aller Hirnareale ist in der Forschung noch nicht zur Gänze geklärt, es sind noch etliche Fragen offen, die beantwortet werden müssen. Vom Bewusstsein selbst, über dem erfindungsreichem Lügen bis zur Faszination von Nahtoderfahrungen. Dem menschlichen Gehirn steht eine Kapazität von 10^{11} Neuronen als Recheneinheit zur Verfügung [RN12]. Durch diese Anzahl an verfügbaren Neuronen können wir Berechnungen durchführen, Voraussagen treffen, philosophische Gedankengänge stricken, mathematische Formeln aufstellen und verstehen, Musik komponieren, Bilder malen, Menschen führen, Neues lernen und andere lehren. Warum sollte diese Fähigkeit des Menschen nicht auf eine Maschine übertragbar sein? Die ersten die ein mathematisches Modell eines künstlichen Neurons aufgestellt haben, waren die zwei Neuropsychologen Warren McCulloch und Walter Pitts [Fit44], die in ihrer Arbeit 'A logical calculus of the ideas immanent in nervous activity' eine vereinfachte neuronale Struktur aufzeigen, die die Grundlage für ein künstliches neuronales Netz bieten soll.

2.4.1 Biologische neuronale Netze

Die menschlichen Neuronen in einem neuronalem Netz sind spezielle Zellen, die sich im zerebralen Kortex befinden. Wie jede andere Zelle, hat es einen Zellkern, Mitochondrien, Ribosomen und alle weiteren wichtigen Bestandteile einer Zelle. Zusätzlich hat es verschieden lange 'Arme', die wie Tentakel aussehen. Zwei spezielle Arten sind die Dendriten und die Axone. Ein Neuron hat mehrere Dendrite, aber nur ein Axon. Über die Dendrite werden Informationen empfangen, über das Axon werden Informationen an andere Neuronen gesendet. Die Verbindung wird über die Axonverzweigungen hergestellt, Telodendria genannt. An den Enden der Telodendria befinden sich die Synapsen, über die dann die Kommunikation mit anderen Neuronen über deren Dendriten erfolgt. Die Synapsen sind nicht direkt mit den Dendriten einer anderen Zelle verknüpft, sondern haben einen gewissen Abstand dazu und senden nur chemische Signale. Diese Signale werden nur dann weitergegeben, wenn der elektrische Impuls stark genug ist, die Barriere zu überwinden - ansonsten gibt es keine Kommunikation. Im menschlichen Körper gibt es mehr als 86 Milliarden Neuronen, die miteinander kommunizieren - ein gigantisches biologisches Netzwerk! Auf Basis dieses Aufbaus wurde das künstliche neuronale Netz erstellt, besser gesagt war es eher eine Inspiration, als eine Vorlage. Die Technologie hat eigene Wege eingeschlagen, zugrunde liegt aber immer noch die Idee ein künstliches Gehirn zu erschaffen, welches verschiedene Fähigkeiten besitzt und verschiedene Probleme lösen kann [RN12], [Ger17], [CP19].

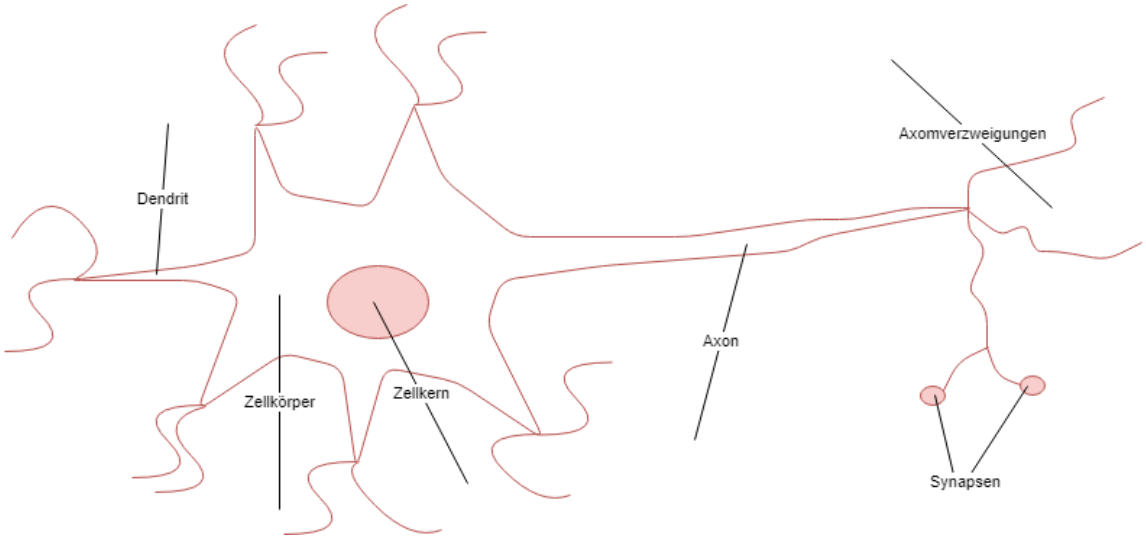

Abbildung 2.3: Vereinfachte Darstellung des Aufbaus einer Nervenzelle - ein Neuron [RN12][Ger17]

2.4.2 Künstliche neuronale Netze

Das Modell von McCulloch und Pitts basiert auf das Prinzip der Barriereüberwindung. Ein biologisches Neuron benötigt einen gewissen starken elektrischen Impuls, um die Information weiterzuleiten. Für das Modell des künstlichen Neurons, haben die beiden das binäre System verwendet, mehrere Verbindungen sind möglich und werden über die logischen Ausdrücke AND, OR oder NOT gesteuert. Neuron A aktiviert oder sendet eine Information an Neuron B, wenn A selbst aktiviert wird, also 1. Neuron C wird nur dann aktiviert, wenn Neuron A UND Neuron B aktiviert sind, also AND. Neuron C wird aktiviert, wenn Neuron A ODER Neuron B aktiviert werden, also OR. Neuron C wird nur dann aktiviert, wenn Neuron B NICHT und Neuron A aktiviert sind, also NOT. Somit wird die Barriere, die im biologischen Neuron mittels dem elektrischen Impuls überwunden werden muss, über logische Ausdrücke simuliert. Es müssen aber nicht logische Ausdrücke sein, die ein Neuron aktivieren, es können auch verschiedene Werte sein, was wiederum näher am biologischen Vorbild liegt. Zum Beispiel das Perzeptron von Frank Rosenblatt. Das Perzeptron ist eine Architektur eines neuronalen Netzes, daher sind hier bereits mehrere Neuronen miteinander verbunden. Die Eingangssignale werden gewichtet. Durch diese Gewichtung ergeben sich etliche Einstellungsmöglichkeiten [Ros58]. Diese meisten neuronalen Netze, die heutzutage verwendet werden, basieren auf den Perzeptron von Rosenblatt.

2.4.3 Convolutional Neuronal Network

Die Idee des Convolutional Neuronal Networks (CNN), wurde durch die Untersuchungen des menschlichen Hirns inspiriert, genauer gesagt durch Untersuchungen des visuellen Kortex. Bei der visuellen Wahrnehmung wird der Fokus bei Tieren und beim Menschen auf einen mehr oder weniger fixen Punkt gelegt, der ungefähr 3 cm breit ist. Dafür werden mehrere Neuronen gebündelt verwendet um das Gesamtbild zu erkennen. CNNs wurden zuallererst für die Bilderkennung verwendet, was naheliegend ist durch die Forschung des visuellen Aspekts. Der erste Einsatz war in Banken für die automatische Ziffernerkennung bei Schecks [LBBH98]. Heutzutage werden CNNs für viele visuelle Einsatzgebiete verwendet, wie zum Beispiel bei selbstfahrenden Autos, die Kameras verwenden, bei der Bildersuche oder auch in der Ge-

sichtserkennung. Zudem wurde das Gebiet erfolgreich auf die Bereiche der Stimmerkennung und der NLP (natural language processing) erweitert. CNNs werden ebenso im Bereich des supervised learnings eingesetzt.

In CNNs werden mehrere Filter angewandt, sogenannte Feature Maps. Es werden Filter für horizontale oder vertikale Linien, Filter für die Farben Rot, Grün, Blau verwendet - auch für die Lichtfrequenz gibt es einen eigenen Filter. Diese Filter bilden im CNN 2D-Gitter, die später aufeinander gestapelt werden und so eine 3D-Form erhalten - für jeden Filter wird eine eigene Feature Map verwendet. Diese Feature Maps bilden den Convolution Layer mit anschließender Aktivierungsfunktion, zum Beispiel einer ReLU, welche den Input der Neuronen in den Output verwandelt und negative Werte auf 0 setzt um überflüssige Daten zu entfernen.

Der Aufbau enthält mehrere Convolution Layer und darauffolgende Pooling Layer. Bei jedem Schritt durch diese Layer wird die Faltungsmatrix angewandt, die die Eingabe der Neuronen als inneres Produkts des Filterkernels (Feature Maps) berechnet und die nun kompaktere Information weiter an die nächste Schicht übermittelt. Die Nachbarn der jeweiligen Neuronen werden durch die oben genannten 2D-Form miteinbezogen. Die Gewichtung des Inputs ist bei jedem Neuron gleich und die Auswirkung auf die Neuronen erfolgt durch das 2D-Gitter nur lokal. Der Fokus liegt daher wie bei der Wahrnehmung des Auges auf das punktuelle Warhnehmungsfeld. Durch die verschiedenen Layer (Convolutional, Pooling) und den veränderten Wahrnehmungsbereich, werden immer feinere Pixel erkannt. Die Funktion kann man sich vorstellen wie einen Zoom auf einen gewissen Punkt in der Landschaft. Mit jedem Zoomfaktor wird die Umgebung größer und es werden mehr Details enthüllt.

Nach dem Durchlaufen des Convolutional Layers, werden die Informationen an den Pooling Layer geleitet. Dieser entfernt überflüssige Informationen durch die Aggregatfunktion. Ein Max-Pooling-Layer behält immer nur den 'Wert' des stärksten bzw. des größten Neurons. Ein Neuron aus der zweiten Schicht ist mit wenigen aus der ersten Schicht verbunden. Der Vorteil des Pooling Layers ist, dass durch das Durchlassen nur des größten Wertes, die weitergeleitete Information kompakter wird. Somit entsteht mehr Platz und es werden weniger Ressourcen für die Berechnung benötigt, es erhöht sich auch die Geschwindigkeit der Berechnung. Overfitting kann so vermieden werden.

Der Output des Pooling Layers landet als Input im Fully-Connected-Layer. Der Aufbau dieses Layers ist wie die des Perzeptrons. Nach diesem Layer erfolgt der tatsächliche Output. In der Anwendung der Ziffern im Scheckbeispiel von oben wäre das Ergebnis die Ziffer die erkannt wurde, die auf dem Scheck geschrieben steht.

2.4.4 Recurrent Neuronal Network

Im Gegensatz zu dem Feed-Forward-Prinzip eines CNNs, funktioniert ein rekurrentes neuronales Netzwerk (RNN) nach dem Prinzip der Rückkopplung, wenn es auch keinen bestimmten Output erreichen muss, wie in der Regelungstechnik. Die Rückkopplung erfolgt, um die Verarbeitung von Informationen zu ermöglichen, die zeitlich abhängig sind. Das bedeutet, dass die entscheidende Information X auf den Knoten A über eine Rückkopplung zum Knoten B weitergegeben wird, welcher sonst nur die Information Y enthalten würde. Durch diese Methode werden Informationen vorwärts und rückwärts im System geleitet und bilden so einen gewissen Speicher, eine Art Kurzzeitgedächtnis [RN12]. Die Rückkopplung selbst erfolgt entweder direkt nach vorne, indirekt zurück zum vorigen Neuron oder seitlich zu einem benachbarten Neuron.

