

MASTER THESIS  
Sandra Verena Lassahn

# **3D-Simulation und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Labyrinths**

---

FAKULTÄT TECHNIK UND INFORMATIK  
Department Informations- und Elektrotechnik

Faculty of Engineering and Computer Science  
Department of Information and Electrical Engineering



Sandra Verena Lassahn

# 3D-Simulation und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Labyrinths

Masterarbeit eingereicht im Rahmen der Masterprüfung  
im Studiengang *Master of Science Automatisierung*  
am Department Informations- und Elektrotechnik  
der Fakultät Technik und Informatik  
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuer Prüfer: Prof. Dr.-Ing. Marc Hensel  
Zweitgutachterin: Prof. Dr. rer. nat. Ulrike Herster

Eingereicht am: 27. September 2024



**Sandra Verena Lassahn**

**Thema der Arbeit**

3D-Simulation und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Labyrinths

**Stichworte**

Reinforcement Learning, 3D-Simulation, BRIO-Labyrinth, Deep Q-Learning, Motorsteuerung

**Kurzzusammenfassung**

In dieser Masterarbeit wird ein KI-gesteuertes Labyrinth mittels Reinforcement Learning entwickelt. Zunächst wird eine detaillierte 3D-Simulation mit mehreren Spielplatten erstellt, die als Trainings- und Evaluierungsumgebung dient. Anschließend wird ein Demonstrator des automatisierten BRIO-Labyrinths mit Bildverarbeitung zur Kugelerkennung realisiert.

**Sandra Verena Lassahn**

**Title of Thesis**

3D simulation and prototypical setup of a labyrinth controlled by reinforcement learning

**Keywords**

Reinforcement Learning, 3D-Simulation, BRIO-Labyrinth, Deep Q-Learning, Motor Control

**Abstract**

In this master's thesis, an AI-controlled labyrinth is developed using reinforcement learning. First, a detailed 3D simulation with several game boards is created, which serves as a training and evaluation environment. Subsequently, a demonstrator of the automated BRIO-labyrinth with image processing for ball recognition is realized.



# Inhaltsverzeichnis

<b>Abbildungsverzeichnis</b>	<b>ix</b>
<b>Tabellenverzeichnis</b>	<b>xiii</b>
<b>Listings</b>	<b>xv</b>
<b>Abkürzungen</b>	<b>xvii</b>
<b>1 Einleitung</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Zielsetzung . . . . .	2
1.3 Aufbau der Arbeit . . . . .	2
<b>2 Grundlagen</b>	<b>3</b>
2.1 BRIO Labyrinth . . . . .	3
2.2 Maschinelles Lernen . . . . .	4
2.3 Künstliches neuronales Netzwerk . . . . .	5
2.3.1 Deep Learning . . . . .	6
2.3.2 Architekturen von künstlichen neuronalen Netzen . . . . .	7
2.3.3 Aktivierungsfunktionen . . . . .	10
2.3.4 Gradientenverfahren . . . . .	16
2.3.5 Backpropagation (Fehlerrückführung) . . . . .	19
2.3.6 Verlustfunktionen . . . . .	21
2.3.7 Optimierer . . . . .	23
2.3.8 Gewichtsinitialisierung . . . . .	25
2.3.9 Batch-Normalisierung . . . . .	28
2.3.10 Häufige Probleme beim Trainieren von neuronalen Netzen . . . . .	29
2.4 Reinforcement Learning . . . . .	31
2.4.1 Grundlagen . . . . .	31
2.4.2 Q-Learning . . . . .	34

## *Inhaltsverzeichnis*

---

2.4.3	Deep Q-Learning . . . . .	36
2.4.4	SARSA . . . . .	36
<b>3</b>	<b>Stand der Technik</b>	<b>39</b>
3.1	DFKI . . . . .	39
3.2	ETH Zürich (CyberRunner) . . . . .	42
3.3	Kantonsschule Stadelhofen . . . . .	43
3.4	Linköping University . . . . .	44
3.5	Zusammenfassung vorhandener Arbeiten . . . . .	45
<b>4</b>	<b>Anforderungsanalyse</b>	<b>47</b>
4.1	Systemumgebung . . . . .	47
4.2	Stakeholder . . . . .	48
4.2.1	Auftraggeber . . . . .	48
4.2.2	Entwicklerin . . . . .	49
4.2.3	Nutzer . . . . .	49
4.2.4	Personen im Bereich Weiterentwicklung . . . . .	50
4.3	Virtuelle Umgebung . . . . .	50
4.3.1	Anwendungsfälle . . . . .	50
4.3.2	Anforderungen . . . . .	52
4.4	Physischer Demonstrator . . . . .	55
4.4.1	Anwendungsfälle . . . . .	55
4.4.2	Anforderungen . . . . .	56
<b>5</b>	<b>Konzept</b>	<b>59</b>
5.1	Simulation und Software . . . . .	59
5.1.1	Programmiersprache . . . . .	59
5.1.2	Simulationsumgebung . . . . .	60
5.1.3	Machine Learning Framework . . . . .	60
5.1.4	Zustandsrepräsentation und Aktionsauswahl . . . . .	61
5.1.5	Belohnung . . . . .	63
5.1.6	Agent . . . . .	68
5.2	Hardware . . . . .	71
5.2.1	Mechanik . . . . .	72
5.2.2	Motorenauswahl . . . . .	76

<b>6 Entwicklung der virtuellen Umgebung</b>	<b>79</b>
6.1 Übersicht der virtuellen Trainingsumgebung . . . . .	79
6.2 3D-Modell . . . . .	79
6.2.1 Layouts und Geometrie . . . . .	81
6.2.2 Render3D . . . . .	84
6.3 Physikalische Eigenschaften der Kugelbewegung . . . . .	87
6.3.1 Verhalten einer rollenden Kugel auf einer schiefen Ebene . . . . .	87
6.3.2 Bestimmung der Rollreibungszahl . . . . .	90
6.3.3 Kollisionsdetektion zwischen Kugel und Wand . . . . .	91
6.3.4 Verhalten der Kugel nach der Kollision mit einer Kante . . . . .	92
6.3.5 Bestimmung der Stoßzahl . . . . .	95
6.3.6 Verhalten der Kugel nach der Kollision mit einer Ecke . . . . .	96
6.3.7 Kollisionsdetektion zwischen Kugel und Loch . . . . .	99
6.4 OpenAI Gym-Environment . . . . .	99
6.4.1 Environment Grundlagen . . . . .	99
6.4.2 Implementierung . . . . .	100
6.4.3 Drehbewegung der Spielplatte . . . . .	102
6.5 Rewards . . . . .	104
6.5.1 Rewards by Areas . . . . .	104
6.5.2 Rewards by Thresholds . . . . .	106
6.6 Deep Q-Learning . . . . .	107
6.6.1 Replay-Buffer . . . . .	108
6.6.2 QNet . . . . .	110
6.6.3 Agent . . . . .	110
6.6.4 Main . . . . .	112
<b>7 Entwicklung des physischen Demonstrators</b>	<b>119</b>
7.1 Hardware . . . . .	119
7.1.1 Elektronik . . . . .	119
7.1.2 3D-gedruckte Elemente . . . . .	122
7.1.3 Eigene Spielplatten . . . . .	122
7.2 Software . . . . .	123
7.2.1 physische Umgebung . . . . .	124
7.2.2 Servokommunikation . . . . .	126
7.2.3 Bildverarbeitung . . . . .	129

*Inhaltsverzeichnis*

---

<b>8 Evaluierung</b>	<b>131</b>
8.1 Tests zur RL-Parameterbestimmung . . . . .	131
8.1.1 Spielplatte HOLES_2 . . . . .	131
8.1.2 Spielplatte HOLES_2_VIRTUAL . . . . .	143
8.1.3 Spielplatte HOLES_0 . . . . .	144
8.1.4 Spielplatte HOLES_0_VIRTUAL . . . . .	146
8.1.5 Spielplatte HOLES_8 . . . . .	146
8.1.6 Spielplatte HOLES_21 . . . . .	150
8.1.7 Weitere Testerfahrungen . . . . .	151
8.2 Anforderungsprüfung . . . . .	151
8.2.1 Virtuelle Umgebung . . . . .	151
8.2.2 Physischer Demonstrator . . . . .	154
<b>9 Fazit und Ausblick</b>	<b>157</b>
<b>10 Danksagung</b>	<b>159</b>
<b>Literaturverzeichnis</b>	<b>161</b>
<b>A Anhang</b>	<b>169</b>
<b>Selbstständigkeitserklärung</b>	<b>171</b>

# Abbildungsverzeichnis

2.1	Typische BRIO Labyrinthe . . . . .	4
2.2	Aufbau eines künstlichen neuronalen Netzes . . . . .	6
2.3	Schwellwert Aktivierungsfunktion . . . . .	7
2.4	Aufbau Perzeptron . . . . .	8
2.5	Netzarchitekturen . . . . .	9
2.6	Faltungsnetz . . . . .	9
2.7	Pooling . . . . .	10
2.8	Sigmoid Aktivierungsfunktion . . . . .	12
2.9	ReLU Aktivierungsfunktion . . . . .	13
2.10	Leaky ReLU Aktivierungsfunktion . . . . .	14
2.11	ELU Aktivierungsfunktion . . . . .	15
2.12	Gradientenverfahren am Beispiel einer Bergsteigers . . . . .	16
2.13	Herausforderungen beim Gradientenverfahren . . . . .	18
2.14	Over- und Underfitting bei einer Klassifikationsaufgabe . . . . .	30
2.15	Grundprinzip RL . . . . .	32
2.16	RL Standard-Framework . . . . .	32
2.17	Vergleich Q-Learning und DQN . . . . .	37
2.18	SARSA . . . . .	37
2.19	Vergleich Q-Learning und SARSA . . . . .	38
3.1	Motoransteuerung mit Riemen (DFKI) . . . . .	39
3.2	Gesamtaufbau (DFKI) [50] . . . . .	40
3.3	Ballmagazin (DFKI) . . . . .	41
3.4	Simulationsumgebung (DFKI) [50] . . . . .	41
3.5	CyberRunner (ETH Zürich) [15] . . . . .	42
3.6	Prototypischer Aufbau (Kantonsschule Stadelhofen) [2] . . . . .	43
3.7	Labyrinthweiterung der ersten Masterarbeit (Linköping University) [24]	44

3.8	Simulation und genutzte Policy Parameter der zweiten Masterarbeit (Lin- köping University) [23] . . . . .	45
4.1	Systemumgebung [13] . . . . .	48
4.2	Anwendungsfalldiagramm der virtuellen Umgebung . . . . .	51
4.3	Anwendungsfalldiagramm des physischen Demonstrators . . . . .	55
5.1	Fortschritt durch Kacheln und Zwischenziele . . . . .	65
5.2	Interpolationsarten . . . . .	67
5.3	Direkte Anbringung auf der Welle [12] . . . . .	72
5.4	Anbringung über ein Verbindungsstange . . . . .	73
5.5	Innen befestigter Motor [24] . . . . .	73
5.6	Nutzung von Riemen [17] . . . . .	74
5.7	Nutzung von Zahnrädern . . . . .	75
6.1	Klassenübersicht der virtuellen Umgebung . . . . .	80
6.2	3D-Simulation . . . . .	81
6.3	Labyrinthe und Spielplatten . . . . .	82
6.4	Klassendiagramm der Layouts und geometrischen Abmessungen . . . . .	83
6.5	Klassendiagramm Render3D und Ballphysik . . . . .	85
6.6	Koordinatenursprung . . . . .	86
6.7	Box Objekt . . . . .	86
6.8	Kräfte einer rollenden Kugel an einer schiefen Ebene . . . . .	87
6.9	Aufbau zur Bestimmung der materialspezifischen Koeffizienten . . . . .	91
6.10	Kollision zwischen Kugel und Wand . . . . .	92
6.11	Kugelverhalten nach einer Kollision eines inelastischen Stoßes . . . . .	93
6.12	Schaubild zur Bestimmung der Stoßzahl . . . . .	95
6.13	Geschwindigkeitsanpassung bei Ecken Kollision . . . . .	97
6.14	Positionsanpassung bei Eckenkollision . . . . .	98
6.15	Kollision zwischen Kugel und Loch . . . . .	99
6.16	Klassendiagramm der Labyrinth Environment . . . . .	101
6.17	Drehbewegung der Spielplatte . . . . .	104
6.18	Klassendiagramm Rewards . . . . .	105
6.19	Schwellenüberquerungsberechnung . . . . .	107
6.20	DQN Algorithmus [51] . . . . .	108
6.21	Klassendiagramm des DQN-Agenten . . . . .	109
6.22	Benutzeroberfläche für das Training . . . . .	113