Eingesetzt werden diese Netze im Bereich NLP-Bereich (natural langauge processing), in

der Analyse und Voraussage der Aktienkurse und auch in autonomen Fahrzeugen, die durch den Input der Sensoren Staus, Unfälle oder sonstige Verkehrssituationen vorausberechnen. Durch das Kurzzeitgedächtnis ermöglicht es eine gewisse Vorhersage zu bestimmen, welche im Magenta-Projekt für die Vorhersage einer Melodie eingesetzt und von Google 2016 veröffentlicht wurde (`https://www.youtube.com/watch?v=5rzbej320zs&t=68s`). Auch Deep-speare [LCB+18] (von 'Shakespeare' abgeleitet) verwendet das Prinzip der Vorhersage und wurde entwickelt um zu zeigen, dass auch eine KI Gedichte verfassen kann - ob diese auch Sinn ergeben, ist eine Frage der vorhandenen Gabe der Kunstinterpretation.

Ein einfaches RNN-Netz ist das Hopfield-Netz. Hier erhalten die einzelnen Neuronen als Input die Summe des Outputs aller anderen Neuronen, jedoch nicht den eigenen Output. Jedes Neuron ist bidirektional mit allen anderen Neuronen im Netz verbunden, daher gibt es auch nicht mehrere Schichten, sondern im Grunde nur eine einzelne. Der Wert der einzelnen Neuronen wird durch den Wert der anderen Neuronen multipliziert mit der Gewichtung der Graphen ersetzt, bis alle Neuronen einen stabilen Zustand erhalten. Das Training der Graphen erfolgt nach dem Erreichen des stabilen Zustands, indem der Wert der beiden Neuronen, an denen der Graphen angehängt ist, als Ausgangsbasis angenommen wird. Die Wertespanne reicht von 1 bis -1. Wenn der Wert der beiden Neuronen gleich ist, wird der Wert des Graphen um 1 erhöht. Wenn der Wert ungleich ist, wird der Wert des Graphen vermindert um 1. Die Graphen selbst behalten die Werte nach dem Training. Beim Testen erreichen die Neuronen irgendwann einen stabilen Zustand und durch die Berechnung mittels Graphen, werden die trainierten Werte wiedergegeben, insofern kann hier von einer Speicherung gesprochen werden. Hopfield-Netze sind jedoch fehleranfällig, da je nachdem an welchem Neuron die Berechnung startet, die Neuronen andere Werte annehmen können, als die trainierten/gespeicherten. Daher werden diese Netze nur bedingt verwendet, zum Beispiel als sogenannten Assoziativspeicher, welcher ein gewisses Rauschen in Kauf nimmt. So können Bilder einfach wiedergegeben werden, wenn auch mit einigen Pixelfehlern. Hier wurde das Hopfield-Netz beschrieben, um die Idee der Rückkopplung einfach beschreiben zu können [KBB+15].

2.4.5 Backpropagation

Das Backpropagationverfahren wird in einem DRL-Netz als eine Art mathematischer Regelkreis verwendet, der sich selbstständig und somit das Netz und den Output anpasst. Durch 'Feedforward' werden die Werte ermittelt durch das Durchlaufen der verschiedenen Layer, vom Input Layer über den Hidden Layer in den Output Layer. Der Soll- und der Ist-Wert werden verglichen und so der Fehler über eine Fehlerfunktion bestimmt. Damit das Netz von diesem Fehlerwert lernen kann, muss nun der Fehler vom Output Layer an den Input Layer über den/die Hidden Layer zurückgeführt (backpropagate) werden. So kann der Fehler durch mehrere Durchgänge minimiert werden [Ras16][RN12][SB18].

3 Robotik

Stellen wir uns vor, wir fahren von unserem Zuhause aus ins Altersheim, in dem unser Vater untergebracht ist. Auf dem Weg dorthin, halten wir kurz bei einem Restaurant an, bei welchem die Servierung am Parkplatz erfolgt. Wir bestellen ein paar Cheeseburger, Pommes und eine Cola. Solange wir im Auto warten, lassen wir uns durch die eingebauten Massagesessel im Auto durchkneten. Die Massagesessel können per Sprachsteuerung angesprochen werden. Je nachdem wo massiert werden soll, kann der Massagesessel auf Befehl die Rollen ansetzen und anfangen zu massieren. Nach einer gewissen Wartezeit, kommt ein Roboter aus dem Restaurant und bringt uns unser bestelltes Essen, es fragt uns nach unserem Wohlbefinden und ob wir noch Wünsche offen hätten, da fällt uns ein, dass wir keine Sauce mitbestellt haben. Wir sagen das dem freundlichen Roboter und aus einem eingebauten Kühlfach im Rumpf des Roboters, holt er eine Käsesauce heraus. Wir danken recht herzlich und der Roboter wünscht uns einen angenehmen Tag. Wir schalten das Navigationssystem unseres Autos ein, geben die Adresse des Altersheims ein und lehnen uns zurück. Aus den fahrzeuginternen Lautsprechern ertönt eine angenehme Stimme, die sagt 'Wir erreichen unser Ziel in genau 20 Minuten und 34 Sekunden' und das Auto fährt von alleine los. Wir haben nun ungefähr 20 Minuten Zeit um uns eine Folge 'Brooklyn Nine-Nine' anzusehen. Nach genau 20 Minuten und 34 Sekunden erreichen wir das Altersheim - unser Auto parkt sich selbstständig. Nun können wir endlich unseren Vater besuchen. Am Eingang wartet bereits ein weiterer Roboter, der uns begrüßt und uns nach unserem Befinden fragt. Außerdem bietet er uns seine Hilfe an, falls wir diese benötigen. Wir sagen dankend nein und gehen weiter ins Zimmer unseres Vaters. Dort begrüßen wir ihn und den 'Andrea', den Assistenzroboter unseres Vaters, Andrea hat ihm gerade die Mahlzeit und seine drei Tabletten, die er täglich nimmt und von Andrea auch kontrolliert wird. Andrea geht aus dem Zimmer und wir können den Nachmittag mit unserem Vater verbringen. Uns fällt auf, dass wir von uns Zuhause keinen aktiv bewegenden Menschen gesehen haben von Zuhause bis zum Altersheim. Alle die wir gesehen haben, haben entweder im eigenem Auto auf der Autobahn Zeitung gelesen, haben mit uns auf das Essen gewartet oder sind selbst Patienten im Altersheim oder Besucher. Wir denken nicht weiter darüber nach, früher mussten noch viele Dinge vom Menschen geführt und gelöst werden, heute haben uns die freundlichen Roboter die Arbeit abgenommen und wir können uns auf unsere Hobbies konzentrieren, mit unseren Familien Zeit verbringen, lesen und einfach das Leben genießen.

So oder so ähnlich könnte es in einigen Jahrzehnten auf unserem Planeten aussehen, zumindest in einigen wenigen und reicheren Teilen der Erde. Die Robotik hat die Revolution in der Industrie vorangetrieben, was die Leistung, die Qualität und die Schnelligkeit betrifft. Auch im Bereich der Sicherheit hat sich in der Industrie einiges getan, auch mithilfe der Sicherheitsbeauftragten, die Unfälle und Beinah-Unfälle dokumentiert und durch verschiedene Maßnahmen beseitigt haben. Die oben genannten Serviceroboter und autonomen Fahrzeuge sind bis zu einem gewissen Teil bereits Realität. Jedoch wird es noch einige Zeit dauern und einiges an finanziellen Mitteln für die Forschung, bis wird dieses Szenario als Realität haben werden.

Es werden verschiedene Arten von Robotern unterschieden: Industrieroboter oder Manipulatoren, mobile Roboter oder unbemannte Fahrzeuge und die mobilen Manipulatoren, wie

zum Beispiel der oben erwähnte Restaurantroboter, welcher selbstständig agieren kann und auch mobil ist.

Roboter nehmen ihre Umgebung durch verschiedene Sensoren wahr, seien es Kameras, Radar, Lidars oder Drucksensoren die ihnen sagen, wie weit oder sich ein Objekt befindet, wie fest der Roboter greifen darf oder mit welcher Geschwindigkeit sich Objekte von oder auf uns zu bewegen. Für die Fortbewegung und die Formung, also die eigentliche Bewegung der Gelenke sind Effektoren zuständig. Diese bestehen aus Motoren und Getriebe, wiederum Sensoren für die exakte Bewegung während einer Operation oder dem Klebeverfahren in der Herstellung von Smartphones. Hier können Roboter weit genauer und zielsicherer agieren als eine menschliche Hand. Für die Berechnung des eigenen Standorts verwenden mobile Roboter, die im Freien arbeiten GPS. Seien es autonome Fahrzeuge oder Drohnen des US-Militärs. Im Inneren kann das Drahtlossignal durch die Analyse mehrerer Signale der Basisstationen, für die genaue Position im Raum verwendet werden.

Die erlernte Bewegung der Roboter kann in drei Kategorien eingeteilt werden, mit denen wir uns in den nächsten drei Kapiteln genauer befassen.

- Der **klassische** Ansatz, der die Matrixmultiplikation verwendet
- Der Ansatz des **maschinellen Lernens**, genauer: mittels Deep Reinforcement Learning
- Der **regelbasierte** Ansatz, grob gesagt eine komplexe Art der If-Bedingung

3.1 Klassischer Ansatz

In der klassischen Robotik werden exakte mathematische Berechnungen durchgeführt beziehungsweise programmiert. Der klassische Ansatz findet sich häufig in der Automatisierungstechnik wieder, wo die Abläufe sich nur selten verändern. Durch den Vorteil der Geschwindigkeit und Genauigkeit der Roboter, können so identische Produkte entstehen, seien es Autos, Spielzeuge oder abgefüllte Milchpackungen. Die genauen Abläufe werden Schritt für Schritt analysiert und der Roboter darauf programmiert. Einfache Handgriffe, die vorher ein Mensch stundenlang monoton machen musste, können so ersetzt werden. Ein Roboter mit 6 Achsen kann somit viele verschiedene Arbeiten durchführen, die vorher programmiert wurden. Hier ist bereits der größte Nachteil. Jede Änderung in den Prozessschritten, bringt eine Änderung des Programms mit sich. Der Roboter ist darauf angewiesen, dass ein Programmierer ihn neu einstellt. Menschliches Versagen ist nicht auszuschließen, Fehler im Programm können zum Totalschaden führen. Außerdem verwendet fast jeder Hersteller eine eigene Programmiersprache, die die Programmierung und die Schulung der Programmierer umso schwerer und kostenintensiver gestaltet. Das Verfahren, in dem ein Programmierer ein Programm vorab ändern oder schreiben muss, sei es nun textuell oder CAD-unterstützt, nennt sich die indirekte Programmierung. Im Gegensatz dazu gibt es die direkte Programmierung, die wesentlich einfacher ist. Hier wird der Prozessschritt nicht klassisch programmiert, sondern die Schritte oder das gesamte Verfahren dem Roboter 'vorgespielt'. Der Techniker fährt den Prozess ab und speichert die Punkte im Raum und die Arbeitsschritte ab. So kann der Roboter ohne Programmierung und ohne Programmierkenntnisse des Technikers eingestellt werden. Zusätzlich gibt es noch das kombinierte Verfahren, in welchem das Programm offline geschrieben wird und durch die nötigen exakten Positionen online ergänzt wird [Cor17].

Für die indirekte Programmierung wird das Wissen der Koordinatentransformation benötigt. Hier unterscheidet man die direkte und die inverse Kinematik oder auch Rückwärtstransformation genannt, mit letzterem befassen wir uns nun. Ausgegangen wird vom Endeffektor,

zum Beispiel der Greifarm des Roboters. Um diesen zum Objekt A befördern zu können, damit er das Objekt greifen kann. Hierfür wird immer der Wert, in dem Fall der Winkelwert des Greifarms der Steuerung durchgegeben. Durch dieses Verfahren müssen sich die Roboterteile, die starren Körper und die Joints, an den Greifarm anpassen. Kurzerhand, der Greifarm bestimmt, wie sich die Joints und die Basis des Roboters rotieren und neigen müssen, damit das Objekt A gegriffen werden kann. Hierfür werden dem Bediener die Koordinatensystem die Raumpunkte angezeigt, wo sich der Greifarm befindet. Nun ergibt sich natürlich das Problem, dass es bei mehreren Joints es auch mehrere Degrees of Freedom gibt, die die Bedienung und die Programmierung des Roboters komplizierter macht. Hierfür ist es notwendig, die Beziehung aller Joints zu dem Endeffektor herzustellen, und diese Beziehung in ein Koordinatensystem zu vereinen. Im Verfahren der Denavit-Hartenberg-Transformation, werden die Werte in einer Matrix geschrieben um so die exakte Position des letzten Roboterstücks zu erhalten.

3.1.1 Koordinatensysteme

Der oben beschriebene Vorgang benötigt einen Tool-Center-Point für die Position im Raum, welcher im Mittelpunkt des Endeffektors liegt. Das Koordinatensystem wird für die Lage benötigt, die Rotationen der Achsen werden im Koordinatensystem erfasst. Für die Erfassung der Lage beziehungsweise Position der Körper im Raum, wird ein Bezugspunkt erfasst, den Nullpunkt, der die weiteren Werte bestimmt. Je nach Art des Koordinatensytems, können Winkelangaben, Längenangaben oder auch die Kombination dieser Angaben gemacht werden. Durch Transformationsgleichungen werden Parameter aus einem System in ein anderes transformiert. Es gibt drei verschiedene Koordinatensysteme, das kartesische Koordinatensystem, das Zylinderkoordinatensystem und das Kugelkoordinatensystem, wobei das kartesische das verbreitetste ist, vor allem bei CNC-Maschinen [BHS98].

3.2 Deep Reinforcement Learning Ansatz

Der Deep Reinforcement Learning Ansatz verfolgt das Ziel, eine veränderliche Umgebung nachvollziehen zu können und diese nicht als Hindernis zu betrachten. Roboter, die ein DRL implementiert haben, können so eine Reihe an Arbeiten durchführen, für die sie nicht programmiert wurden. Sie lernen selbstständig einen neuen Weg zu gehen und eine neue Strategie zu entwickeln, um das Produkt so effizient wie möglich zu produzieren oder transportieren. Die Funktion beruht auf das Trial-and-Error-Prinzip, bei dem der Roboter verschiedene Pfade ausprobiert, bis ein guter, im besten Fall optimaler, Weg gefunden wurde. Um diesen Pfad zu finden bedient sich der Roboter verschiedener Sensoren um die Umwelt zu observieren. Damit eine Strategie entwickeln werden kann, muss die Umwelt bekannt sein, die Rahmenbedingungen müssen gelernt werden, wie in etwa die Reichweite der Arme oder die maximale Rotationsgeschwindigkeit und die Bewegungsfreiheit. Je nach Anzahl der vorhandenen Bewegungsgrade (DoF - Degrees of Freedom), wird das System mit jeder weiteren Dimension komplexer. Dieses Problem wurden von Bellman als 'Fluch der Dimensionalität' bezeichnet [SB18] und sagt aus, dass mit jeder zusätzlichen Dimension, die Möglichkeiten zunehmen. Die Anzahl der Möglichkeiten steigt rapide mit jeder hinzugefügten Dimension, wie bei einem Industrieroboter, welcher 6 Joints besitzt und mit einem Roboter verglichen wird, der nur 2 Joints besitzt - die Dimensionalität ist bereits erheblich höher.

Der Roboter wird durch Belohnungen und Bestrafungen motiviert, um das Ziel zu erreichen, noch unabhängig der Dimensionen. Durch diese Belohnungen kann eine Strategie

entwickelt werden, die zum folgendem Ziel führt: je schneller das Ziel erreicht wird, desto besser. Das Ziel des Roboters ist es, eine Strategie zu entwickeln, die den optimalen Weg zum Ziel oder Zielobjekt findet und damit die Belohnung so hoch wie möglich ausfällt. Hier gibt es verschiedene Ansätze, wie und ob ein Roboter belohnt werden kann und soll.

Markov-Entscheidungsproblem

Das Markov'sche Entscheidungsproblem (englisch Markov decision process - MDP) ist ein wichtiger, wenn nicht sogar ein essentieller Punkt im Bereich des Deep Reinforcement Learnings. Dieser besagt, dass die Entscheidungen eines Agents abhängig sind vom derzeitigen Status s, welcher eine Abfolge von einer Menge von Entscheidungen ist, die in der Vergangenheit getroffen wurden. Die Wahrscheinlichkeit, dass der Agent einen zukünftigen Status s' erreicht, ist nur abhängig vom aktuellen Status s, nicht aber von vorherigen Status. Wenn nun T (transition probability) die Wahrscheinlichkeit des nächsten Zustands ist und a die Aktionen, die ausgeführt werden, kann folgendes aufgestellt werden: T(s', a, s) = π(s'|s, a), wobei π die Strategie (policy) ist. Die Strategie beruht nun darauf, dass die Menge der zukünftigen Status s' die Bedingungen s, a erfüllen [KBP13] [SB18]. Dieser Aspekt zeigt, dass ein Agent abhängig ist vom aktuellen Status und der ausgeführten Aktion.

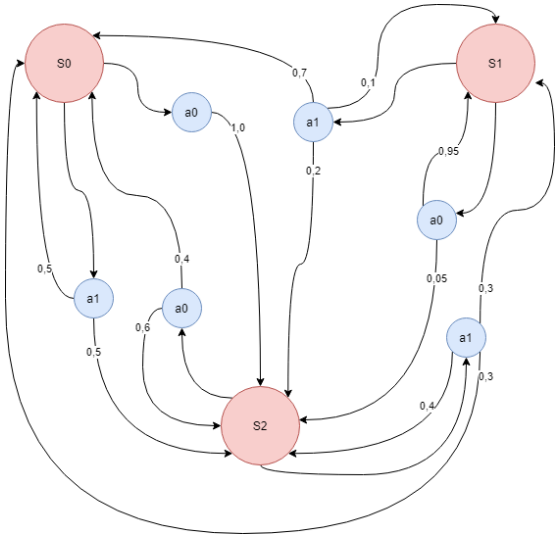

Abbildung 3.1: Darstellung Markov'sches Entscheidungsproblem [Ger17]

3.2.1 DRL-Elemente

Die Hauptelemente, die für einen DRL benötigt werden sind folgende [SB18]:

- Strategie
- Belohnungsfunktion
- Wertfunktion
- Umgebungsmodell

Strategie

Um die unzähligen Möglichkeiten die zum Ziel führen können systematisch durchgehen und bewerten zu können, muss eine Strategie entwickeln werden. Der Vorteil eine Strategie zu haben ist es, dass das Modell sich nach dieser richtet und diese verfolgt, anstatt die Wege die gewählt werden könnten errechnet werden, was uns zu einer erhöhten Ressourcennutzung führen würde. Es wird versucht, die Strategie so zu wählen, dass die Kombination des aktuelles Status s und der Aktion a zu einer möglichst hohen Belohnung führt, daher ist das Markov-Entscheidungsproblem beziehungsweise die Methodik der Herangehensweise die Voraussetzung für eine Strategie. Zur Verbesserung einer Strategie können verschiedene Vorgehensweisen gewählt werden, wie zum Beispiel die iterative Optimierung der Strategie, indem jede vorherige Strategie evaluiert wird. So kann eine fortlaufende Reihe mit dem Statuswert V^{π}, der Evaluierung E und der Optimierung O aufgestellt werden [SB18]:

$$\pi 0 \underset{\rightarrow}{E} V^{\pi 0} \underset{\rightarrow}{I} \pi 1 \underset{\rightarrow}{E} V^{\pi 1} \underset{\rightarrow}{I} \pi 2 \underset{\rightarrow}{E} V^{\pi 2} \; .... \underset{\rightarrow}{I} \pi * \underset{\rightarrow}{E} V^{\pi *}$$

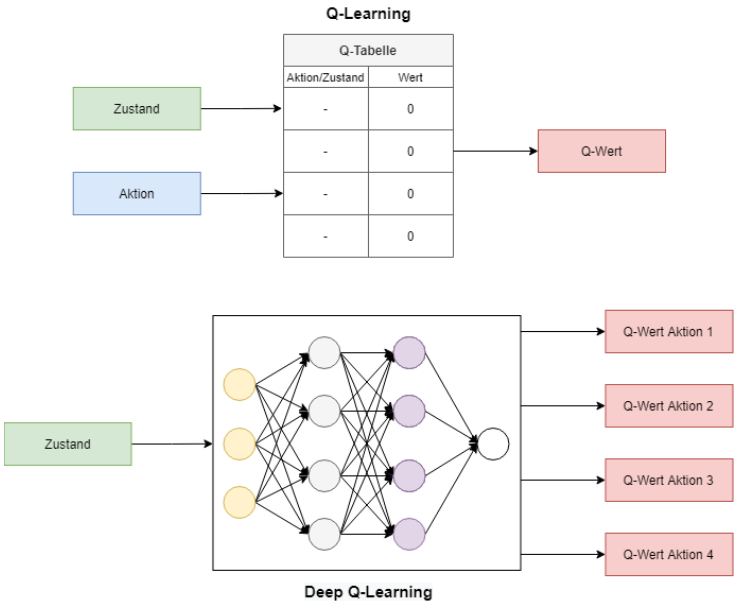

Abbildung 3.2: Q-Learning & Deep Q-Learning [Cho20]

Q-Learning: Die Nutzung einer Strategie in einem RL-Netz kann zum Beispiel über das bekannte Q-Learning erfolgen, das Q leitet sich von der Q-Funktion ab. Hierfür wird eine Aktion/Zustand-Tabelle erstellt, in dieser wird der Zustand s' geschrieben, welcher durch die Aktion a im Zustand s erzeugt wurde. Damit ein Netz Wissen aneignen kann und noch unbekannte Möglichkeiten und Zustände erreicht, werden mittels Exploration und Q-Learning die Werte berechnet und führen dazu zu noch unbekannten Werten. Das Netz weiß nun, welche Werte und damit welche Wege die meisten Punkte einbringen. Q-Learning selbst benötigt kein Modell in Form von T(s', a, s) = π(s'|s, a), daher wird es auch als modellfreie Methode bezeichnet. Eine Aktion muss nicht ausgeführt werden, denn die Werte werden im Vorhinein berechnet. Falls der berechnete Pfad x mehr Punkte als die Pfade y und z, wird in die Tabelle nur der Wert x eingetragen, der Startpunkt der Berechnung erfolgt per Zufall. Diese Tabelle kann jedoch sehr lang werden, die Suche darin kann schnell unübersichtlich für den Agent werden. Daher wird in einem DRL-Netz das Deep Q-Learning (DQL) verwendet. Es unterscheidet sich dadurch, dass das DQL sich der Q-Funktion nur annähert. Als Input dient nur der Zustand, die Q-Werte mit allen möglichen Aktionen entstehen als Output [RN12].

Belohnungsfunktion

Die Belohnungsfunktion oder das Belohnungssystem ist frei konfigurierbar, und kann verschiedene Belohnungsszenarien enthalten. Das Ziel für den Agent ist es, die Belohnung so hoch wie möglich ausfallen zu lassen, indem er das Ziel so schnell wie möglich erreicht. Die ist möglich, wenn die Strategie immer wieder angepasst und optimiert wird, ausgelöst durch die eingestellten Belohnungen und Bestrafungen, die von der Umgebung ausgegeben werden (siehe Abbildung 3.2). Ein Roboter, welcher ein Objekt greifen soll, kann eine Belohnung von +1 erhalten wenn das Objekt mit dem Greifer gefasst wurde und eine Bestrafung von -1 wenn der Greifer mit anderen Objekten als dem Ziel kollidiert. Einem Agent in einem Labyrinth Punkte zeitabhängig abgezogen werden, je länger er sich im Labyrinth befindet, desto mehr Punkte verliert er. Die Aktionen müssen daher so gewählt werden, dass der Zustand des Agents sich in die Richtung verändert, die am Ende die meisten Belohnungen einbringt, in diesem Beispiel würde sich der Agent so schnell wie möglich Richtung Ausgang begeben, damit die Belohnung hoch ausfällt. Die Belohnungsfunktion ist eine Summe von Belohnungen B die addiert werden. Je nach Summand, ob positiv oder negativ, wird die Belohnung höher oder niedriger ausfallen. Eine zeitabhängige Belohnungsfunktion würde den Faktor der Zeit miteinbeziehen, jeder Schritt t kann sich auf die Belohnung negativ oder positiv auswirken [SB18]:

$$R_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_{t+n}$$

In der Realität müssten natürlich auch die finanziellen Kosten miteinbezogen werden. Wenn die Schnelligkeit zum Ziel einen hohen Ressourcenverbrauch mit sich bringt, ein langsameres Antasten zum Ziel länger dauert, dafür aber weniger Kosten und Risiken verursacht, würde man eher die zweite Strategie verfolgen, wenn die Kosten ein Faktor sind. Daher ist es wichtig, das Belohnungssystem auf den jeweiligen Anwendungsfall anzupassen.