6.23 Klassendiagramm der Main . . . . .	115
6.24 Benutzeroberfläche für das Evaluieren . . . . .	117
6.25 Fehlermeldungen der Benutzeroberfläche . . . . .	117
7.1 Verdrahtungsplan . . . . .	120
7.2 PWM-Signale . . . . .	121
7.3 Kameragestell . . . . .	122
7.4 CAD-Zeichnungen . . . . .	123
7.5 Realisierte Spielplatten . . . . .	124
7.6 Klassendiagramm zum physischen Demonstrator . . . . .	125
7.7 Kalibrationshinweis zur Bildverarbeitungskalibration . . . . .	126
7.8 Kalibrierung für die Bildverarbeitung . . . . .	127
7.9 Bestätigung des Episodenstarts . . . . .	127
7.10 Bestimmung des Winkel-Puls-Verhältnisses . . . . .	129
8.1 Kacheln und Zwischenziele der Spielplatte HOLES_2 . . . . .	137
8.2 Schwellen der Spielplatte HOLES_2 . . . . .	138
8.3 Kacheln und Zwischenziele der HOLES_2_VIRTUAL Spielplatte . . . . .	143
8.4 Kacheln und Zwischenziele der Spielplatte HOLES_8 . . . . .	148

*Abbildungsverzeichnis*

---

# Tabellenverzeichnis

3.1	Kurzzusammenfassung vorhandener Arbeiten . . . . .	46
5.1	Motor Anbringungsübersicht . . . . .	75
7.1	Elektronik-Bauteilkomponenten . . . . .	120
7.2	Weitere Bauteilkomponenten . . . . .	121
8.1	Schichtentest . . . . .	134
8.2	Verlustfunktionstest . . . . .	135
8.3	Aktivierungsfunktionstest . . . . .	135
8.4	Belohnungen mit Ziel (und Loch) . . . . .	136
8.5	Auswertung zu Ziel (und Loch) . . . . .	136
8.6	Belohnungen mit Labyrinthfortschritt . . . . .	137
8.7	Auswertung zu Kacheln und Zwischenzielen . . . . .	138
8.8	Belohnungen mit Schwellenbelohnung . . . . .	139
8.9	Auswertung zu Schwellen . . . . .	139
8.10	Epsilonontest . . . . .	139
8.11	Lernperiod-/Batchsizetest . . . . .	140
8.12	Gammatest . . . . .	140
8.13	Zustandsraumtest . . . . .	141
8.14	Aktionsraumtest . . . . .	141
8.15	Seedtest . . . . .	142
8.16	Test ohne Seed . . . . .	142
8.17	Platzierungstest Kacheln und Zwischenziele . . . . .	143
8.18	Parameter des neuronalen Netzes der HOLES_8 Spielplatte . . . . .	147
8.19	Gesamtausgaben . . . . .	155
A.1	Packetinstallation . . . . .	169

*Tabellenverzeichnis*

---

# Listings

6.1	Main . . . . .	116
8.1	Belohnungen HOLES_2 . . . . .	133
8.2	Belohnungen HOLES_0 . . . . .	145
8.3	Belohnungen HOLES_0_VIRTUAL . . . . .	146
8.4	Belohnungen HOLES_8 . . . . .	148
8.5	Belohnungen HOLES_21 . . . . .	151

*Listings*

---

# Abkürzungen

<b>Adam</b>	Adaptive Moment Estimation (adaptive Momentschätzung).
<b>CNN</b>	Convolutional Neuronal Network (Faltungsnetz).
<b>DFKI</b>	Deutsches Forschungszentrum für Künstliche Intelligenz.
<b>DQN</b>	Deep Q-Network.
<b>ELU</b>	Exponential Linear Unit (exponentielle lineare Einheit).
<b>KI</b>	Künstliche Intelligenz.
<b>KNN</b>	Künstliches neuronales Netz.
<b>MAE</b>	Mean Absolut Error (mittlere absolute Abweichung).
<b>MDP</b>	Marcov Decision Process (Markow-Entscheidungsprozess).
<b>MSE</b>	Mean Square Error (mittlere quadratische Abweichung).
<b>PWM</b>	Pulse Width Modulation (Pulsweitenmodulation).
<b>ReLU</b>	Rectified Linear Unit (gleichgerichtete lineare Einheit).
<b>RL</b>	Reinforcement Learning (verstärkendes Lernen).
<b>RMSProp</b>	Root Mean Square Propagation.
<b>SGD</b>	stochastic Gradient Descent (stochastischer Gradientenabstieg).

*Abkürzungen*

---

# 1 Einleitung

Dieses Kapitel bietet einen kurzen Überblick über die Motivation, das Ziel und den Aufbau bzw. die Gliederung der vorliegenden Abschlussarbeit.

## 1.1 Motivation

Maschinelles Lernen ist ein rasant wachsendes Themengebiet, das in den letzten Jahren immense Fortschritte gemacht hat. Künstliche Intelligenz (KI) wird dabei als zukunftsweisende Technologie beschrieben und findet eine breite Anwendung in verschiedensten Bereichen wie der Robotik, bei autonomen Systemen oder Sprachassistenten, im Gesundheitswesen oder der Energieoptimierung. Die Fähigkeit von Systemen selbstständig aus Erfahrungen zu lernen und komplexe Aufgaben in Umgebungen zu bewältigen, bietet ein enormes Potenzial. Gerade in der heutigen Zeit, in der KI zunehmend in den Alltag integriert wird, steigt das öffentliche Interesse an diesem Thema. Um dem wachsenden Interesse und verbreiteteren Einsatz dieser Technologie gerecht zu werden, braucht es gut ausgebildete Fachkräfte mit Expertise, die sich den Herausforderungen dieses Gebiets stellen und innovative Lösungen entwickeln.

Die bemerkenswerten Fähigkeiten, Vielseitigkeit oder auch Leistungsfähigkeit von KI-Systemen zeigen sich beispielsweise auch beim Lösen von verschiedensten komplexen Spielen, in denen die Systeme den Menschen übertreffen [9]. Durch die Entwicklung einer 3D-Simulation und eines physischen Demonstrators kann das Potenzial von KI nicht nur theoretisch, sondern auch praktisch veranschaulicht werden. Ein Labyrinth-Spiel stellt dabei eine hervorragende Möglichkeit dar, ein tieferes Verständnis für dieses Themengebiet zu entwickeln und KI für andere erfahrbar zu machen. Dabei werden unterschiedlichste Disziplinen wie Informatik, Mathematik, Neurowissenschaften, aber auch Elektronik und Mechanik miteinander verbunden.

## **1.2 Zielsetzung**

Das Ziel dieser Masterarbeit ist die Entwicklung eines Reinforcement-Learning-Systems, welches eine Kugel autonom durch ein komplexes Labyrinth navigieren kann. Hierfür wird zunächst eine detaillierte 3D-Simulation entwickelt, die als Trainings- und Evaluierungsumgebung für das System dient. Diese virtuelle Umgebung ermöglicht es, das System unter kontrollierten Bedingungen zu trainieren und seine Leistungsfähigkeit umfassend zu testen. Im Anschluss daran wird ein physischer Demonstrator realisiert, auf den die in der Simulation erlernten Fähigkeiten auf eine physische Hardware übertragen und weiter trainiert werden können.

Der Erfolg des Systems wird daran gemessen, wie effektiv und präzise es in der Lage ist, die Kugel durch das Labyrinth zu steuern und das Ziel zu erreichen, ohne dass die Kugel in Löcher fällt. Um dies zu gewährleisten, werden in dieser Arbeit die theoretischen Grundlagen des maschinellen Lernens eng mit praktischen Implementierungen und umfangreichen Tests verknüpft.

Diese Arbeit stellt eine ausgezeichnete Gelegenheit dar, die Prinzipien des maschinellen Lernens auf praxisnahe Weise zu erforschen, und legt gleichzeitig eine fundierte Basis für zukünftige Weiterentwicklungen und Anwendungen in diesem Bereich. Der Bearbeitungszeitraum ist dabei durch die Rahmenbedingungen einer Masterarbeit festgelegt.

## **1.3 Aufbau der Arbeit**

Die Struktur dieser Abschlussarbeit gliedert sich in verschiedene Kapitel. Nach dieser Einleitung werden in Kapitel 2 die theoretischen Grundlagen des maschinellen Lernens erläutert, die für das Verständnis und die Umsetzung der Arbeit unerlässlich sind. Darauf folgt in Kapitel 3 eine Übersicht von bestehenden autonomen Lösungen ähnlicher Projekte. Kapitel 4 widmet sich der Anforderungsanalyse und definiert dabei die spezifischen Systemumgebungen und -anforderungen, sowie die relevanten Stakeholder. In Kapitel 5 wird das Konzept der Arbeit erläutert. Die Entwicklung und konkrete Umsetzung der virtuellen Umgebung wird in Kapitel 6 behandelt, während Kapitel 7 die Entwicklung des physischen Demonstrators vorstellt. Schließlich erfolgt in Kapitel 8 die Evaluierung verschiedenster Tests und die Überprüfung der Anforderungen. Den Abschluss dieser Arbeit bildet das Kapitel 9 mit einem Fazit und einem Ausblick auf mögliche zukünftige Entwicklungen.

## 2 Grundlagen

Dieses Kapitel bietet eine Einführung in die theoretischen Grundlagen, die für das Verständnis und die Umsetzung eines durch Reinforcement Learning gesteuerten Labyrinths notwendig sind. Zunächst wird das Geschicklichkeitsspiel BRIO Labyrinth vorgestellt, welches die Basis für die Simulation und den Demonstrator bildet. Anschließend wird ein Überblick über das maschinelle Lernen und dessen Gebiete gegeben. Darauf aufbauend werden die Konzepte und Architekturen künstlicher neuronaler Netzwerke sowie deren wesentliche Komponenten und Herausforderungen erläutert. Schließlich wird eine Einführung in Reinforcement Learning und ausgewählte Methoden und Algorithmen gegeben.

### 2.1 BRIO Labyrinth

Bei dem Geschicklichkeitsspiel Kugellabyrinth geht es darum eine Stahlkugel mit Hilfe von zwei Drehköpfen, mit denen man die Spielfläche kippen kann, durch ein Labyrinth zu bewegen. Dabei besteht das Labyrinth aus einem vorgegeben Pfad, entlang dessen die Kugel von einem Startpunkt zu einem Ziel manövriert werden soll. Zusätzlich gibt es Löcher und Wände. Der Spieler muss verhindern, dass die Kugel in eines der vorhandenen Löcher fällt. Je weiter die Kugel durch das Labyrinth rollt, desto höher ist die erreichte Punktzahl des Spielers. Das Spiel wurde im Jahr 1946 von dem schwedischen Spielzeugunternehmen BRIO [11] populär gemacht. Mittlerweile wurde das Unternehmen vom deutschen Spielehersteller Ravensburger übernommen. Mit seinen einfachen Regeln und dennoch anspruchsvollen Herausforderungen hat es sich zu einem Klassiker unter den Familienspielen entwickelt, von dem es mittlerweile eine Vielzahl an Variationen gibt. Zwei klassische Varianten sind in der Abbildung 2.1 dargestellt. Vom traditionellen Holzlababyrinth bis hin zu modernen, futuristischen Designs, die aus verschiedenen Materialien gefertigt sind, gibt es für jeden Geschmack und jede Vorliebe eine passende Version von einer Vielzahl an Herstellern. Wem die zweidimensionalen



(a) Klassisches Holzlababyrinth [14]      (b) Rote Variante mit verschiedenen Übungsplatten [13]

Abbildung 2.1: Typische BRIO Labyrinthe

Labyrinthe zu einfach sind oder Spieler, die eine neue Herausforderung wollen, für die gibt es auch dreidimensionale Labyrinthe. Mit Simulationen, die beispielsweise für Tablets und Smartphones verfügbar sind, können Spieler das Spiel virtuell erleben und die Kugel durch das Labyrinth manövrieren, indem sie ihre Geräte kippen und neigen.