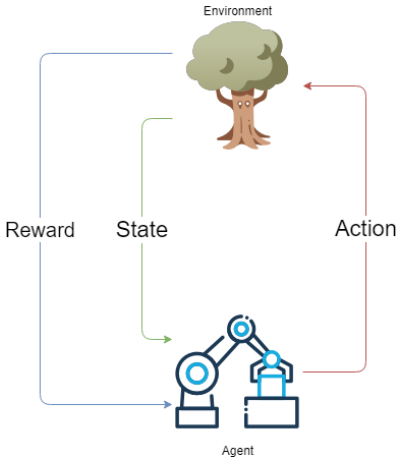

Abbildung 3.3: Darstellung Belohnungssystem [Ger17]

Wertfunktion

Die Wertfunktion ist für die Bewertung des aktuellen Zustandes zuständig. Mithilfe der Wertfunktion, wird der aktuelle Zustand evaluiert und die Vorhersage getroffen, welcher zukünftige

Zustand die höchste Belohnung mit sich bringt. Die Funktion führt die Berechnung vom jetzigen Zustand s und die Weiterführung der Abfolge von einer Aktion a durch die beeinflussende Strategie π [SB18]. Die Wertfunktion beim Training mit einem 6-DoF-Roboter würde in diesem Sinne um einiges komplexer ausfallen, als das Training mit einem Klotz, welches sich nur auf der X und Z-Achse bewegen kann und ein anderes Objekt erreichen muss. Die Vorhersage des Wertes im nächsten Zustand mit zwei Achsen benötigt weniger Ressourcen, als wenn 3 Achsen berechnet werden müssen.

Umgebungsmodell

Das Umgebungsmodell sind alle Faktoren die dem System helfen, eine Voraussage zu treffen. Alle Elemente, die einen zukünftigen Status berechnen können, werden in diesem Modell miteinbezogen. Die Umgebung, der aktuelle Status, das Belohnungssystem und der Punktestand werden herangezogen für das Modell, welches eine Prognose abgibt. Unterscheidet wird unter dem Beispielmodell (sample) und dem Distributionsmodell. Das Distributionsmodell errechnet alle Möglichkeiten und deren Wahrscheinlichkeit, dies bringt eine enorme Ressourcennutzung mit sich. Das Beispielmodell wiederum errechnet nur eine einzelne Möglichkeit, also ein Beispiel der vielen Möglichkeiten aus dem Distributionsmodell [SB18].

3.2.2 Herausforderungen

Die Herausforderungen, die bei der Lösung des Problems auftreten sind zahlreich. Die Schwierigkeit und Komplexität der Vorhersage, die ein zukünftiger Status mit sich bringt ist nicht zu unterschätzen. Es gibt unzählige Möglichkeiten, die ausprobiert werden können und verschiedene Belohnungssysteme führen zu verschiedenen Situationen, die eine Vorhersage komplexer machen. Dies wiederum führt zum Problem der Zeitfrage, denn das Training selbst kann mitunter Wochen dauern. Ein wochenlanger übender Roboter kann in der Produktion nicht eingesetzt werden. Es entstehen Kosten, die gedeckt werden müssen. Hier zeigt sich der Vorteil der Simulation in Editoren, wie zB Unity. Roboter sind komplexe Maschinen und durch ein DRL-Algorithmus wird eine nicht weniger komplexe Lösung implementiert. Durch die sich dadurch ergebene doppelte Komplexität, können sowohl Probleme als auch Möglichkeiten entstehen. Eine der Probleme, die diese Komplexität mit sich bringt sind die Informationen die entstehen und transportiert werden müssen. Die Informationen müssen während dem Training in Echtzeit zur Verfügung gestellt werden, damit der Roboter bzw. das DRL-Modell alle benötigten Informationen zur Verfügung gestellt bekommt um weiter vorgehen zu können. Somit ist auch das Problem der Echtzeitdaten zu berücksichtigen. Die Daten müssen zu gegebener Zeit zur Verfügung stehen, die Geschwindigkeit und etwaige Störung des Informationsflusses sind daher ebenfalls zu beachten. Die Kosten für einen Roboter sind erheblich, falls es bei einem Training zu einem Unfall kommen sollte, sind die Kosten der Ersatzteile miteinzubeziehen. Durch falsche Anweisungen oder Zielsetzung ist ein Totalschaden oder zumindest ein längerer Zeitraum in dem das Training pausiert wird nicht auszuschließen. Sensoren und Bestrafungen können einen Totalausfall abwenden, indem sie verhindern, dass gewisse Barrieren überschritten werden.

3.3 Regelbasierter Ansatz

Für den regelbasierten Ansatz werden für verschiedene Szenarien die eintreten können, Regeln aufgestellt. Vereinfacht könnte gesagt werden, dass hier mit Use Cases gut gearbeitet

werden kann, indem die genannten Szenarien aufgestellt werden und für jedes eine genaue Herangehensweise festgelegt wird. Nehmen wir als Beispiel einen Roboter, der vor einem Hindernis steht. Der Roboter detektiert das Hindernis und sieht in sein 'Regelbuch' nach. Wenn es ein Hindernis ist, welches nicht höher als einen halben Meter hoch ist und die Erhöhung eben weiter geht, dann soll der Roboter das erste Bein heben und auf die Erhöhung setzen, das Gewicht ausgleichen und das zweite Bein anheben und auf die Erhöhung setzen. Falls das Hindernis höher als einen halben Meter hoch ist, soll der Roboter sich nach rechts drehen und einen anderen Weg um das Hindernis suchen. Diese Art Roboter sind auf verschiedene Sensoren angewiesen, um die nähere Umgebung wahrnehmen zu können. Staubsaugroboter mit der Bounce Navigation sind ein einfaches Beispiel für diese Art der Wahrnehmung. Indem diese Staubsauger auf ein Hindernis stoßen, drehen sie sich in eine andere Richtung und führen weiter ihre Arbeit aus, nach den implementierten Regeln. Auch in der Prozessautomatisierung wird dieser Ansatz verfolgt, RPA (Robot Process Automation) genannt, indem Prozesse genau dokumentiert sind und von einem Software-Roboter nach gewissen Regeln abgearbeitet werden. Ein sehr einfaches Beispiel ist die regelbasierte 'WENN-DANN'-Funktion von Microsoft Excel und auch die IF-Bedingungen in den Programmiersprachen können einem Roboter implementiert werden, um sich in einem spezifischen Bereich zu bewegen oder zu arbeiten. Wie in der klassischen Robotik ist hier ein gut geschulter Programmierer notwendig, um die Implementierung und etwaige Änderungen durchzuführen. Als Nachteil kann sowohl dieser Aspekt gewertet werden als auch der Punkt, dass so eine Art Roboter sich nie in einer sich veränderbaren Welt bewegen werden kann, durch die schier unendliche Menge an Möglichkeiten die eine chaotische Welt mit sich bringt.

Funktionsweise

In der Robotik werden Regeln in Form von Kontrollmechanismen eingebaut. Sobald ein Sensor einen Input erhalten hat, wird dieser interpretiert und gewertet. Durch die eingebauten Regeln, wird ein Signal an den/die Effektor/en gesendet. Die Regeln sollen so einfach wie möglich aufgebaut sein, damit diese zu keiner Missinterpretation führt durch mehrere Inputs von verschiedenen Sensoren. Diese Regeln werden von unten nach oben aufgebaut (bottom-up), was bedeutet, dass Regeln aufeinander aufbauen und so komplexere Ausführungen durchgeführt werden können. Diese Art von System wurde für den Bereich der Portalrobotik entwickelt, welcher sich für aufbauende Regelmechanismen gut eignet [MM08].

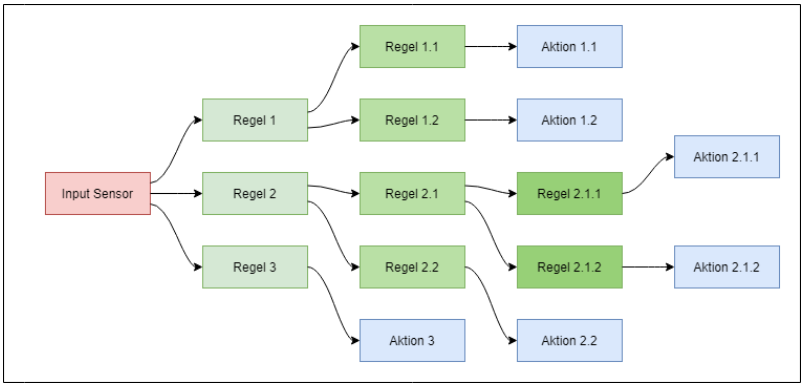

Abbildung 3.4: Darstellung einer regelbasierten Aktionsausführung [MM08]

4 Unityprojekt: Roboter mit Reinforcement Learning Modell

4.1 Ziel des Projekts

Ziel dieses Projekts ist es, für einen Roboter eine optimale Strategie zu finden, um ein Objekt zu erreichen. Verschiedene Belohnungssysteme werden verglichen und validiert. Durch abgeänderte Belohnungen und Bestrafungen soll gezeigt werden, ob ein simulierter Roboter schneller oder langsamer das Ziel erreichen kann. Durch das Zulassen mehrerer Trainingsschritte soll ebenso gezeigt werden, ob der Roboter mit einer längeren Trainingsphase erfolgreicher ist, oder eine kürzere Phase mit einem bestimmten Belohnungssystem besser ist. Mittels TensorBoard werden die Ergebnisse visuell dargestellt, um auch die Lernkurve des Agents betrachten zu können.

4.2 Tools

In diesem Kapitel befassen wir uns mit den Werkzeugen, die für das Projekt benötigt werden. Die hier genannten Tools sind wohlbekannt und können auf verschieden Weisen eingesetzt werden, nicht nur im Bereich des Machine Learnings. Unity wird auch in der Filmindustrie, in der Spielbranche und auch in Architekturbüros eingesetzt, wenn auch nicht als Hauptwerkzeug. Python hingegen ist eine höhere Programmiersprache und ist seit einigen Jahren die Basissprache, wenn es um Machine Learning geht, auch wegen den etlichen Bibliotheken die es speziell für diese Sprache gibt.

4.2.1 Unity

Uniy ist ein 2D- und 3D-Editor, der in der Spieleindustrie weit verbreitet ist. Unity ist eine Cross-Plattform und ist daher auf verschiedenen Plattformen verfügbar. Durch die integrierte Physik-Komponente, können so realitätsgetreue Abläufe dargestellt werden. Dieser Aspekt ist ein Vorteil, wenn wir eine uns bekannte Umgebung simulieren wollen, wie zum Beispiel einen simulierten Roboter, der im Schnellverfahren das Training für einen gewissen Einsatzzweck absolvieren soll. Durch die verschiedenen Pakete, die durch den Package Manager hinzugefügt werden können, kann der Editor beliebig erweitert werden. Jedem Objekt (in Unity ist alles ein GameObject) können Eigenschaften zugewiesen werden, die Konfiguration erfolgt entweder durch bereits vorhandene Eigenschaften durch hinzugefügte Komponenten oder über Skripte, die erstellt und wie bereits vorhandene Komponenten hinzugefügt werden. In der Szenenansicht (SceneView) kann ein Objekt manuell verändert werden, neue Objekte hinzugefügt oder entfernt werden. Die Spielansicht (GameView), wird durch den Playbutton aktiviert und die in der Szenenansicht bearbeitete Szene gestartet. In dieser Ansicht können die Eigenschaften im Inspektor Live geändert werden und haben so einen direkten Einfluss auf die Szene, vorteilhaft wenn verschiedene Abläufe ausprobiert werden sollen. Im Projektmanager werden alle Elemente wie zum Beispiel Modelle, Prefabs, Skripte, Materialien oder

Szenen wie in einem Dateimanager sortiert und aufgelistet. Unity bietet sehr viele kreative Möglichkeiten und ist zudem benutzerfreundlich zu bedienen. Daher wird dieser Editor auch in diesem Projekt verwendet, auch zumal die Community sehr groß ist und es viele verwendbare Assets gibt, die teilweise kostenlos, teilweise kostenpflichtig zur Verfügung stehen.

ML Agents

Das ML Agents Toolkit ist ein von Unity bereitgestelltes Open-Source-Projekt, welches verwendet werden kann mit verschiedenen Machine Learning-Varianten durch die vorhandene Python API. Durch dieses Paket können Modelle trainiert werden, die dann in Spielen oder in Simulationen eingesetzt werden können. NPCs (Agents) können trainiert werden indem sie durch Skripte Anweisungen erhalten, wie sie sich bewegen können, welches Ziel sie verfolgen und erreichen sollen und durch eingestellte Belohnungen beziehungsweise Bestrafungen kann so die Strategie beeinflusst werden. Für das Training werden drei Komponenten benötigt: Observations, um die Umgebung wahrzunehmen, Aktionen, die der Agent durchführen kann, je nach Aufbau des Agents und die oben erwähnten Belohnungen, welche als Skalar fungieren, je nachdem ob die durchgeführte Aktion 'Gut' oder 'Schlecht' im Skript bewertet wurde. Es gibt verschiedene Observations für verschiedene Szenarien, die in den Unterkapiteln weiter unten näher beleuchtet werden. Die Python API spielt hier eine große Rolle und führt die Manipulation für die Lernumgebung durch. Die Python API ist kein Teil von Unity und liegt außerhalb, verbunden durch den Kommunikator mit Unity, mehr dazu im Unterkapitel Python.

Agent: Ein Agent ist im Grunde nichts anderes als ein Charakter im Spiel. Der Agent muss Aktionen durchführen, basierend auf seine Wahrnehmung, seines bereits vorhandenem Wissen und auf das festgelegte Ziel. Auf das Erreichen des Ziels liegt der Fokus und dieses soll er autonom erreichen. Viele Wege führen nach Rom und auch hier führen viele Wege zum Ziel. Erwartet wird, dass das beste Ergebnis erzielt wird, das wiederum gesteuert werden kann durch Anweisungen und dem konfigurierbaren Belohnungssystem. Der Agent muss lernen ein besseres Ergebnis zu erzielen, indem er sich dem Gegebenheiten anpasst. Hierfür ist es notwendig, dass er die Umgebung wahrnehmen kann, durch verschiedene Observationen. Doch nicht nur das Ziel ist wichtig, sondern auch der Weg. Als Beispiel nehmen wir einen Staubsaugroboter, der einen Raum reinigen soll. Wenn nun das Ziel ist, so viel Schmutz wie möglich einzusammeln, zumal es mehr Punkte bringt, dann würde der Agent lernen, dass wenn er den Schmutz aufsaugt, diesen dann wieder auslässt und wieder aufsaugt, mehr Belohnungspunkte erhält. Um dies zu verhindern, können verschiedene Einstellungen vorgenommen werden, wie zum Beispiel Punkte zu vergeben, je schneller der Schmutz eingefangen wurde, anstatt Punkte zu vergeben für den aufgefangenen Schmutz. [RN12] In diesem Projekt, soll der Roboter ein Objekt erreichen, indem er alle seine Roboterteile in die Richtung des Objekts steuert, damit der Greifer es berühren kann. Dafür ist wie oben erwähnt notwendig, dass er seine Umgebung wahrnehmen kann. Nachfolgend sind drei Möglichkeiten beschrieben, wie unser Roboter dies erreichen könnte. Im Projekt wird die Vector Observation verwendet.

Vector Observations: ML Agents verwendet ein Fully-connected neuronales Netzwerkmodell. Das Modell kann so von Beobachtungen lernen und kann ebenfalls konfiguriert werden, indem man die Anzahl der verborgenen Schichten (hidden layer) hinzufügt oder minimiert. Der Agent, in unserem Fall der Roboter, versucht eine Strategie zu finden, um das Objekt zu erreichen, indem er das Ziel ausmacht. Dem Ziel selbst wird in Unity ein einfaches Skript hinzugefügt mit dem Label 'Ziel'. Dem Roboter fügen wir ein Skript hinzu,

in welchem wir 'CollectObservation' überschreiben, damit die Koordinaten des Zielobjekts gefunden werden können. Außerdem fügen wir dem Roboter Beobachtungen für die Koordinaten seiner eigenen Bestandteile hinzu. In der Simulation versucht der Roboter nun das Ziel zu orten. Die eigenen Koordinaten kann er nun auch 'beobachten' und bewegt seine Joints in die Richtung des Zielobjekts.

Visual Observation: Für diese Beobachtung sind Kameras notwendig, die in Unity über das Kameraobjekt hinzugefügt werden können. So können mehrere Winkel abgedeckt werden und der Agent erhält eine weite Perspektive um Beobachtungen durchzuführen. Für diese Beobachtungen wird ein CNN verwendet, welches durch den Input der Kamerabilder lernt. Diese Art der Beobachtung kann hinzugefügt werden, wenn Vektorbeobachtungen nicht ausreichen oder die gewünschten Ergebnisse nicht erreicht werden. So können mehr Information gesammelt werden, wodurch aber auch der Ressourcenverbrauch steigt.

Raycast Observation: Diese Beobachtungsart ähnelt der visuellen Beobachtung, mit dem Unterschied, dass einzelne Strahlen für die Beobachtung genutzt werden. Diese Strahlen können im 3D-Raum einzelne Bereiche abtasten, anstatt wie in der obigen Variante das gesamte Umfeld mit der Kamera zu detektieren. Die Anzahl der Strahlen, der Radius und die Länge können in Unity über den Inspektor eingestellt werden.

4.2.2 TensorFlow

TensorFlow ist eine Programmierbibliothek in der Programmiersprache Python und C++. Sie wurde vom Google-Brain-Team entwickelt und steht frei zur Verfügung, da es 2017 als Open-Source-Plattform freigegeben wurde. TensorFlow wird hauptsächlich im Bereich des Machine Learnings eingesetzt. TensorFlow stellt für das neuronale Netz des ML Agents Toolkit ein mathematisches Modell zur Verfügung, indem Knoten die mathematische Operationen darstellen, zu einem Graphen verbunden werden. Die Knoten selbst sind in einem Diagramm enthalten, die untereinander mit Kanten verbunden sind. Ein Tensor ist ein In diesem Projekt benötigen wir TensorFlow um das mathematische Modell zu erhalten und zusätzlich für TensorBoard.

TensorBoard

TensorBoard ist ein Werkzeug von TensorFlow für die Visualisierung des Trainingsfortschritts des Agenten. Durch verschiedene Modellgraphen und können wir beobachten wie sich unser Modell verhält und wie es auf unser Belohnungssystem anspricht. Durch verschiedene Modelle, die miteinander verglichen werden können, können wir sehen, ob die aktuelle Strategie es wert ist weiter zu verfolgen oder wir eine komplett andere Vorgehensweise ausprobieren müssen. Die Grafiken sind sehr hilfreich, auch zumal unser Gehirn auf visuelle Reize besser und schneller anspricht.

4.2.3 Python

Die höhere Programmiersprache Python ist im Bereich des Machine Learning weit verbreitet und wird sehr häufig für die Erstellung und das Training eines neuronalen Netzes verwendet. Ein Grund dafür ist sicherlich auch die strukturierte und lesbare Weise, die diese Sprache aufweisen kann. Die Verbindung zu Unity beziehungsweise zum ML-Agents Toolkit erfolgt durch den Kommunikator in der Lernumgebung, hier entsteht die Verbindung zur Python

API, die wiederum mit dem Python Trainer verbunden ist. Der Python Trainer enthält alle Algorithmen, die für das Training notwendig sind.

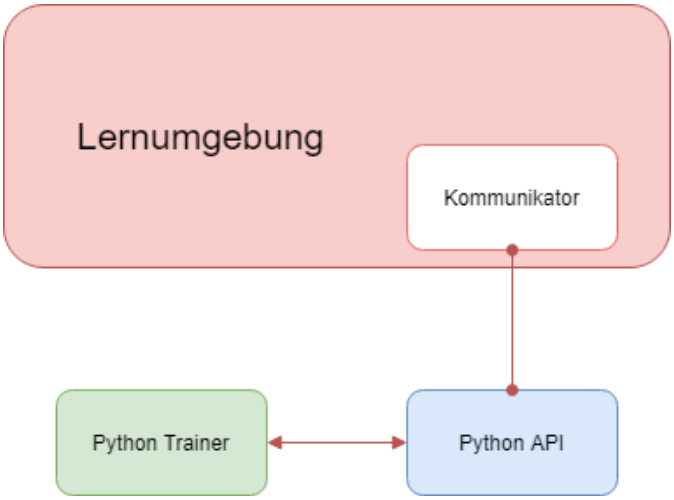

Abbildung 4.1: Python API [Tec]

PyTorch

PyTorch wird im Projekt benötigt und verwendet, um das Modell zu erhalten, welches in einer Datei gespeichert wird mit der Endung '.onnx'. Dieses Modell kann nach abgeschlossenem Training dem Roboter über den Inspektor in Unity hinzugefügt werden. PyTorch selbst ist eine Deep Learning Open-Source Bibliothek für Python, die mit dem ML-Agents Toolkit ausgeliefert wird. Die Bibliothek wurde vom Entwicklerteam für KI des Internetgiganten Facebook entwickelt.

4.3 Parameter, Belohnungssystem und Roboter

Das hier verwendete Roboter- und ML-Modell wurde vom GitHub-User 'rkandas' erstellt und auf der Plattform online und frei verfügbar gestellt. Das Projekt wurde übernommen und für das Training angepasst. Zu finden ist das GitHub-Projekt unter folgendem Link: `https://github.com/rkandas/RobotArmMLAgentUnity`

Um nun zu zeigen, welche Strategie die bessere ist um ein Ziel zu erreichen, wurden verschiedene Parameter eingestellt und verändert. In diesem Projekt wird der Trainertyp PPO (Proximal Policy Optimization) verwendet, ein Reinforcement Learning Algorithmus entwickelt vom OpenAI-Team, welcher die Vorteile der Dateneffizienz und der Leistung vom TRPO-Algorithmus (trust region policy optimization) übernimmt und zusätzlich die Trainingsdaten in mehrere, gleich große Pakete (Batches) teilt. Der Algorithmus wird so im Vergleich zum TRPO und anderen bekannten Algorithmen einfacher und performanter [SWD+17].

Die wichtigsten Parameter sind hier aufgelistet und beschrieben [Tec]:

- **batch size:** Die Batch-Größe ist die Anzahl an gesammelten Erfahrungen, die in einem Schritt gesammelt werden. Die Batch-Größe sollte immer um ein Vielfaches kleiner sein,

als die Buffer-Größe, die die Batches enthält. Mit den im Buffer enthaltenen Batches wird die Strategie aktualisiert, sobald die Zahl der Batches im Buffer erreicht ist.

- **hidden units:** Dieser Parameter stellt die verborgenen Einheiten in der verborgenen Schicht im neuronalen Netz ein. Je nachdem wie komplex das zu lösende Problem ist, sollen weniger (Problem nicht komplex) oder mehr (Problem sehr komplex) Einheiten hinzugefügt werden. Je mehr Einheiten benötigt werden, desto größer ist der Ressourcenverbrauch.
- **num layers:** die im oberen Punkt genannten verborgenen Einheiten sind in verborgenen Schichten enthalten, die Anzahl dieser Schichten wird hier gesetzt.
- **max steps:** hier wird die maximale Anzahl an Durchgängen gesetzt. Dieser Parameter darf nicht vergessen, ansonsten trainiert das Netz ewig oder solange bis der User es manuell abbricht. Wenn mehrere Agents parallel trainieren, werden die Schritte addiert. Somit kann die Dauer des Trainings erheblich reduziert werden, wenn die notwendigen Ressourcen vorhanden sind.