Die Faszination für das Kugellabyrinth liegt nicht nur in seiner spielerischen Herausforderung, sondern auch in den Fähigkeiten, die es fördert. Somit wird das Spiel für seine Kombination aus Geschicklichkeit, Konzentration und Hand-Augen-Koordination geschätzt. Das relativ simple Spiel erfordert außerdem gute feinmotorische Fähigkeiten und räumliches Denkvermögen, wodurch es einiges an Übung benötigt das Spiel zu beherrschen. Zudem gibt es weitere zu berücksichtigende Herausforderungen wie beispielsweise die Haftreibungseffekte zwischen Kugel und Boden bzw. Wänden oder auch Unregelmäßigkeiten in der Spieloberfläche.

Den aktuellen von einem Menschen erzielten Rekord im Lösen des klassischen Holzspiels hält seit 2022 Lars-Göran Danielsson, der eine beeindruckende Zeit von 15,41 Sekunden erzielte [11]. Den generellen Rekord beim Lösen des Labyrinthes hält jedoch eine künstliche Intelligenz. Diese schaffte es die Kugel innerhalb von 14,48 Sekunden [15] zum Ziel zu manövrieren. Auf dieses System namens CyberRunner wird im weiteren Verlauf der Arbeit weiter eingegangen (siehe Kapitel 3.2).

## 2.2 Maschinelles Lernen

Maschinelles Lernen [56] ist ein Teilbereich der künstlichen Intelligenz. Es beschäftigt sich mit der Entwicklung von Algorithmen und statistischen Modellen, die es Computern ermöglichen, Aufgaben auszuführen, ohne explizit dafür programmiert zu sein. Im

Kern basiert maschinelles Lernen auf der Fähigkeit von Systemen, aus Daten zu lernen und Muster zu erkennen, um Vorhersagen oder Entscheidungen zu treffen. Drei Kategorien des maschinellen Lernens, das überwachte (Supervised Learning), unüberwachte (Unsupervised Learning) und verstärkende Lernen (Reinforcement Learning) werden nachfolgend genauer betrachtet.

Beim **überwachten Lernen** werden einem Algorithmus gelabelte (kategorisierte) Daten präsentiert. Diese bestehen aus Eingabedaten und den entsprechenden Ausgabewerten. Das Ziel besteht darin, aus den Daten Muster oder Zusammenhänge zu erkennen, so dass der Algorithmus in der Lage ist neue, nicht gelabelte Daten zu klassifizieren. Hingegen werden beim **unüberwachten Lernen** keine Beziehungen mit den Eingabedaten bereitgestellt, das heißt dem Algorithmus werden keine gelabelten Daten präsentiert. Diese Art des Lernens wird verwendet um Muster oder Strukturen in Eingabedaten zu entdecken, die auf Grund der Datenmenge nicht gleich ersichtlich sind, um diese Daten anschließend klassifizieren zu können. **Reinforcement Learning** (RL) stellt im Bereich maschinellem Lernens den vielversprechendsten Allzweck-Lernalgorihmus dar [78]. Beim RL-Algorithmus werden Anreize (positive Belohnungen) gegeben ein gesetztes Ziel zu erreichen und Aktionen bestraft (negative Belohnungen), die wir nicht wollen. Der RL-Algorithmus versucht seine Belohnungen zu maximieren wodurch bestimmte Verhaltensweisen durch diese Belohnungssignale verstärkt werden. Im Unterkapitel 2.4 wird weiter auf Reinforcement Learning eingegangen. Künstliche neuronale Netze (KNN) machen erst die großen Erfolge von RL-Algorithmen möglich, daher werden zunächst KNN vorgestellt.

## 2.3 Künstliches neuronales Netzwerk

Ein künstliches neuronales Netzwerk [78] ist ein aus mehreren Schichten bestehendes Machine-Learning-Modell. Es ist ein abstrahiertes Modell von verbundenen Neuronen, die der Funktionsweise des menschlichen Gehirns nachempfunden sind. Neuronale Netze bestehen aus drei Arten von Neuronen, den Eingabeneuronen (Input unit), verborgenen Neuronen (Hidden unit) und Ausgabeneuronen (Output unit). Eingabeneuronen erhalten ihre Informationen von der Umgebung. Diese Zustandseingaben werden lediglich weitergeleitet. Verborgene Neuronen sind die Neuronen, die weder den Eingaben noch den Ausgabeneuronen zugeordnet werden können. Sie dienen der internen Weiterverarbeitung der Informationen. Ausgabeneuronen enthalten Informationen, welche an

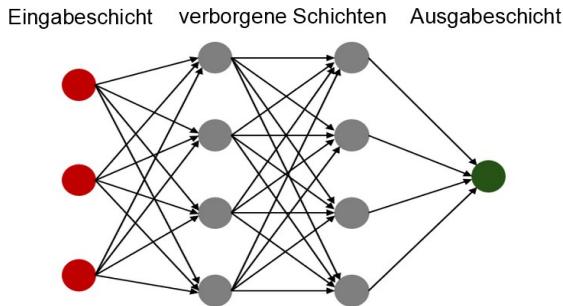


Abbildung 2.2: Aufbau eines künstlichen neuronalen Netzes

die Außenwelt zurückgegeben werden wie beispielsweise die zu wählende Aktion. Übereinander angeordnete Knoten fasst man als Schichten (Layer) zusammen (siehe Abbildung 2.2). Neuronen innerhalb einer Schicht sind normalerweise nicht untereinander verbunden, sondern nur mit den direkt folgenden Schichten. KNN haben nur eine Ein- und Ausgabeschicht, sie können hingegen keine, eine oder mehrere verborgene Schichten besitzen. Neuronen sind untereinander durch Kanten (Links) verbunden. Jede Kante zwischen Neuronen ist mit einem Gewicht versehen. Die Gewichte beschreiben dabei wie stark eine solche Verbindung ist. Je größer der Betrag des Gewichtes  $w$ , desto größer ist der Einfluss eines Neurons auf ein anderes. Das Wissen des neuronalen Netzes ist in seinen Gewichten gespeichert. Das Lernen stellt dabei eine Gewichtsveränderung zwischen den Neuronen dar. Die Effizienz und Flexibilität ist bei keinem anderen Algorithmus so hoch wie hier [78]. KNN ermöglichen es Regelmäßigkeiten und Zusammenhänge zu lernen, um die wichtigen Informationen aus großen Datenmengen herauszufiltern.

### 2.3.1 Deep Learning

Netze mit zwei oder mehr verborgenen Schichten werden als tiefe neuronale Netze bezeichnet [53], man spricht dabei auch von Deep Learning. Mit Hilfe von neuronalen Netzen können Reinforcement Learning Algorithmen auf komplexere Umgebungen angewendet werden. Der am weitesten verbreitete Deep Reinforcement Learning Algorithmus ist das Deep-Q-Learning [28]. Dieser wird in Kapitel 2.4.3 genauer betrachtet.

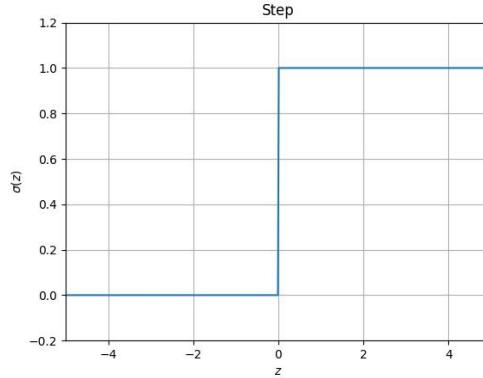


Abbildung 2.3: Schwellwert Aktivierungsfunktion

### 2.3.2 Architekturen von künstlichen neuronalen Netzen

Ein Perzepron stellt die älteste Struktur von neuronalen Netzen dar, anhand dessen sich die Grundfunktionalität eines KNN gut darstellen lässt. KNN unterscheidet man hinsichtlich ihrer Architekturen. Als wichtigste Vertreter gelten Feedforward Netze, Rekurrente neuronale Netze und Convolutional Neural Networks (Faltungsnetze).

#### Perzepron

Das Perzepron wurde von Frank Rosenblatt im Jahr 1958 vorgestellt [68]. Ein Perzepron  $x$  verfügt über  $n$  binäre Eingänge, mit jeweils einer Gewichtung  $w$ , sowie einem binären Ausgang  $o$  [53]. Die erste Verarbeitung der erhaltenen Information des Neurons wird als **Eingangsfunktion**  $z$  bezeichnet und ist mathematisch wie folgt definiert [53]:

$$z = \sum_{i=1}^n (x_i \cdot w_i) + b \quad (2.1)$$

Der Bias (Schwellenwert)  $b$  beschreibt dabei einen konstanten Faktor, mit dessen Hilfe die Aktivierungsschwelle verschoben werden kann. Anschließend wird die Ausgabe  $o$  mit Hilfe der **Aktivierungsfunktion**  $\sigma$  bestimmt [53].

$$o = \sigma(z) \quad (2.2)$$

Es gibt verschiedenste Arten von Aktivierungsfunktionen. Eine einfache Funktion hierbei ist die Schwellwertfunktion  $f_0(z)$ , auch Stufenfunktion genannt [55] (siehe Abbildung 2.3). Basierend auf den erhaltenen Eingaben wird dabei nach folgender Regel

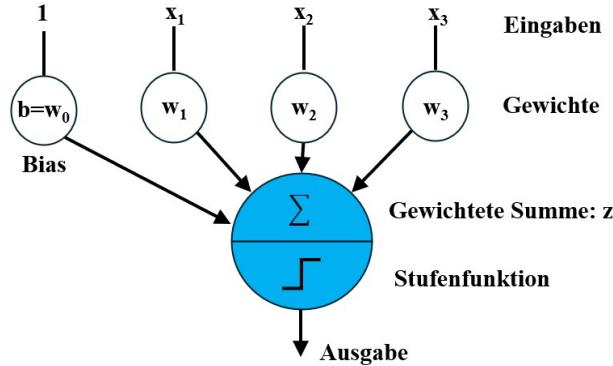


Abbildung 2.4: Aufbau Perzepton

entschieden, ob ein Neuron aktiviert wird ( $f_0(z) = 1$ ) oder nicht ( $f_0(z) = 0$ ) [53]:

$$f_0(z) = \begin{cases} 1, & \text{falls } z > 0, \\ 0, & \text{falls } z \leq 0 \end{cases} \quad (2.3)$$

Ein Perzepton beschreibt damit eine Linear Threshold Unit (lineare Schwellenwerteinheit), die in Abbildung 2.4 dargestellt ist. Diese Schwellenwertfunktion wird jedoch nicht so oft als Aktivierungsfunktion verwendet. Eine kleine Veränderung am Eingang kann zu schlagartigen Änderungen am Ausgang des Neurons führen, wodurch sich das Verhalten des Netzwerkes stark verändern kann. Zudem ist der Gradient (Steigung) null, wodurch kein gradientenbasiertes Lernen (siehe Kapitel 2.3.4) möglich ist. Im Kapitel 2.3.3 werden weitere Aktivierungsfunktionen vorgestellt, die mehr Anwendung finden.