Von Unity vorgegebene Startwerte für ML-Agents:

```
behaviors:
    default:
        trainer_type: ppo
        hyperparameters:
            batch_size: 256
            buffer_size: 2560
            learning_rate_schedule: linear
            learning_rate: 3.0e-4
        network_settings:
            hidden_units: 256
            normalize: false
            num_layers: 3
            vis_encode_type: simple
            memory:
                memory_size: 256
                sequence_length: 256
        max_steps: 10.0e5
        time_horizon: 64
        summary_freq: 1000
        reward_signals:
            extrinsic:
                strength: 1.0
                gamma: 0.99
```

Parameterauswahl/Strategie

Für die Simulation wurden die Parameter wie erwähnt verschieden gewählt und die Ergebnisse sollen zeigen, wie der Roboter sich durch diese Parameter verhält und wie viel Zeit ein Training in Anspruch nimmt in Bezug auf gewählte Einstellungen. Hierzu wurden durch die eingegebenen Parametern 10er Batch-Pakete erstellt in einem Buffer. Beginnend mit einer Batch-Größe von 512, folgt daraus ein Buffer von 5.120 Paketen. Es erfolgen drei Trainingsabfolgen mit einer Buffergröße von 5.120 Paketen, bei jeder Abfolge verdoppeln sich jedoch die

Parameterübersicht			
	Parameter A	Parameter B	Parameter C
Training #1			
batch size	512	512	512
buffer size	5.120	5.120	5.120
hidden units	32	64	128
num layers	3	3	3
Training #2			
batch size	1.024	1.024	1.024
buffer size	10.240	10.240	10.240
hidden units	32	64	128
num layers	3	3	3
Training #3			
batch size	2.048	2.048	2.048
buffer size	20.480	20.480	20.480
hidden units	32	64	128
num layers	3	3	3

Tabelle 4.1: Parameterübersicht Robotersimulation

verborgenen Einheiten. Startend mit einer Anzahl von 32 verborgeneren Einheiten im ersten Training, verdoppelt es sich 64 Einheiten beim zweiten Training und im dritten Training werden 128 Einheiten verwendet. Für alle Trainingsdurchgänge wurden die maximalen Schritte mit 1.000.000 begrenzt Eine Auflistung der Trainingsrunden soll Übersicht verschaffen:

Belohnungssystem

Belohnungssystem		
Aktion	Belohnung	Bestrafung
Zielobjekt erreichen	+1	
Annäherung Zielobjekt	Startpunkt - Distanz zum Zielobjekt	
Boden berühren		-1
Entfernung Zielobjekt		letzter Standort - Distanz zum Zielobjekt

Tabelle 4.2: Belohnungssystem Robotersimulation

Das Belohnungssystem besteht aus Belohnungen und Bestrafungen und wurden dem Netz je nach Verhalten und Bewegung dem Punktekonto addiert oder subtrahiert. Der Roboter wurde mit einem Punkteabzug (Startpunkteabzug -1) bestraft, wenn der Endeffektor den Boden berührte. Wenn der Roboter das Zielobjekt erreicht hat, wurde es mit einem Pluspunkt belohnt (Startpluspunkteanzahl +0.5). Zusätzlich gab es Plus- und Minuspunkte für die Entfernung zum Zielobjekt, je nach Entfernung eine stärkere oder schwächere Auswirkung auf die Punktevergabe/-abgabe. Für den Trainingsanfang wurden die Plus-Werte +0.5 bzw. der Nähe-Bonus wenn der Greifer in der Nähe des Objekts ist und Minus-Werte -1 bzw. Ab-

zug abhängig von der Entfernung zum Objekt gewählt. Nach Auswahl der besten Strategien, wurden diese Werte verändert, um die besten ausgewählten Strategien weiter zu optimieren beziehungsweise die Auswirkungen zu erheben.

Ein Beispielcode für eine mögliche Belohnung: wenn der Greifer das Objekt erreicht, werden Punkte vergeben:

```
public void JackpotReward(Collider other)
{
   if (other.transform.CompareTag("Components"))
   {
      float reward = 0.5f;
      AddReward(reward);
      UpdateNearestComponent();
   }
}
```

Ein Beispielcode für eine mögliche Bestrafung: wenn der Greifer den Boden berührt, gibt es Minuspunkte und die Episode des Tests wird beendet:

```
public void GroundHitPenalty()
{
   AddReward(-1f);
   EndEpisode();
}
```

4.3.1 Roboter

Der Roboter besteht aus fünf Achsen, welche sich im Winkel von -90 bis 90 Grad und von -120 bis 120 Grad beugen können, eine rotierende Achse als Basis des Roboters, welche um 360 Grad rotieren kann und einen Greifer. Mit diesen Eigenschaften wird ein 6DoF erzeugt, also sechs Freiheitsgrade. Die verschiedenen Joints und starren Körper können durch folgenden Ausdruck auf die Position des Endeffektors E schließen, welcher zu einer Brücke zum Robotikkapitel wird. R steht für die Rotation, T für die Transformation der Verbindungen, V für die Verbindung, j für die Joints und a für die Verbindungen selbst [Cor17]:

$$E(j) = R(j1) \oplus T(a1) \ \oplus \ R(j2) \oplus T(a2) \ \oplus \ R(j3) \oplus T(a3) \ \oplus \ R(j4) \oplus T(a4) \ \oplus \ R(j5) \oplus T(a5) \tag{4.1}$$

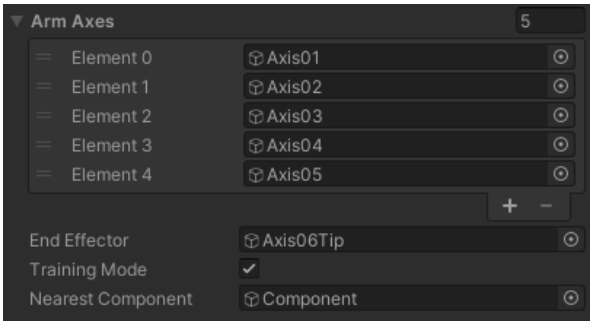

Abbildung 4.2: Unity-Inspektor: Joints als 'Axis' bezeichnet, Endeffektor als 'Axis06Tip' und Zielobjekt als 'Component'

In Unity ist der Roboter hierarchisch aufgebaut. Jede Verbindung und jeder Joint ist mit dem Parent verknüpft, diese werden mit der Basis über die Hierarchie 'physikalisch' verbunden und sensorisch über den Inspektor (Abbildung 4.2). Der Endeffektor wird im Inspektor eigens hinzugefügt. Das Zielobjekt, als 'Component' bezeichnet, wird ebenso im Inspektor eingestellt. Für das Training wurde der gesamte Roboter inklusive dem Zielobjekt und der Grundfläche kopiert und mehrmals eingefügt. In diesem Szenario waren es neun Roboter die parallel das Ziel verfolgten, das Zielobjekt zu erreichen. Das hat den Hintergrund, dass hier neun verschiedene Szenarien durchgespielt werden können in einer Episode. Ein einzelner Roboter würde demnach neun mal länger für die gleichen Ergebnisse benötigen. Es können grundsätzlich auch mehrere Kopien des Roboters das Training parallel durchführen, jedoch ist der Ressourcenverbrauch auch dementsprechend höher. Für dieses Training wurden daher nur neun verwendet, da die Ressourcenkapazität beschränkt war.

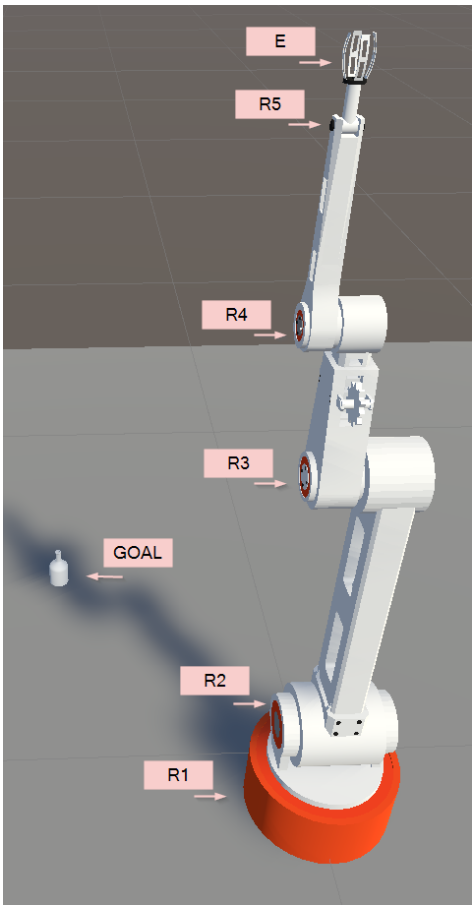

Abbildung 4.3: Roboter in Unity

4.4 Ergebnisse

In diesem Kapitel werden die Ergebnisse der durchgeführten Trainingsdurchläufen, neun an der Zahl. Die Parameter aus der Tabelle 4.1 wurden angewendet um neun verschiedene Szenarien vergleichen zu können. Die Ergebnisse wurden mittels TensorFlow in einem Ergebnisordner im Projekt abgespeichert, die Daten werden von dort in TensorBoard grafisch aufbereitet. Der Vergleich der verschiedenen Trainingsszenarien kann durch Auswahl der gewünschten Durchläufen dargestellt werden. TensorBoard fasst die Daten in folgende Kategorien zusammen: Environment (darunter Cumulative Reward und Episode Length), Losses (darunter Policy Loss und Value Loss) und Policy (darunter Beta, Entropy, Epsilon, Extrinsic Reward, Extrinsic Value Estimate und Learning Rate). Die Werte können in TensorBoard verschieden angezeigt werden, für diesen Vergleich wurden die Werte pro Schritt genommen. Alle 200.000 Schritte wurde der Wert verglichen. Pro Parametereinstellung aus der Tabelle 4.1 wurden insgesamt 1.000.000 Schritte von neun Kopien des Roboters, wie weiter oben erwähnt, durchgeführt.

Für die weiter unten vorgestellten Ergebnisse wurden folgende Kategorien und Unterkategorien verglichen:

1. Cumulative Reward (Environment): Die kumulierte Punkteanzahl für alle Agents die das Training absolviert haben. Dieser Wert sollte während einem erfolgreichen Training

zunehmen.

2. Policy Loss (Losses): Dieser Wert zeigt an, um welchen Faktor sich die Strategie im Laufe des Trainings ändert. Je höher der Wert, desto öfter wird die Strategie verändert. Das Ziel ist einen niedrigen Wert zu erreichen, welcher damit zeigt, dass eine optimale Strategie gefunden wurde oder auf dem besten Weg ist.
3. Value Loss (Losses): Dieser Wert zeigt an, wie gut das Modell den Wert des zukünftigen Zustand prognostizieren kann. Je weiter fortgeschritten der Trainingsprozess ist, desto niedriger sollte der Wert sein und sich auch stabil verhalten. Ein hoher Wert zeigt an, dass das Modell noch keine Erfahrung mit der Kombination aus Status und Aktion und der daraus resultierenden Belohnung hat.
4. Entropy (Policy): Die Entropie zeigt an, wie zufällig die Entscheidungen bzw. die Aktionen sind. Je höher der Wert, desto zufälliger werden die Entscheidungen getroffen, da die Strategie noch nicht bekannt ist und somit auch die Aktionen die zu einem Punktverlust oder zu einer Belohnung führen können.