### Feedforward Netze

Netze ohne Rückkopplung heißen Feedforward Netze [26], sie sind die einfachsten KNN. Die Informationen bzw. Signale laufen dabei immer von der Eingabeschicht in Richtung der Ausgabeschicht (siehe Abbildung 2.5 (a)). Typischerweise entspricht es einem Netz, bei dem jedes Neuron einer Schicht mit jedem Neuron der folgenden Schicht verbunden ist. Es gibt auch spezielle Typen von Feedforward Netzen, bei denen das nicht der Fall ist. Ein solches spezielles Netz ist das Faltungsnetz, welches in diesem Unterkapitel noch genauer betrachtet wird.

### Rekurrente neuronale Netze

Netze mit Rückkopplungsschleifen heißen Rekurrente Netze [53] (siehe Abbildung 2.5 (b)). Die Idee ist es, Neuronen zu haben, die für eine begrenzte Zeitdauer immer wieder aktiviert werden, bevor sie ruhig werden. Diese erneute Aktivierung kann andere Neuronen

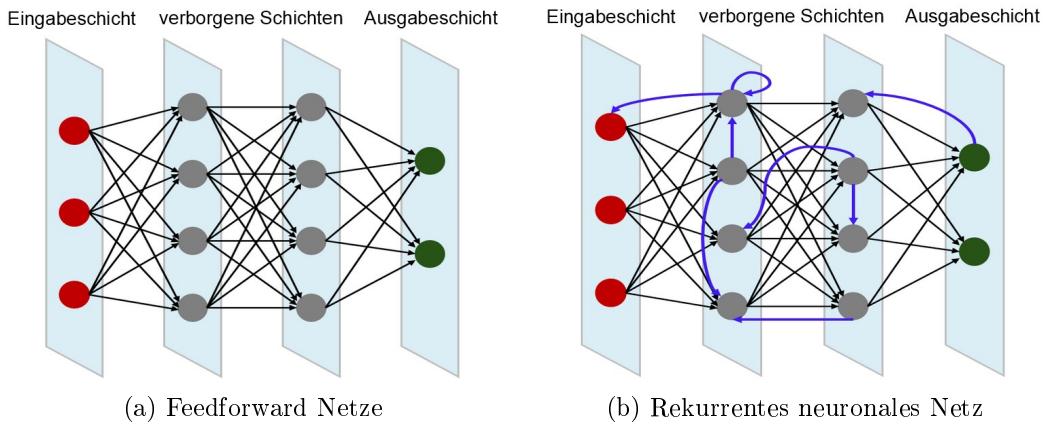


Abbildung 2.5: Netzarchitekturen

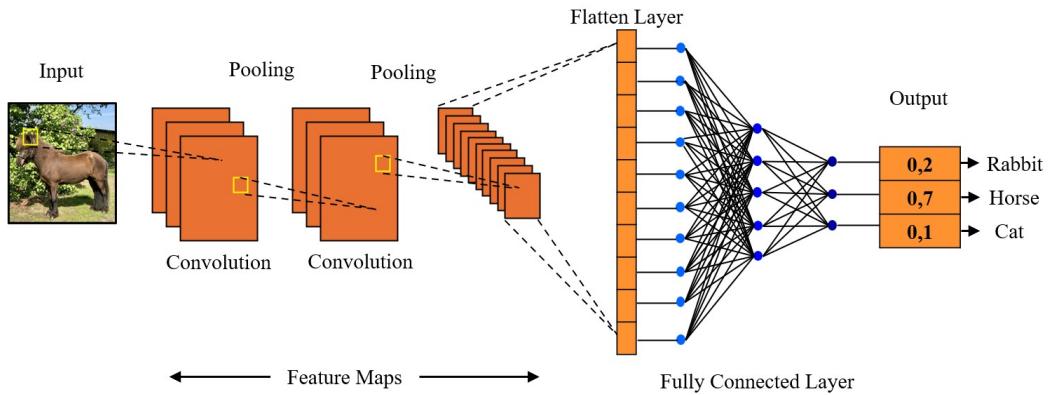


Abbildung 2.6: Faltungsnetz

stimulieren, die etwas später ebenfalls für eine begrenzte Zeitdauer wieder aktiviert werden. Dadurch erhält man im Laufe der Zeit eine Kaskade von Neuronen, die öfters erneut aktiviert werden.

### Convolutional Neural Networks (Faltungsnetze)

Convolutional Neural Networks (CNN) [61, 22], auch Faltungsnetz genannt, werden sehr häufig im Bereich der Videoanalyse oder Bilderkennung verwendet. Das Faltungsnetz ist eine spezieller Typ des Feedforward-Netzes. Ein exemplarischer Aufbau eines Faltungsnetzes ist in Abbildung 2.6 dargestellt. Es werden drei grundlegende Ansätze genutzt: lokale Rezeptivfelder, geteilte Gewichte und Pooling [53]. Wenn ein Bild als Input für das Netz dient, reagieren Neuronen nur auf einen bestimmten Bereich von Bildpixeln, auch lokales Pixelfenster oder **lokales Rezeptivfeld** genannt. Dies bedeutet, dass jedes Neuron in der Faltungsschicht nur mit einem kleinen, lokalen Bereich

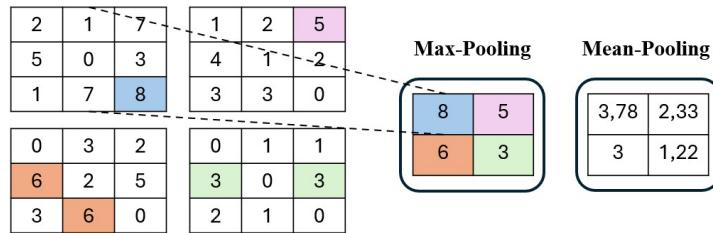


Abbildung 2.7: Pooling

des Eingangsbildes verbunden ist und nicht mit jedem einzelnen Pixel. Dieser lokale Bereich wird durch Filter definiert, die über das Bild gleiten bzw. geschoben (konvolutioniert) werden. Ein Filter, auch Kernel genannt, stellt dabei eine kleine Matrix von Gewichten (z. B. 3x3 oder 5x5) dar. Für die Neuronen einer Schicht bzw. eines Filters werden konstante Gewichte (**geteilte Gewichte**) verwendet. Dies bedeutet, dass dasselbe Filter, unabhängig von seiner Position auf dem Bild, die gleichen Parameter verwendet. Durch die beiden beschriebenen Eigenschaften kann die Anzahl der benötigten Parameter (Gewichte) erheblich reduziert werden und es können einfacher auffällige Merkmale erkannt werden als bei klassischen vollständig verbundenen Feedforward Netzen. Die Faltungsschicht ist das Herzstück des CNN, hier werden die lokalen Rezeptivfelder und die geteilten Gewichte angewendet. Die Ausgabe der Faltungsschicht kann auch als Feature-Map (Merkmalskarte) bezeichnet werden. Das **Pooling** ermöglicht es eine gewisse lokale Invarianz einzubringen, das heißt kleine Änderungen in lokaler Nachbarschaft des Bildbereiches führen nicht zu einem anderen Ergebnis. Durch diese eingeführte Pooling-Schicht, die typischerweise nach der Faltungsschicht folgt, können die in der Faltungsschicht ermittelten Informationen komprimiert werden. Es gibt verschiedenen Arten von Pooling. Zwei Pooling-Arten sind in Abbildung 2.7 dargestellt. Als letzte Schicht folgt eine Flattening-Schicht, die aus den Bildmatrizen einen Vektor für das folgende klassische Feedforward-Netz erzeugt. In dem klassischen Feedforward-Netz findet schlussendlich die Bildklassifizierung statt.

### 2.3.3 Aktivierungsfunktionen

Um komplexere Beziehungen zu erlernen sind nicht lineare Aktivierungsfunktionen ein unverzichtbarer Teil von neuronalen Netzen. Ohne Aktivierungsfunktion bzw. mit linearer Aktivierungsfunktion würde sich ein mehrschichtiges Netz auf nur eine gewichtete

Summe reduzieren [57], die sich als Matrix  $\mathbf{W}$  darstellen lässt.

$$\mathbf{y} = \mathbf{W} \cdot \mathbf{x} \quad (2.4)$$

Dabei beschreibt  $\mathbf{y}$  die Ausgangsschicht (Ausgangsvektor) und  $\mathbf{x}$  die Eingangsschicht (Eingangsvektor). Selbst wenn man mehrere Schichten von Neuronen mit linearen Aktivierungsfunktionen stapelt, wird das Ergebnis letztendlich auf eine einzige lineare Funktion reduziert. Damit könnten nur lineare Beziehungen trainiert werden. Typischerweise wird innerhalb der verborgenen Schichten nur mit einer Art von Aktivierungsfunktion gearbeitet. Innerhalb dieser Schichten findet kein Wechsel statt. Die Aktivierungsfunktion für die Ausgabe sollte aber je nach Aufgabe angepasst werden. Sie kann sich von der Aktivierungsfunktion der verborgenen Schichten unterscheiden. Nachfolgend werden einige häufig verwendete Aktivierungsfunktionen  $\sigma(z)$  vorgestellt. Anders als bei der schon beschriebenen Schwellenwertfunktion wird bei den nachfolgend behandelten Aktivierungsfunktionen das Verhalten des Netzwerkes schrittweise an das gewünschte Verhalten angenähert. Es wird nicht mehr mit rein binären Ein- und Ausgängen gearbeitet, sondern mit reellen Zahlen, wodurch es zu weicheren Aktivierungen des Neurons kommt.

### Sigmoid

Die stetige und differenzierbare Sigmoid-Funktion  $\text{sig}(z)$  [55] ist wie folgt definiert:

$$\text{sig}(z) = \frac{1}{1 + e^{-z}} \quad (2.5)$$

Für die Ableitung ergibt sich:

$$\text{sig}'(z) = \text{sig}(z)(1 - \text{sig}(z)) \quad (2.6)$$

Die Sigmoid-Funktion wird selten für die verdeckten Schichten verwendet, sondern eher in der Ausgabeschicht. Die Funktion ist besonders gut für Anwendungen mit Wahrscheinlichkeitsvorhersagen geeignet, da der mögliche Wertebereich zwischen 0 und 1 liegt (siehe Abbildung 2.8). Ein Beispiel wäre die Bildklassifizierungen zwischen einem Hund und einer Katze. Dabei würden beispielsweise eine Ausgabe von  $\text{sig}(z) < 0.5$  der Klasse Hund und eine Ausgaben  $\text{sig}(z) > 0.5$  der Klasse Katze zugeordnet werden. Die Sigmoid-Funktion kann nur dazu genutzt werden zwei binäre Klassen zu unterscheiden. Sie bringt aber auch einige Probleme mit sich. Wenn die Funktion eine große positive oder negative Zahl erhält, kann es zur Sättigung kommen. Damit findet die Operation

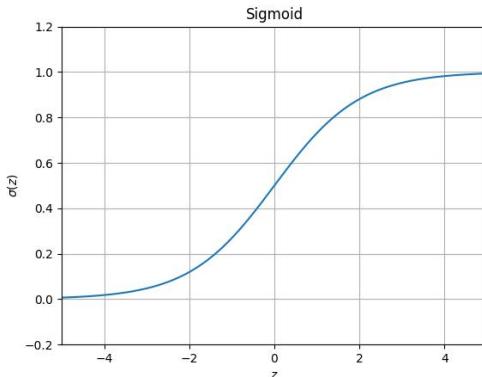


Abbildung 2.8: Sigmoid Aktivierungsfunktion

in einem der beiden flachen, weit vom Mittelpunkt entfernten Ausläufern statt. Dabei strebt der Gradient beim Gradientenverfahren<sup>1</sup> gegen null und verlangsamt das Lernen oder das Lernen ist nicht mehr möglich. Alle darauf folgenden Neuronen werden als **tote Neuronen** [57] bezeichnet, da sie durchs Gradientenverfahren nicht mehr erreicht werden können. Ein weiteres Problem sind **verschwindende Gradienten** [57]. Diese entstehen beim Einsatz von Backpropagation (siehe genaueres in Kapitel 2.3.5) bei tiefen neuronalen Netzen. Es werden lokale Gradienten von der Ausgabe- bis zur Eingabeschicht zusammengesammelt. Bei jeder Schicht wird der Gradient (lokale Steigung der Sigmoid-Funktion maximal 0.25) mit den Gradienten der vorherigen Schichten multipliziert. Auf dem Weg rückwärts durch das Netz kann somit der Gradient verschwindend klein werden, so dass die ersten Schichten nicht mehr viel lernen. Um Netze leistungsfähiger zu machen, werden oft weitere Schichten verwendet. Jedoch bringt es bei der Sigmoid-Funktion ab einem gewissen Zeitpunkt nichts, da die Gradienten der ersten Schichten dann gegen null gehen und diese nicht lernen. Zur Lösung von Problemen der Sigmoid-Funktion wurden weitere Aktivierungsfunktionen entwickelt.