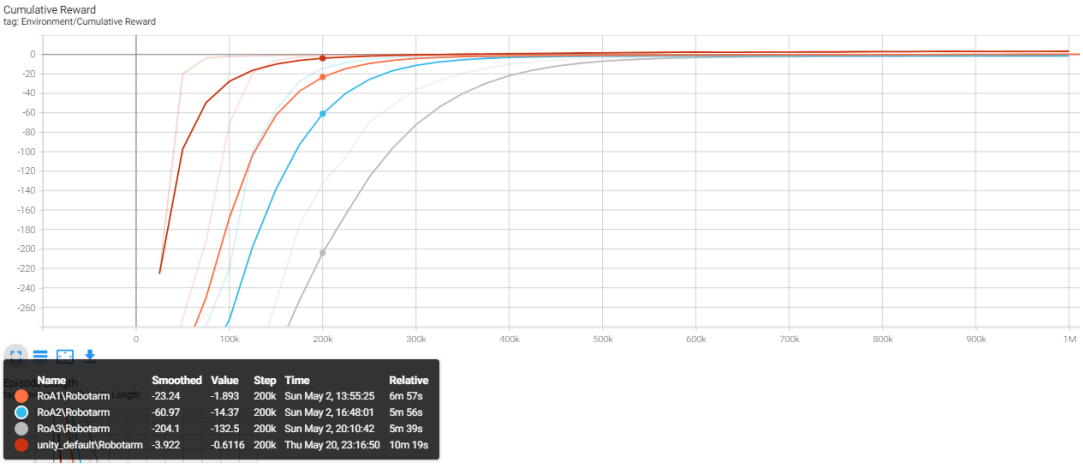

Abbildung 4.4: Tensorboard: Cumulative Reward

Ergebnisse Cumulative Rewward

Der Zuwachs der Punkte der Agents ist in der Tabelle 4.3 zu sehen. Je mehr Schritte absolviert wurden, desto mehr Punkte wurden erhalten. Der erste Durchgang mit einer batch size von 512 (A1, B1, C1) zeigt, dass der Unterschied zwischen der Punkteanzahl der ersten 200k Schritten und der letzten 1.000k nicht sehr groß ist, jedoch nach spätestens 1.000k Schritten ein Pluswert erreicht wird, teilweise auch schon nach 600k Schritten. Der Zuwachs in diesem Training betrug maximal 3,75 Punkten von 200k auf 1.000k (B1). Der zweite Durchgang mit einer batch size von 1.024 zeigt mehr Zuwachs, wenn aber auch keine der Trainingsdurchgänge eine positiven Punktestand erreicht hat. Den größten Punktgewinn gab es bei A2, mit 32 hidden units, 13,21 war die Differenz zwischen Schritt 200k und Schritt 1.000k. Im dritten Durchgang sieht man erstaunlich gut, wie schnell die Punkte kumuliert wurden im Gegensatz zu den Ergebnissen der vorherigen Konfigurationen. A3 hat satte 130,89 Punkte vom Schritt 200k zu Schritt 1.000k kumuliert. Die Ergebnisse zeigen, dass der Punktgewinn bei einer Anzahl von 32 hidden units und einer batch size von 2.048 bei weitem größer. Dementsprechend bedeuten mehr eine größere batch size mit einer geringeren Anzahl an hidden units auch eine

schnellere Punktekumulierung.

Parameterauswirkung auf Cumulative Reward					
Schritte	200k	400k	600k	800k	1 000k
A1	-1.89	-1.16	-0.51	-0.09	0.41
B1	-1.47	-0.37	0.67	1.25	2.28
C1	-1.53	-0.35	0.39	0.95	1.15
A2	-14.37	-1.79	-1.32	-1.23	-1.16
B2	-4.73	-1.47	-1.08	-0.76	-0.65
C2	-4.51	-1.37	-0.68	-0.39	-0.17
A3	-132.50	-10.08	-2.48	-1.83	-1.61
B3	-78.69	-7.19	-2.20	-1.67	-1.50
C3	-67.34	-4.03	-1.75	-1.46	-1.39

Tabelle 4.3: Parameterauswirkung auf Cumulative Reward

Ergebnisse Policy Loss

Im ersten Durchgang ist der Wert höher, demnach wird auch die Strategie öfter angepasst. Im Laufe des Trainings gibt es sprunghafte Anstiege aber auch Anpassungen nach unten. Die Ergebnisse mit des ersten Durchgangs mit einer batch size von 512 zeigen an, dass eine optimale Strategie krampfhaft gesucht, jedoch nicht unbedingt auch gefunden wurde. Der Wert sollte sich allmählich nach unten orientieren, was im Falle von B1 und C1 nicht passiert. Ähnlich verhält es sich im zweiten Durchgang mit einer batch size von 1.024, unabhängig von der Anzahl der hidden units. Hier ist zwar der Wert grundsätzlich niedriger, jedoch gibt es immer noch sprunghafte Entwicklungen nach oben sowie nach unten. Anders wiederum im letzten Durchgang mit einer batch size von 2.048. Hier zeigen zumindest die Durchgänge A3 und C3 eine kurzzeitige Stabilisierung der Werte und ein stetiges Abflachen der Kurve. In diesem Zusammenhang hat auch hier die Konfiguration mit der höheren batch size gewonnen.

Parameterauswirkung auf Policy Loss					
Schritte	200k	400k	600k	800k	1 000k
A1	0.0344	0.0367	0.0366	0.0311	0.0314
B1	0.0341	0.0346	0.0375	0.0368	0.0340
C1	0.0326	0.0328	0.0327	0.0380	0.0365
A2	0.0283	0.0216	0.0198	0.0252	0.0239
B2	0.0171	0.0200	0.0251	0.0214	0.0210
C2	0.0269	0.0263	0.0214	0.0221	0.0239
A3	0.0171	0.0167	0.0160	0.0160	0.0149
B3	0.0126	0.0180	0.0158	0.1345	0.0163
C3	0.0178	0.0154	0.0167	0.0167	0.0149

Tabelle 4.4: Parameterauswirkung auf Policy Loss

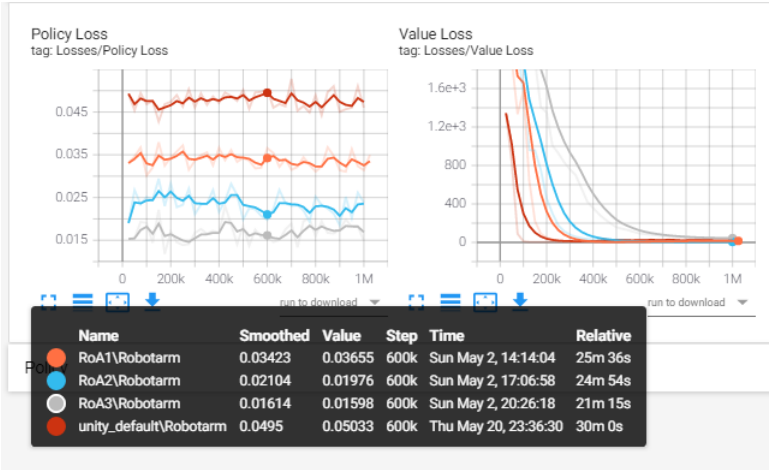

Abbildung 4.5: Tensorboard: Policy & Value Loss

Ergebnisse Value Loss

In diesen Ergebnissen zeigt sich wieder das Potential der größeren batch size, welcher wieder alle anderen Konfigurationen übertrifft. Fangen wir aber mit dem ersten Durchgang an. Bei A1 sieht man bei 200k den hohen Anfangswert, der zeigt, dass das Modell die Vorhersage noch nicht gut beherrscht, je weiter der Trainingsprozess andauert, desto besser wird die Vorhersage. Anders herum die anderen zwei Durchgänge B1 und C1, in denen sich die Vorhersage sogar verschlechtert. Im zweiten Durchgang mit einer batch size von 1.024 sind die Werte stabiler im Abstieg der Werte und somit in der Sicherheit der Vorhersage. A2 startet mit einem hohen Wert, der wiederum zeigt, dass das Modell versucht den nächsten Schritt vorherzusagen, aber scheitert. Am Ende des Trainings ergibt sicher aber ein niedrigerer Wert als in den anderen Durchgängen zuvor. Das beste Ergebnis mit zusätzlich dem höchsten Startwert erreicht der Durchgang C3 mit einer batch size von 2.048 und 128 hidden units. Der Startwert übertrifft alle anderen, wird stetig weniger bis am Ende sich der Wert relativ stabilisiert und den niedrigsten Wert hat von allen anderen Durchgänge. Hier ist der klare Gewinner die Konfiguration mit einer batch size von 2.048 und einer Anzahl von 128 hidden units.

Parameterauswirkung auf Value Loss					
Schritte	200k	400k	600k	800k	1 000k
A1	49.22	3.39	11.29	11.88	11.17
B1	6.24	7.18	12.64	62.48	29.92
C1	4.69	8.71	13.56	18.19	21.38
A2	459.90	30.34	9.29	5.33	4.84
B2	120.20	4.50	3.50	5.90	6.93
C2	89.05	5.46	7.25	8.72	10.48
A3	1351.00	358.40	93.48	51.86	42.60
B3	976.80	175.80	38.91	19.97	15.84
C3	1493.00	60.62	7.27	3.68	3.50

Tabelle 4.5: Parameterauswirkung auf Value Loss

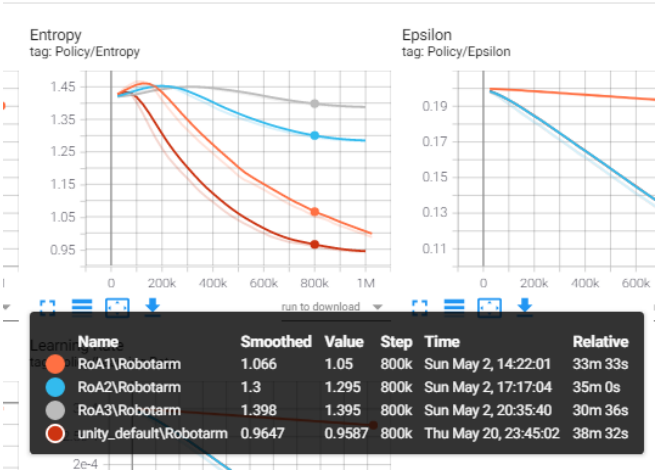

Abbildung 4.6: Tensorboard: Entropy

Ergebnisse Entropy

Der Vergleich dieser Ergebnisse zeigt, dass die verschiedenen Konfigurationen, unabhängig von der batch size, ähnliche Startwerte und einen ähnlichen Abwärtstrend aufweisen. Auch der weitere Verlauf zeigt nur wenige Differenzen zwischen den neun Trainingsdurchgängen auf. Trotzdem zeigen zwei Trainingsdurchläufe, A1 und B1, einen stärkeren Abwärtstrend und somit eine weniger zufällige Entscheidung der nächsten Aktion. Trotzdem zeigen wie schon erwähnt alle der Trainingsdurchgänge einen Trend der abwärts zeigt, daher sollte man auch hier auf ein Parameterkonstellation zurückgreifen, mit einer größeren batch size.