### ReLU (Rectified Linear Unit)

Als Standard-Ersatzfunktion für die Sigmoid-Funktion gilt die gleichgerichtete lineare Einheit, kurz ReLU. Sie ist wie folgt definiert [77]:

$$\text{relu}(z) = \begin{cases} z, & \text{falls } z \geq 0, \\ 0, & \text{falls } z < 0 \end{cases} \quad (2.7)$$

---

<sup>1</sup>Das Gradientenverfahren, welches ein Verfahren zur Optimierung von Modellen durch Minimierung von Fehlerfunktionen ist, wird ausführlicher in Kapitel 2.3.4 erläutert.

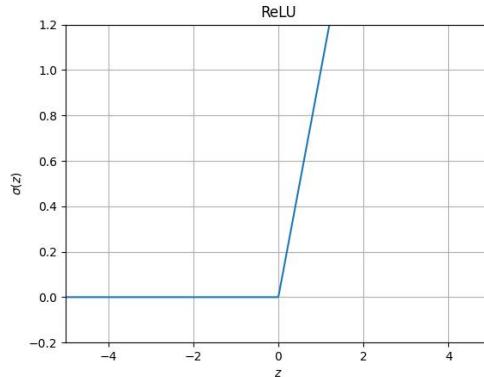


Abbildung 2.9: ReLU Aktivierungsfunktion

Diese Funktion ist eine nichtlineare Funktion, die aus zwei geraden Segmenten besteht (siehe Abbildung 2.9). Da es sich hierbei um eine sehr einfache Funktion handelt, ermöglicht sie ein schnelles Training. Diese Aktivierungsfunktion wird oft für die versteckten Schichten von neuronalen Netzen verwendet. Das Problem von verschwindenden Gradienten kann hierbei für positive Zahlen vermieden werden, da für positive Eingaben  $z > 0$  der Gradient immer 1 ist [57]. Das Problem von toten Neuronen besteht aber weiterhin und es wird sogar verstärkt, da bei negativen Eingaben der Gradient immer 0 ist. Solange es nicht zu viele tote Neuronen werden, kann es sogar von Vorteil sein, da so irrelevante Informationen ignoriert werden, wodurch das Training schneller und effizienter werden kann [57]. Wenn es zu viele tote Neuronen werden, kann das Training aber zum Erliegen kommen. Dann ist die Verwendung von Leaky ReLU geeigneter.

### Leaky ReLU

Im Vergleich zur ReLU-Funktion weist Leaky ReLU (undichte ReLU) im negativen Bereich eine leichte Steigung auf (siehe Abbildung 2.10). Hierdurch wird garantiert, dass der Gradient nie 0 werden kann. Die Neuronen können nicht sterben. Die Leaky ReLU Funktion ist wie folgt definiert [77]:

$$\text{lrelu}(z) = \begin{cases} z, & \text{falls } z \geq 0, \\ a \cdot z, & \text{falls } z < 0 \end{cases} \quad (2.8)$$

Mit dem Parameter  $a$ , kann die Steigung im negativen Bereich eingestellt werden, wobei mit einem Wert von  $a = 0,01$  sehr gute Erfahrungen gemacht wurden [26]. ReLU und leaky ReLU sind beide bei  $z = 0$  nicht differenzierbar. Eine gute Aktivierungsfunktion

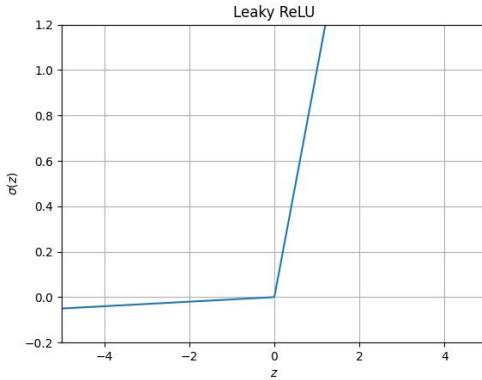


Abbildung 2.10: Leaky ReLU Aktivierungsfunktion

sollte aber glatt sein damit es möglich ist die Gradienten zu berechnen (die Ableitungen zu bilden). In der Praxis ist es sehr unwahrscheinlich, dass die Aktivierungsfunktion eine Eingabe von genau 0 erhält. In den realen Implementierungen wird hierfür im Fall der Eingabe 0 der Gradient auf 1 gesetzt.

### ELU (Exponential Linear Unit)

Die ELU Funktion [16] verbessert auch das Problem der ReLU Funktion bezüglich der sterbenden Neuronen. Es gibt keinen Bereich mit der Steigung 0 bzw. keinen Bereich, in dem der Gradient 0 wird. Die ELU Funktion ist wie folgt definiert:

$$\text{elu}(z) = \begin{cases} z, & \text{falls } z > 0, \\ a(e^z - 1), & \text{falls } z \leq 0 \end{cases} \quad (2.9)$$

mit der Ableitung:

$$\text{elu}'(z) = \begin{cases} 1, & \text{falls } z > 0, \\ \text{elu}(z) + a, & \text{falls } z \leq 0 \end{cases} \quad (2.10)$$

Der Parameter  $a$  definiert den Endwert, an den sich die ELU-Funktion bei sehr großen negativen Werten von  $z$  annähert. Typischerweise ist  $a = 1$  (siehe Abbildung 2.11). Die ELU-Funktion ist an allen Stellen differenzierbar, auch bei  $z = 0$ . Die ELU-Funktion lässt sich wegen der Exponentialfunktion langsamer als ReLU berechnen. Beim Trainieren wird dies aber durch die schnellere Konvergenz im Gradientenverfahren kompensiert. Mit der ELU-Funktion sollen bessere Ergebnisse erzielt werden können als mit der ReLU-Funktion.

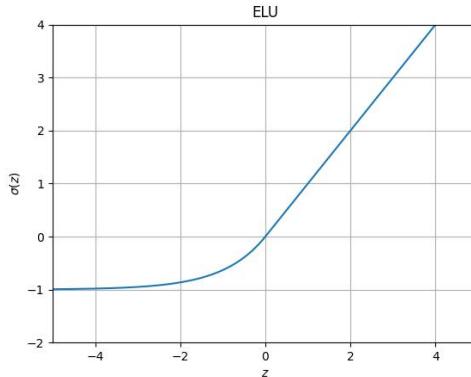


Abbildung 2.11: ELU Aktivierungsfunktion

### Softmax

Die Softmax-Funktion [61], auch normalisierte Exponentialfunktion genannt, wird hauptsächlich bei Multi-Klassifizierungen verwendet. Hierbei wird sie meist als letzte Aktivierungsfunktion in der Ausgabeschicht des neuronalen Netzes eingesetzt, wenn mehr als zwei Klassen zu unterscheiden sind. Sie kann aber auch dafür eingesetzt werden, wenn zwischen mehreren möglichen Aktionen unterschieden werden muss. Als Eingabe erhält die Funktion einen Vektor, wobei  $M$  die Anzahl der möglichen Klassen beschreibt. Die Softmax-Funktion ist für jedes Element  $i$  des Eingabevektors wie folgt definiert:

$$\text{softmax}(z_i) = \frac{e^{z_i}}{\sum_{j=1}^M e^{z_j}} \quad (2.11)$$

Mit Hilfe der Softmax-Funktion kann wie bei der Sigmoid-Funktion ein Zahlenbereich von 0 bis 1 abgebildet werden. Der Verlauf ähnelt dem S-förmigen Verlauf der Sigmoid-Funktion. Als Ergebnis liefert die Funktion keinen einzelnen Index zurück, sondern die Wahrscheinlichkeiten für die verschiedenen Klassen. Die Summe der Ausgabe ist stets 1. Die Softmax-Funktion ist allerdings auch anfällig gegen eine Unausgewogenheit von positiven und negativen Eingabedaten. Zudem ist sie empfindlich gegenüber großen negativen oder positiven Eingaben. In beiden Fällen kann es dazu kommen, dass keine sinnvollen Wahrscheinlichkeiten zugewiesen werden können.

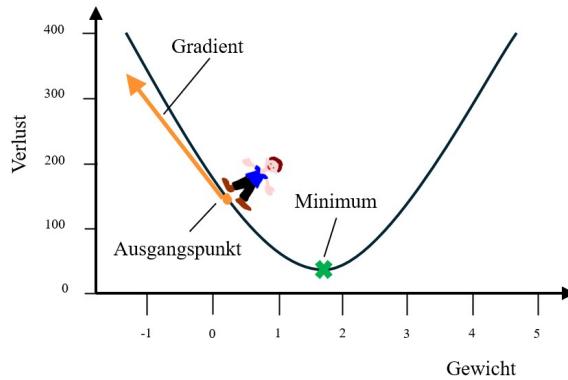


Abbildung 2.12: Gradientenverfahren am Beispiel einer Bergsteigers

#### 2.3.4 Gradientenverfahren

Das Training eines neuronalen Netzes kann mit Hilfe des Gradientenverfahrens durchgeführt werden. Bei einem Gradientenverfahren wird eine **Kostenfunktion** (auch Verlustfunktion genannt) [57] benötigt, um ein Maß für die Abweichung zwischen vorhergesagtem Wert eines Modells und dem erwarteten Wert zu bestimmen. Spezifische Kostenfunktionen werden in Kapitel 2.3.6 genauer vorgestellt. Das Ziel des Trainings besteht darin, die Werte der Gewichtung  $w$  und dem Bias  $b$  zu finden, sodass die Kostenfunktion  $C(w, b)$  minimiert wird und am Ende des Trainings das tatsächliche Ergebnis mit dem geschätzten Ergebnis übereinstimmt. Für diese Minimierung der Kostenfunktion kann das **Gradientenverfahren** [57] verwendet werden. Das Prinzip des Gradientenverfahrens kann gut an dem Beispiel einer Wanderin erklärt werden. Die Wanderin befindet sich auf einem Berg und sucht bei dichtem Nebel ihr Lager, welches sich im Tal befindet. Um dieses Lager zu finden, kann sie einfach immer den Weg mit dem steilsten Abstieg nehmen. Deshalb wird das Gradientenverfahren auch Verfahren des steilsten Abstiegs genannt. Somit führt sie jeder Schritt ein Stück näher an das Lager. Die Berg-Tal-Umgebung stellt dabei die Kostenfunktion als Verlustkurve dar (siehe Abbildung 2.12). Dabei wurde zuerst vereinfacht nur das Gewicht  $w$  als Variable berücksichtigt, deshalb ergibt sich eine zweidimensionale Darstellung. Das Ziel besteht darin das Minimum zu finden. Um den Weg des steilsten Abstiegs zu finden, muss die Steigung (Ableitung) bestimmt werden. Der Gradient ist dafür ein Maß. Um das Minimum zu erreichen, muss die Wanderin in entgegengesetzte Richtung zum Gradienten gehen, da dieser immer in Richtung des stärksten Anstiegs zeigt.

### Mathematische Beschreibung

Für die Bestimmung des Gradienten muss die partielle Ableitung der Kostenfunktion  $C(w, b)$  durchgeführt werden. Nachfolgend wird das Gradientenverfahren vereinfacht am Beispiel von zwei Parametern  $w$  und  $b$  erläutert. In der Praxis entspricht  $b$  meist einem mehrdimensionalen Vektoren und  $w$  einer Matrix (siehe Formel 2.21). Zusammengesetzt entspricht dies dem Gradienten  $\nabla C$  [53]:

$$\nabla C = \begin{pmatrix} \frac{\partial C(w, b)}{\partial w} \\ \frac{\partial C(w, b)}{\partial b} \end{pmatrix} \quad (2.12)$$

Der Gradient der Kostenfunktion beschreibt einen Vektor, der die Richtung und Steigung angibt, in der die Kostenfunktion am steilsten ansteigt. Anhand dieses Gradienten können die Parameter  $w$  und  $b$  wie folgt aktualisiert werden [57, 53]:

$$b \rightarrow b' = b - \alpha \cdot \frac{\partial C(w, b)}{\partial b} = b + \Delta b \quad (2.13)$$

mit

$$\Delta b = -\alpha \cdot \frac{\partial C(w, b)}{\partial b} \quad (2.14)$$

bzw.

$$w \rightarrow w' = w - \alpha \cdot \frac{\partial C(w, b)}{\partial w} = w + \Delta w \quad (2.15)$$

mit

$$\Delta w = -\alpha \cdot \frac{\partial C(w, b)}{\partial w} \quad (2.16)$$

Die Lernrate  $\alpha$  entspricht einem kleinen positiven Wert (bspw. 0,001 [57]). Sie hat Einfluss auf die Schrittgröße der Bewegung und somit auf die Geschwindigkeit des Abstiegs [53]. Die Wahl einer geeigneten Lernrate hat einen entscheidenden Einfluss auf den Lernprozess. Ist die Lernrate zu hoch, kann es zu Oszillationen kommen (siehe Abbildung 2.13) (a)), während eine zu kleine Lernrate den Lernprozess verlangsamt. Das Gradientenverfahren wird iterativ angewendet, indem die Parameter wiederholt aktualisiert werden, bis eine Konvergenz zum Minimum erreicht ist. Dies ist erfolgt, wenn die Änderungen der Kostenfunktion oder der Gradientenwerte sehr klein ist.

### Lokale Minima

Es können Probleme beim Gradientenverfahren auftreten, beispielsweise dass ein lokales Minimum gefunden wird. Das globale Minimum kann dann nicht mehr erreicht werden (siehe Abbildung 2.13 (b)). Um mit diesen lokalen Minima umgehen zu können und die Konvergenz zum globalen Minimum zu beschleunigen, kann ein **Momentum**

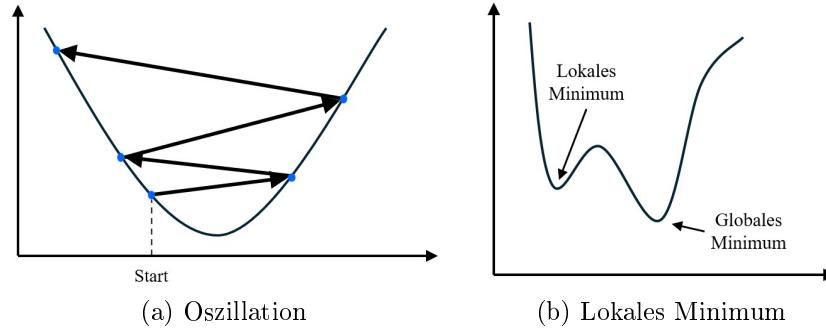


Abbildung 2.13: Herausforderungen beim Gradientenverfahren

[26] verwendet werden. Die Idee dahinter besteht darin, dass dem aktuellen Schritt eine Komponente hinzugefügt wird, die von der vorherigen Bewegung abhängig ist. Dies hilft dabei, die Bewegung über flache Regionen und kleine lokale Minima hinaus zu beschleunigen. Nachfolgend ist die Aktualisierung der Parameter am Beispiel des Gewichtes  $w$  bei der Erweiterung um das Momentum  $m$  gezeigt:

$$w \rightarrow w' = w - \alpha \cdot \frac{\partial C(w, b)}{\partial w} + \beta \cdot m \quad (2.17)$$

Der Hyperparameter  $\beta$  liegt dabei in einem Wertebereich von 0 (normales Gradientenverfahren) und 1 (reibungsfreie Bewegung, die eine Beschleunigung bewirkt). In der Praxis hat sich für  $\beta$  ein Wert von 0,9 sehr gut bewährt [26].

### Mini-Batch Gradientenverfahren

Als weitere Herausforderung kann genannt werden, dass die Berechnung des zuvor erwähnten Gradienten bei großen Datensätzen sehr zeitaufwändig werden kann [53]. Der Gradient muss theoretisch für alle Trainingsdatenpunkte bestimmt werden. Um dieses Problem zu beheben, kann der Mini-Batch Gradientenabstieg [53] verwendet werden. Dadurch kann das Lernen beschleunigt werden. Hierbei wird eine kleine Teilmenge  $m$  ((Mini-)Batch) an zufällig ausgewählten Trainingsdatenpunkten  $x$  berechnet, um den durchschnittlichen Gradienten zu schätzen. Somit muss nur eine kleine Menge an Gradienten  $\nabla C_x$  berechnet werden, um eine Näherung des Gesamtgradienten  $\nabla C$  zu erhalten:

$$\nabla C \approx \frac{1}{m} \sum_{j=1}^m \nabla C_{x_j} \quad (2.18)$$

### 2.3.5 Backpropagation (Fehlerrückführung)

Des Weiteren ist zu erwähnen, dass es bei modernen neuronalen Netzen aufgrund ihrer Schichten sehr schwierig ist die komplette Kostenfunktion zu formulieren, geschweige noch die Ableitung zu bestimmen [57]. Deshalb wird die manuelle Berechnung der Gradienten für jede Schicht extrem kompliziert und zeitaufwändig. Aus diesem Grund verwendet man Backpropagation. Backpropagation ermöglicht die effiziente Berechnung der Gradienten der Kostenfunktion bezüglich aller Gewichte und Bias im Netz. Dies geschieht in einem Vorwärts- und einem Rückwärtsdurchlauf durch das Netz [26]. Im Vorwärtsdurchlauf werden die Vorhersagen des Netzwerks berechnet, und im Rückwärtsdurchlauf werden die Fehler (Differenzen zwischen den Vorhersagen und den tatsächlichen Werten) zurück durch das Netzwerk propagiert, um die Gradienten zu berechnen.

#### Vorwärtsdurchlauf

Nachfolgend entspricht  $\sigma$  der genutzten Aktivierungsfunktion. Die zuvor vorgestellten Aktivierungsfunktionen, mit der Ausnahme der Schwellenfunktion, sind für Backpropagation verwendbar. Die Schwellenwertfunktion enthält nur horizontale Abschnitte mit der Steigung null, daher gibt es hier keine Gradienten und das Gradientenverfahren kann nicht angewendet werden. Die Aktivierung bzw. der Ausgang  $a$  des Neurons  $j$  in der Schicht  $l$  kann wie folgt bestimmt werden [53]:

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) = \sigma(z_j^l) \quad (2.19)$$

$k$  entspricht der Anzahl an Neuronen der vorherigen Schicht  $a_k^{l-1}$  bzw. der Anzahl der Eingänge des Neurons. Die Aktivierung der gesamten Schicht  $l$  kann in vektorieller Form wie folgt dargestellt werden

$$\mathbf{a}^l = \sigma(\mathbf{W}^l \mathbf{a}^{l-1} + \mathbf{b}^l) \quad (2.20)$$

mit der Gewichtsmatrix

$$\mathbf{W}^l = \begin{pmatrix} w_{11} & w_{12} & \dots & w_{1k} \\ w_{21} & w_{22} & \dots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1} & w_{j2} & \dots & w_{jk} \end{pmatrix}. \quad (2.21)$$

#### Rückwärtsdurchlauf

Um den Einfluss einer bestimmten Gewichtsanpassung auf den bekannten Gesamtfehler

am Ausgang zu ermitteln muss rückwärts durchs Netz durchgegangen werden. Die Berechnung der Gradienten der aktuellen Schicht kann nur erfolgen, wenn die Gradienten der nachfolgenden Schicht bekannt sind. Für die effiziente Berechnung geht man deshalb rückwärts durch die Schichten. Im Rückwärtsdurchlauf wird der Fehler (Differenz zwischen vorhergesagter und tatsächlicher Ausgabe) von der Ausgabeschicht zurück zur Eingabeschicht berechnet, um die Gradienten der Kostenfunktion zu berechnen. Zur Anwendung dieses Verfahrens wird eine differenzierbare Aktivierungsfunktion benötigt, wie aus den nachfolgend vorgestellten Gleichungen ersichtlich wird.

### 1. Fehler in der Ausgabeschicht

Der Fehler  $\delta_j^l$  des Neurons  $j$  in der Ausgabeschicht  $l$  kann mit der Kostenfunktion

$$C = \frac{1}{2}(a_j^l - y_j)^2 \quad (2.22)$$

wie folgt berechnet werden [53]:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l} = (a_j^l - y_j) \cdot \sigma'(z_j^l) \quad (2.23)$$

Vom geschätzte Wert  $a_j^l$  (Aktivierungswert bzw. Ausgangswert des Ausgabeneurons, der auch  $\sigma(z_j^l)$  entspricht) wird der tatsächliche Wert  $y_j$  abgezogen und mit der Ableitung der Aktivierungsfunktion des Ausgabeneurons multipliziert. An der Formel 2.23 lässt sich erkennen, dass die Kettenregel ( $f(g(x))' = f'(g(x)) \cdot g'(x)$ ) beim Backpropagation Verfahren angewendet wird.

### 2. Fehler in den versteckten Schichten

Anschließend wird der Fehler wie folgt rückwärts durch das gesamte Netz geführt [53]:

$$\delta_j^l = \left( \sum_k w_{kj}^{l+1} \delta_k^{l+1} \right) \cdot \sigma'(z_j^l) \quad (2.24)$$

Dabei entspricht  $w_{kj}^{l+1}$  dem Gewicht der Verbindung zwischen Neuron  $j$  in Schicht  $l$  und Neuron  $k$  in Schicht  $l + 1$ .  $\delta_k^{l+1}$  beschreibt den Fehler des Neurons  $k$  der Schicht  $l + 1$ .

### 3. Gradienten der Kostenfunktion

Die Gradienten der Kostenfunktion bezogen auf die Gewichte und Biase können dann

durch folgende Formeln ermittelt werden [53]:

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \quad (2.25)$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \quad (2.26)$$

Die Gewichte und Biase können mit Hilfe der bestimmten Gradienten, wieder wie in den Formeln 2.13 und 2.15 dargestellt, aktualisiert werden.

### 2.3.6 Verlustfunktionen

Die Verlustfunktion (loss function) wird auch als Kosten- oder Fehlerfunktion bezeichnet. Kostenfunktionen sind eine entscheidende Komponente für die Optimierung und das Training von Modellen. Mit ihrer Hilfe kann der Unterschied bzw. Fehler zwischen dem vorhergesagten Ausgabewert des Modells und dem erwarteten Ausgabewert quantifiziert werden. Dadurch kann die Modellqualität bewertet werden. Je kleiner der Fehler zwischen vorhergesagtem und erwarteten Wert ist, desto besser ist das Modell. Beim Training geht es deshalb darum die Kostenfunktion zu minimieren. Nachfolgend werden einige Kostenfunktionen vorgestellt.