Parameterauswirkung auf Entropy					
Schritte	200k	400k	600k	800k	1 000k
A1	1.409	1.245	1.134	1.050	0.995
B1	1.352	1.194	1.087	0.998	0.928
C1	1.361	1.235	1.146	1.092	1.074
A2	1.454	1.389	1.329	1.295	1.284
B2	1.434	1.357	1.303	1.276	1.268
C2	1.432	1.343	1.291	1.263	1.253
A3	1.447	1.446	1.417	1.395	1.387
B3	1.431	1.418	1.382	1.360	1.353
C3	1.437	1.423	1.385	1.365	1.361

Tabelle 4.6: Parameterauswirkung auf Entropy

Vergleich: eigene Ergebnisse mit Ergebnisse Unity Defaultwerte

Um die Ergebnisse mit einem externem Modell vergleichen zu können, wurde ein letzter Trainingsdurchgang durchgeführt mit den Defaultwerten von Unity´s ML-Agents. Der Vergleich dieser Ergebnisse soll zeigen, wie stark ein Modell mit optimierten Parameter gegenüber nicht optimierten Parametern sein kann. Die Werte der kumulativen Belohnung sprechen eindeutig für die Unity-Parameter. Gegenüber A1, dem besten der neun Trainings, hat Unity bereits

bei 400.000 Schritten ein positives Belohnungskonto aufzuweisen und steigt am Ende bis auf 3.50. A1 war ebenfalls in Bezug auf die Entropie die beste Variante und zeigt auch hier, dass es mit Unity aufnehmen kann. Am Ende sind die Werte bis auf eine Nachkommastelle gleich. Für den Vergleich der Loss-Ergebnisse wurde Durchgang A3 für den Vergleich der Policy Loss und C3 für den Vergleich der Value Loss herangezogen. A3 zeigt hier die Stärke gegenüber Unity, die Strategie wird weniger oft geändert. Unity hingegen hat einen leichten Zuwachs der Strategiewechselhäufigkeit, diese wird aber bis zum Ende wieder gesenkt. Beim Ergebnis der Value Loss sieht es ganz anders aus. C3 schafft es von einem recht hohen Wert (1.493) auf einen sehr niedrigen Wert (3,5) zu sinken. Unity hingegen schafft es nicht den niedrigen Anfangswert zu halten, jedoch sieht man, dass der Wert sich am Ende relativ gut stabilisiert hat. Aus diesen Ergebnissen kann dennoch die Schlussfolgerung gezogen werden, dass die Unity-Werte weit mehr Punkte einbringen, unabhängig von der schlechteren Bilanz im Bereich der Losses.

Ergebnisse mit Unity Defaultwerten					
C. Rew.	200k	400k	600k	800k	1 000k
A1	-1.89	-1.16	-0.51	-0.09	0.41
Unity	-0.61	1.10	2.58	3.05	3.50
Policy L.	200k	400k	600k	800k	1 000k
A3	0.0171	0.0167	0.0160	0.0160	0.0149
Unity	0.0474	0.0471	0.0503	0.0490	0.0460
Value L.	200k	400k	600k	800k	1 000k
C3	1493.00	60.62	7.27	3.68	3.50
Unity	5.13	14.23	25.17	21.3	20.67
Entropy	200k	400k	600k	800k	1 000k
A1	1.409	1.245	1.134	1.050	0.995
Unity	1.261	1.096	0.100	0.959	0.943

Tabelle 4.7: Ergebnisse Unity Defaultwerte und Vergleich mit besten Werten

5 Zusammenfassung & Ausblick

Das Ziel dieser Arbeit war es, einen Überblick über den Bereich der künstlichen Intelligenz zu verschaffen, genauer dem Teilbereich Deep Learning aus dem KI-Bereich Machine Learning. Verschiedene DL-Themen wurden erläutert, damit auch ein/e LeserIn ein Grundverständnis aufbauen und die Ergebnisse verstehen zu können. Zum besseren Verständnis des späteren Experiments wurden drei Ansätze der Robotik aufgezeigt, mit dem Fokus auf den DRL-Ansatz. Die Kombination eines DRL-Netzes mit einem Roboter wurde in einem Unity-Projekt simuliert. Die Ergebnisse verschiedener Parameter wurden sowohl untereinander, als auch mit den Defaultwerten von Unity.

Schlussfolgernd kann gesagt werden, dass der klassische Ansatz für viele Anwendungen in der Produktion von Waren noch immer vorteilhaft ist, zum Teil auch, weil einem CNC-Fräser oder Maschinenbautechniker die Expertise im Bereich der künstlichen Intelligenz fehlt oder diese nur gering vorhanden ist. In der Massenfertigungen wird ein Programm eingespielt und wenig bis gar nicht verändert, wenn Parameter richtig eingestellt wurden - hier wird im Normalfall keine KI benötigt. Hinsichtlich der Robotik in sich ständig veränder- und unvorhersehbaren Umgebungen kann jedoch sehr wohl auf ein DRL-Modell zurückgegriffen werden. Autonome Fahrzeuge können so von A nach B fahren, unabhängig vom Wetter oder Verkehrsfluss. Roboter im Asteroidenbergbau oder im Aufbau einer Mond- und/oder Marsbasis können so autonome Abhilfe schaffen. Bis aber ein DRL-Modell soweit ist, auf der Straße eigenständig entscheiden zu können und zu dürfen, kann hier der regelbasierte Ansatz eingesetzt werden. Begrenzt wird dieser Ansatz von dem im Vorhinein überlegten Szenarien und Use Cases und auf diese programmierten Bewegungen des Roboters. Ein regelbasierter Roboter auf einem anderen Planeten oder in Katastrophengebieten kann nur so weit helfen, wie die Entwickler und Programmierer gedacht haben. DRL-Modelle werden in Zukunft eine größere Rolle in der Robotik spielen, auch weil Kosten gespart werden können durch sich ständig optimierende Strategien und somit die involvierenden Prozesse, weniger dafür besser geschultes Personal wird die Regel sein. Die Vorteile einer Robotersimulation jeglicher Art liegen auf der Hand: Kosten können in der Entwicklung gespart werden, indem der Roboter oder die Maschine zuerst konzipiert und virtuell gebaut wird. Der Tausch durch teurer Ersatzteile kann großteils vermieden werden, wenn der Prozess vorab simuliert werden kann. Maschinen werden nicht blockiert durch die Programmierung an der Maschine. Auch die Simulierung in Virtual Reality kann viele Fehler und Probleme abfangen durch den Aspekt des Live-Erlebnisses der Maschine oder des Roboters in der späteren tatsächlichen Umgebung. Außerdem kann die Ausführung in VR ebenfalls Kosten einsparen, wenn neues Personal eingeschult werden muss: Brille aufsetzen, Training absolvieren und schon ist die/der neue MitarbeiterIn einsatzbereit!

Weiterführende Arbeiten können auf die hier gezeigten Ergebnisse und verwendeten Parameter in dieser Arbeit als Grundlage verwenden. Der Einsatz des Algorithmus in dieser Arbeit könnte in einem physischen Roboter implementiert werden und die Ergebnisse verglichen. So könnte gezeigt werden, ob und wie sehr die Ergebnisse der Simulation eines virtuellen Roboters von den Ergebnissen eines realen Roboters abweichen.

Literaturverzeichnis

[BBF00] Selmer Bringsjord, Paul Bello, and David Ferrucci. Creativity, the turing test, and the (better) lovelace test. *Minds and Machines*, 2000. 4

[BHS98] Jörg Bartenschlager, Hans Hebel, and Georg Schmidt. *Programmierung von Industrierobotern*. Vieweg+Teubner Verlag, 1998. 16

[Cho20] Ankit Choudhary. Deep q-learning: An introduction to deep reinforcement learning, Apr 2020. 18, 38

[Cor17] Peter Corke. *Robotics, vision and control fundamental algorithms in MATLAB®*. Springer, 2017. 15, 28

[CP19] R. Carter and S. Parker. *The Human Brain Book*. DK Publishing, 2019. 10

[Des08] René Descartes. *Meditations on First Philosophy With Selections from the Objections and Replies*. Oxford World´s Classics, 2008. 6

[Fit44] Frederic B. Fitch. Warren s. mcculloch and walter pitts. a logical calculus of the ideas immanent in nervous activity. bulletin of mathematical biophysics, vol. 5 (1943), pp. 115–133. *Journal of Symbolic Logic*, 9(2):49–50, 1944. 10

[GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`. 9

[Ger17] Aurelien Geron. *Hands-on machine learning with Scikit-Learn and TensorFlow : concepts, tools, and techniques to build intelligent systems*. O'Reilly Media, 2017. 6, 7, 8, 10, 11, 17, 19, 38

[Hea08] Jeff Heaton. *Introduction to Neural Networks for Java, 2nd Edition*. Heaton Research, Inc., 2008. 9

[KBB+15] Rudolf Kruse, Christian Borgelt, Christian Braune, Frank Klawonn, Christian Moewes, and Matthias Steinbrecher. *Computational Intelligence Eine methodische Einführung in Künstliche Neuronale Netze, Evolutionäre Algorithmen, Fuzzy-Systeme und Bayes-Netze*. Springer Vieweg, 2015. 13

[KBP13] Jens Kober, J. Bagnell, and Jan Peters. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research*, 32:1238–1274, 09 2013. 17

[Kur06] Ray Kurzweil. *The Singularity Is Near: When Humans Transcend Biology*. Penguin (Non-Classics), 2006. 4

[LBBH98] Yann Lecun, Leon Bottou, Y. Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86:2278 – 2324, 12 1998. 11

[LCB+18] Jey Han Lau, Trevor Cohn, Timothy Baldwin, J. Brooke, and A. Hammond. Deep-speare: A joint neural model of poetic language, meter and rhyme. *ArXiv*, 2018. 13

[LM18] Yaniv Leviathan and Yossi Matias. google duplex: an ai system for accomplishing real-world tasks over the phone. Website, 2018. `https://ai.googleblog.`

com/2018/05/duplex-ai-system-for-natural-conversation.html; abgerufen am 08. April 2021. 5

[Met01] T. Metzinger. Postbiotisches bewußtsein: Wie man ein künstliches subjekt baut und warum wir es nicht tun sollten. 2001. 4

[MM08] Maja Matarić and François Michaud. *Behavior-Based Systems*, pages 891–909. 01 2008. 21, 38

[Moo06] G. E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006. 4

[O'L19] Daniel O'Leary. Google's duplex: Pretending to be human. *Intelligent Systems in Accounting, Finance and Management*, 26:46–53, 01 2019. 4, 5

[Ras16] Tariq Rashid. *Make Your Own Neural Network*. CreateSpace Independent Publishing Platform, 2016. 13

[RN12] S. Russell and P. Norvig. *Künstliche Intelligenz - ein moderner Ansatz*. Pearson, 2012. 3, 5, 8, 10, 11, 12, 13, 18, 23, 38

[Ros58] F. Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 1958. 11

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018. 13, 16, 17, 18, 19, 20

[SEJ+20] Andrew Senior, Richard Evans, John Jumper, James Kirkpatrick, Laurent Sifre, Tim Green, Chongli Qin, Augustin Žídek, Alexander Nelson, Alex Bridgland, Hugo Penedones, Stig Petersen, Karen Simonyan, Steve Crossan, Pushmeet Kohli, David Jones, David Silver, Koray Kavukcuoglu, and Demis Hassabis. Improved protein structure prediction using potentials from deep learning. *Nature*, 2020. 10

[SHM+16] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 2016. 10

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. 2017. 25

[Tec] Unity Technologies. Ml-agents documentation. Website. https://github.com/Unity-Technologies/ml-agents/blob/main/docs/; abgerufen am 01. Mai 2021. 25, 38

[Tur50] A. M. Turing. Computing machinery and intelligence. *Mind*, 1950. 4, 38

Abbildungsverzeichnis

2.1 Veranschaulichung Turing Test [Tur50] 4
2.2 Unterschied Supervised und Unsupervised Learning [Ger17] 8
2.3 Vereinfachte Darstellung des Aufbaus einer Nervenzelle - ein Neuron [RN12][Ger17] 11

3.1 Darstellung Markov'sches Entscheidungsproblem [Ger17] 17
3.2 Q-Learning & Deep Q-Learning [Cho20] 18
3.3 Darstellung Belohnungssystem [Ger17] 19
3.4 Darstellung einer regelbasierten Aktionsausführung [MM08] 21

4.1 Python API [Tec] 25
4.2 Unity-Inspektor: Joints als 'Axis' bezeichnet, Endeffektor als 'Axis06Tip' und Zielobjekt als 'Component' 28
4.3 Roboter in Unity 29
4.4 Tensorboard: Cumulative Reward 30
4.5 Tensorboard: Policy & Value Loss 32
4.6 Tensorboard: Entropy 33

Tabellenverzeichnis

4.1 Parameterübersicht Robotersimulation 27
4.2 Belohnungssystem Robotersimulation 27
4.3 Parameterauswirkung auf Cumulative Reward 31
4.4 Parameterauswirkung auf Policy Loss 31
4.5 Parameterauswirkung auf Value Loss 32
4.6 Parameterauswirkung auf Entropy 33
4.7 Ergebnisse Unity Defaultwerte und Vergleich mit besten Werten 34