#### MSE (Mean Square Error)

Die mittlere quadratische Abweichung [61] misst die durchschnittliche quadratische Differenz zwischen dem tatsächlichen Wert  $y$  und dem von einem Modell vorhergesagten Wert  $\hat{y}$ .

$$MSE = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2 \quad (2.27)$$

$n$  beschreibt dabei die Anzahl der Beobachtungen bzw. Datenpunkte. Der Mittlere quadratische Fehler wird häufig bei Regressionsproblemen eingesetzt. Bei der Regression besteht das Ziel darin, die Differenz zwischen vorhergesagten und tatsächlichen Werten zu minimieren. Auf Grund der Quadrierung werden größere Fehler stärker gewichtet und haben somit mehr Einfluss auf die Optimierung. Das erhöht die Empfindlichkeit bei Ausreißern (hohen positiven oder negativen Eingaben).

#### MAE (Mean Absolut Error)

Die mittlere absolute Abweichung [26] misst die durchschnittliche absolute Differenz

zwischen dem tatsächlichen Wert  $y$  und dem vom Modell vorhergesagten Werten  $\hat{y}$ .

$$MAE = \frac{1}{n} \sum_{i=1}^n |y_i - \hat{y}_i| \quad (2.28)$$

$n$  beschreibt dabei wieder die Anzahl der Datenpunkte. MAE verwendet die absoluten Differenzen, wodurch jeder Fehler gleich gewichtet wird, was MAE im Vergleich zu MSE weniger empfindlich gegenüber Ausreißern macht. Die Funktion wird auch oft bei Regressionsproblemen eingesetzt.

### Huber Loss

Huber Loss [47] ist eine robuste Verlustfunktion, die MSE und MAE kombiniert. Sie wird somit auch bei Regressionsproblemen eingesetzt. Diese Funktion bleibt gegenüber Ausreißern robust sowie bei kleinen Fehlern effizient. Sie ist für kleine Werte quadratisch (MSE) und für große Werte linear (ähnelt MAE):

$$L_\delta(y - \hat{y}) = \begin{cases} \frac{1}{2} \cdot (y - \hat{y})^2 & \text{für } |y - \hat{y}| \leq \delta \\ \delta(|y - \hat{y}| - \frac{1}{2}\delta) & \text{für } |y - \hat{y}| > \delta \end{cases} \quad (2.29)$$

$\delta$  definiert dabei den Schwellenwert, ab dem die Verlustfunktion zwischen einem quadratischen und einem linearen Verhalten wechselt.

### Cross-Entropy

Kreuzentropie [61] ist eine der am häufigsten verwendeten Verlustfunktionen bei Klassifikationsproblemen. Meist wird damit gemessen, wie gut die vorhergesagte Wahrscheinlichkeit der Kategorie (Klasse) mit der tatsächlichen Zielkategorie übereinstimmt. Typischerweise werden zwei Arten der Kreuzentropie verwendet, die binäre und die kategoriale Kreuzentropie. Für die Unterscheidung von zwei Klassen wird die binäre Kreuzentropie (Binary Cross Entropy) eingesetzt:

$$CE_{bin} = -\frac{1}{n} \sum_{i=1}^n [y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)] \quad (2.30)$$

Hierbei bezeichnet  $n$  die Anzahl der Beobachtungen bzw. Datenpunkte,  $y_i$  den tatsächlichen Klassenwert (entweder 0 oder 1) und  $\hat{y}_i$  ist die vorhergesagte Wahrscheinlichkeit für die Klasse. Bei einer Mehrklassenklassifizierung kommt die kategoriale Kreuzentropie (Categorical Cross Entropy) zum Einsatz [26], wobei nachfolgend  $m$  der Anzahl der

Klassen entspricht:

$$CE_{cat} = -\frac{1}{n} \sum_{i=1}^n \sum_{j=1}^m y_{ij} \cdot \log(\hat{y}_{ij}) \quad (2.31)$$

### 2.3.7 Optimierer

Um das Training durchführen zu können, muss eine Optimierungsfunktion definiert werden, welche die Gewichte und Bias in Richtung des Gradientenabstiegs verschiebt. Somit steuert der Optimierer, wie diese Parameter bei jedem Durchlauf angepasst werden sollen, um die Verlustfunktion zu minimieren. Nachfolgend werden verbreitete Optimierer vorgestellt. Die genauen Aktualisierungsregeln der nachfolgend vorgestellten Optimierer können in den entsprechend zitierten Ursprungspaper oder als Zusammenfassung im Paper [69] nachvollzogen werden.

#### **SGD** (Stochastic Gradient Descent)

Der stochastische Gradientenabstieg ist eine Variante des in Kapitel 2.3.4 vorgestellten traditionellen Gradientenverfahrens (Batch Gradientenverfahren). SGD beruht auf einer von Robbins und Monro eingeführten stochastischen Approximationsmethode [67, 69]. Rosenblatt wendet SGD zum ersten mal im Jahr 1958 auf ein neuronales Netz an [68, 69]. Anstatt den Gradienten der Fehlerfunktion über den gesamten Datensatz gleichzeitig je Epoche zu berechnen, wie es beim Batch Gradientenverfahren gemacht wird, verwendet SGD nur einen einzelnen Trainingsdatensatz zur Zeit. Dies Berechnung und Aktualisierung wird bei jedem Trainingsschritt durchgeführt. Es wird eine feste Lernrate für alle Parameter verwendet. Der Vorteil besteht in der Einfachheit und dem geringen Rechenaufwand pro Aktualisierung. Es können jedoch Schwierigkeiten mit lokalen Minima auftreten. Zudem führt SGD häufige Aktualisierungen mit hoher Varianz durch, die dazu führen, dass die Zielfunktion stark schwankt. Zur Verbesserung kann es durch ein Momentum erweitert oder das Mini-Batch Gradientenverfahren eingesetzt werden. Diese beiden Erweiterungen bzw. Varianten wurde bereits im Kapitel 2.3.4 beschrieben.

#### **AdaGrad**

Der adaptive Gradienten Algorithmus, AdaGrad [20], wurde im Jahr 2011 von John Duchi, Elad Hazan und Yoram Singer vorgestellt. Er passt die Lernrate während des Trainings dynamisch an, dabei wird die Lernrate für jeden Parameter individuell skaliert. Diese Lernratenanpassung erfolgt basierend auf der Summe der quadrierten Gradienten, die bis zum aktuellen Zeitpunkt berechnet wurden. Dies führt zu einer effektiven

und effizienten Optimierung, was den Umgang mit spärlichen Daten oder großen Datensätzen erleichtert und manuelles Tuning der Lernrate überflüssig macht. Ein großer Nachteil des Optimierers ist, dass dieser die Lernrate immer weiter verringert, bis diese so klein ist, dass keine Aktualisierungen mehr stattfinden und somit kein weiterer Lernprozess mehr möglich ist. Die folgenden Algorithmen zielen darauf ab, dieses Problem zu beheben.

### **AdaDelta**

Der adaptive Delta Algorithmus, AdaDelta [80] vorgestellt im Jahr 2012 von Matthew D. Zeiler, ist eine Erweiterung des AdaGrad-Algorithmuses. AdaDelta passt nicht nur die Lernraten dynamisch während des Lernens an, sondern verhindert auch, dass sie im Laufe der Zeit zu klein wird. Anstatt alle quadrierten Gradienten der Vergangenheit zu nutzen, beschränkt AdaDelta sich auf ein gleitendes Fenster bzw. eine bestimmte Anzahl der letzten Gradienten der Vergangenheit. Diese Art der Anpassung sorgt dafür, dass die Lernrate im Laufe der Zeit nicht zu stark abnimmt. Es muss keine Standardlernrate festgelegt werden, da diese aus der Aktualisierungsformel eliminiert wurde.

### **RMSProp** (Root Mean Square Propagation)

RMSProp wurde ursprünglich von Geoffrey Hinton im Rahmen einer Vorlesung vorgestellt [69, 34]. Dieser Algorithmus ist ebenfalls eine Erweiterung des AdaGrad-Algorithmus und wurde etwa zur gleichen Zeit wie AdaDelta entwickelt um zu verhindern, dass die Lernrate im Laufe der Zeit zu klein wird. Es muss eine Standardlernrate angegeben werden, die aber durch Berücksichtigung des Durchschnitts der jüngsten quadrierten Gradienten skaliert wird, sodass sie für jeden Parameter individuell ist. Bei RMSProp werden ausschließlich vergangene Gradienten verwendet, wobei AdaDelta auch vergangene Aktualisierungsschritte berücksichtigt.

### **Adam** (Adaptive Moment Estimation)

Die adaptive Momentschätzung, der Adam-Algorithmus [41], wurde von Kingma und Ba im Jahre 2014 in einem Forschungspapier vorgestellt. Er kombiniert zwei Prinzipien, den RMSprop-Algorithmus mit der dynamischen Lernrate und den SGD mit Momentum. Es werden dabei zwei Momenta eingesetzt. Das erste Momentum hilft dabei, die Richtung des nächsten Schrittes im Parameterbereich zu bestimmen. Das zweite Momentum dient dazu die Schrittgröße anzupassen. Adam hat sich im Vergleich zu anderen Optimierern als sehr effektiv erwiesen, vor allem wenn es um komplexe Netzstrukturen oder große Datensätze geht. Die dynamische Lernrate ermöglicht eine effiziente Konvergenz der Verlustfunktion zu einem Minimum, was zu einem schnellen und stabilen Training

führt. Durch die Momentschätzung werden Probleme mit verschwindenden oder explodierenden Gradienten abgemildert, welche eine häufige Herausforderung beim Training von neuronalen Netzen darstellen. Dieser Algorithmus ist der weit verbreitetste Optimierer bei KI-Anwendungen [57].

### 2.3.8 Gewichtsinitialisierung

Wenn ein neuronales Netzwerk trainiert werden soll, muss es zuerst initialisiert werden. Das bedeutet, jedes Neuron im Netz muss einen Startwert für seine Gewichte bekommen. Eine angemessene Gewichtsinitialisierung wird benötigt um überhaupt komplexe Muster in den Eingabedaten zu erkennen und dementsprechend eine differenzierte Handlungsscheidung zu treffen. Außerdem kann das Risiko für verschwindende oder explodierende Gradienten verringert werden. Eine gute Initialisierung trägt des Weiteren dazu bei, dass das neuronale Netz effizienter und stabiler trainiert werden kann.

#### Nullinitialisierung

Die Nullinitialisierung setzt alle Gewichte auf Null. Dadurch dass alle Gewichte gleich initialisiert wurden, wird jedes Neuron der verdeckten Schichten dieselbe Ausgabe für eine Eingabe erzeugen. Beim Backpropagation erhalten dann alle Neuronen in der verdeckten Schicht die gleichen Gradienten und werden auf gleiche Weise aktualisiert. Dieses Symmetrieproblem [30] führt dazu, dass das Netz nicht in der Lage ist, komplexe Muster in den Daten zu erkennen oder verschiedene Merkmale zu unterscheiden. Somit können keine nützlichen Vorhersagen für zu wählende Aktionen getroffen werden. Um dieses Problem zu vermeiden, müssen die Gewichte unterschiedlich initialisiert werden. Dadurch wird erreicht, dass die Neuronen in jeder Schicht unterschiedlich gewichtete Eingaben und damit unterschiedliche Ausgaben haben. So entstehen unterschiedliche Gradienten und es kommt zu einer differenzierten Gewichtsanpassung. Die unterschiedliche Initialisierung ermöglicht es den Neuronen, unterschiedliche Funktionen bzw. Merkmale zu lernen und so die Lernkapazität des Netzwerks auszuschöpfen. Nachfolgend werden Methoden vorgestellt, wie die Neuronen unterschiedlich initialisiert werden können.

#### Zufällige Initialisierung

Bei der zufälligen Initialisierung werden die Gewichte zufällig aus einer bestimmten Verteilung (standardmäßig: normal oder gleichverteilt) initialisiert. Bei einer zufälligen Normalverteilung [21] werden die Gewichte zufällig aus einer Normalverteilung (auch

Gauß-Verteilung genannt) mit einem Mittelwert  $\mu = 0$  und einer Standardabweichung  $\sigma = 1$  gezogen. Die Standardabweichung ist ein Maß für die durchschnittliche Abweichung aller erhobenen Werte vom Mittelwert. Die Normalverteilung wird wie folgt beschrieben:

$$f(x) = \frac{1}{\sigma\sqrt{2\cdot\pi}} \cdot e^{-\frac{1}{2}\left(\frac{x-\mu}{\sigma}\right)^2} \quad (2.32)$$

Durch Einsetzen der Standardabweichung und des Mittelwertes ergibt sich:

$$f(x) = \frac{1}{\sqrt{2\cdot\pi}} \cdot e^{-\frac{x^2}{2}} \quad (2.33)$$

Diese Verteilung ist durch ihre charakteristische Glockenkurve bekannt. Dabei befinden sich die meisten Werte nahe dem Mittelwert und die Wahrscheinlichkeit für extremere Werte nimmt nach außen hin ab. Hingegen werden bei einer Gleichverteilung [21] die Gewichte zufällig aus einem Intervall  $[a, b]$  gezogen, wobei alle Werte innerhalb dieses Intervalls die gleiche Wahrscheinlichkeit haben, ausgewählt zu werden:

$$f(x) = \begin{cases} \frac{1}{b-a} & \text{für } a \leq x \leq b \\ 0 & \text{sonst} \end{cases} \quad (2.34)$$

Die Gleichverteilung neigt im Vergleich zur Normalverteilung eher zu extremeren Werten (explodierenden Gradienten). Wenn das Intervall hingegen zu klein gewählt wird, kann das Symmetrieproblem wie bei der Nullinitialisierung auftreten. Die zufällige Initialisierung kann bei tiefen neuronalen Netzen zu Problemen mit verschwindenden und explodierenden Gradienten führen.

### Xavier-Initialisierung

Die Xavier-Initialisierung, auch bekannt als Glorot-Initialisierung, wurde im Jahr 2010 von Xavier Glorot und Yoshua Bengio vorgestellt [27]. Die Methode wurde entwickelt, um das Problem der verschwindenden und explodierenden Gradienten bei tiefen neuronalen Netzen zu verringern. Außerdem erhöht sich die Stabilität und Effektivität im Trainingsprozess. Bei dieser Methode wird die Varianz  $\sigma^2$  der Gradienten gleichmäßig auf die Schichten verteilt, um so einer Schicht nicht zu viel Bedeutung zu geben und andere zu vernachlässigen. Die Gewichte werden basierend auf der Anzahl der Eingangsneuronen  $n_{in}$  und Ausgangsneuronen  $n_{out}$  einer Schicht initialisiert. Auch hier kann wieder zwischen einer Normal- und einer Gleichverteilung unterschieden werden.

Bei der Normalverteilung ist der Mittelwert  $\mu = 0$  und die Varianz wird durch

$$\sigma^2 = \frac{2}{n_{in} + n_{out}} \quad (2.35)$$

beschrieben. Anschließend kann die Normalverteilung nach Formel 2.32 berechnet werden. Bei der Gleichverteilung werden die Gewichte aus dem Intervall  $[-a, a]$  gezogen, wobei  $a$  wie folgt berechnet wird [26]:

$$a = \sqrt{\frac{6}{n_{in} + n_{out}}} \quad (2.36)$$

Die Xavier-Initialisierung funktioniert beispielsweise sehr gut für Aktivierungsfunktionen wie Sigmoid, ist aber für die ReLU-Aktivierungsfunktion und seine Varianten nicht so gut geeignet [26].

### He-Initialisierung

Die He-Initialisierung, auch Kaiming Initialisierung genannt, wurde im Jahr 2015 vorgestellt [31]. Sie ist eine Erweiterung der Xavier-Initialisierung, welche speziell für die ReLU-Aktivierungsfunktion und seine Varianten entwickelt wurde. Auch die He-Initialisierung zielt darauf ab, die Probleme der verschwindenden und explodierenden Gradienten zu reduzieren. Wie bei der Xavier-Initialisierung wird die Varianz der Gradienten gleichmäßig auf alle Schichten verteilt. Es gibt wieder zwei Implementierungsmöglichkeiten, die Normal- und die Gleichverteilung. Bei der Normalverteilung ist der Mittelwert  $\mu = 0$  und die Varianz wird durch

$$\sigma^2 = \frac{2}{n_{in}} \quad (2.37)$$

beschrieben. Hier wird lediglich die Anzahl der Eingangsneuronen  $n_{in}$  berücksichtigt. Bei der Gleichverteilung werden die Gewichte aus dem Intervall  $[-a, a]$  gezogen, wobei  $a$  wie folgt berechnet wird [26]:

$$a = \sqrt{\frac{6}{n_{in}}} \quad (2.38)$$

Die He-Initialisierung ist beispielsweise bei der Sigmoid Aktivierungsfunktion nicht so gut geeignet.

### 2.3.9 Batch-Normalisierung

Die He- oder Xavier-Initialisierung kann das Problem der schwindenden oder explodierenden Gradienten zu Beginn des Trainierens deutlich reduzieren, sie sind aber keine Garantie dafür, dass die beiden Probleme nicht später zurückkehren. Aus diesem Grund wurde die Batch-Normalisierung im Jahr 2015 von Sergey Ioffe und Christian Szegedy vorgestellt [36]. Batch-Normalisierung normalisiert die Aktivierungen jedes Mini-Batches auf eine Normalverteilung mit einem Mittelwert  $\mu = 0$  und einer Standardabweichung  $\sigma = 1$ . Dies wird durch die folgenden drei Schritte erreicht [26].

**1. Mittelwert und Varianz:**  $\mu_B$  beschreibt den Mittelwert der Aktivierungen  $x_i$  des aktuellen Mini-Batch der Größe  $m_B$  und  $\sigma_B^2$  beschreibt dabei die Varianz der Aktivierungen im aktuellen Mini-Batch.

$$\mu_B = \frac{1}{m_B} \sum_{i=1}^{m_B} x_i \quad (2.39)$$

$$\sigma_B^2 = \frac{1}{m_B} \sum_{i=1}^{m_B} (x_i - \mu_B)^2 \quad (2.40)$$

**2. Normalisierung der Aktivierungen:**  $\hat{x}_i$  beschreibt eine auf null zentrierte und normalisierte Eingabe. Zudem wird ein kleine Zahl  $\epsilon$  (typischerweise  $10^{-5}$ ) verwendet um eine Division durch Null zu verhindern.

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}} \quad (2.41)$$

Der Mittelwert verschiebt die Verteilung der Aktivierungen so, dass der neue Mittelwert der normalisierten Werte bei 0 liegt. Die Varianz skaliert dabei die Verteilung der Aktivierungen so, dass die neue Standardabweichung der normalisierten Werte 1 ist.

**3. Skalierung und Verschiebung:**  $z_i$  entspricht der Ausgabe und stellt damit die skalierte und verschobene Eingabe dar.

$$z_i = \gamma \cdot \hat{x}_i + \beta \quad (2.42)$$

Der Parameter  $\beta$  dient zum Verschieben, dieser Parameter wird während des Trainings gelernt bzw. durch Backpropagation optimiert/angepasst, ähnlich wie die Gewichte. Er ermöglicht es, den Mittelwert der normalisierten Daten zu verschieben (Offset), um

Flexibilität zu bieten und die Modellleistung zu verbessern.  $\gamma$  dient zum Skalieren der Schicht. Er wird auch während des Trainings erlernt und kann die Normalisierung rückgängig machen oder diese anpassen, um die Repräsentationsfähigkeit zu verbessern. Durch das Hinzufügen der Parameter  $\gamma$  und  $\beta$  kann das Netzwerk somit flexibel lernen wie stark die Normalisierung angewendet werden soll. Begonnen wird oft mit einem  $\beta = 0$  und  $\gamma = 1$ , sodass die normalisierten Aktivierungen nicht verändert sind. Diese Berechnungsschritte werden für jede Schicht in einem neuronalen Netzwerk separat durchgeführt. Typischerweise wird die Operation vor der Aktivierungsfunktion einer neuen Schicht angewendet. Durch diese Normalisierung wird die Abhängigkeit von der Skalierung und Verschiebung der Eingaben reduziert. Dadurch kann der Lernprozess erheblich stabilisiert und die Konvergenz der Kostenfunktion beschleunigt werden.

### 2.3.10 Häufige Probleme beim Trainieren von neuronalen Netzen

Beim Trainieren von neuronalen Netzen stehen Entwickler/-innen vor einer Vielzahl an Herausforderungen, die sich auf die Leistungsfähigkeit und Stabilität der Modelle auswirken können. Es kann vorkommen, dass die Kostenfunktion nicht konvergiert, der Lernprozess stagniert oder auch dass die Netzwerkausgaben mit fortlaufendem Training an Qualität verlieren. Nachfolgend werden einige dieser Herausforderungen betrachtet.

#### Verschwindende Gradienten

Das Problem der verschwindenden Gradienten [57] tritt auf, wenn die Gradienten der Verlustfunktion, die während des Backpropagation-Verfahrens berechnet werden, sehr klein werden. Dadurch werden die Gewichte in den vordersten Schichten nur sehr langsam oder gar nicht aktualisiert, was das Training stark verlangsamt oder das Training zum Erliegen bringt. Auf dieses Problem wurde genauer in Kapitel 2.3.3 eingegangen.

#### Explodierende Gradienten

Das Gegenstück zu den verschwindenden Gradienten sind die explodierenden Gradienten [61]. Dieses Problem tritt auf, wenn die Gradienten während des Backpropagation-Prozesses exponentiell ansteigen und zu sehr großen Werten führen. Dies kann dazu führen, dass die Gewichte des neuronalen Netzes instabil werden und das Training unbrauchbar machen, da die Modellparameter auf sehr große Werte anwachsen können. Ein offensichtliches Anzeichen für dieses Problem ist, wenn die Zahlenwerte der Gradienten oder der Verlustfunktion auf NaN (Not a Number) anwachsen. Abhilfe bei

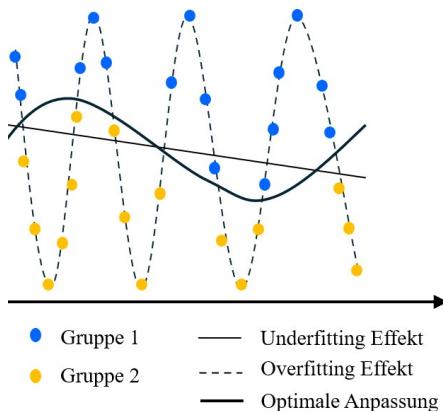


Abbildung 2.14: Over- und Underfitting bei einer Klassifikationsaufgabe

diesem Problem kann beispielsweise durch Batch-Normalisierung, eine andere Gewichtsinitialisierung oder auch Gradient Clipping (Gradienten werden auf einen Maximalwert begrenzt) geschaffen werden.

### Overfitting

Beim Overfittig [76], auch Überanpassung genannt, lernt das Modell durch eine zu hohe Anzahl an Neuronen zu spezifische Eigenschaften aus den Trainingsdaten. Es wird in diesem Fall keine Generalisierung für zukünftige Testdaten erreicht. Außerdem kann es bei einer zu großen Anzahl an Lernzyklen auch zum Overfitting kommen, so dass sich das Netz zu sehr an spezifische Trainingsdaten anpasst und die Generalisierungsfähigkeit verloren geht. Es werden dann ausschließlich ganz spezifische Trainingsdaten, die auswendig gelernt wurden, reproduziert (siehe in Abbildung 2.14 die gestrichelte Linie). Um dem entgegen zu wirken, kann das Netz ausgedünnt werden (auch Pruning genannt) oder die Trainingszyklen frühzeitiger vor dem Nachlassen der Generalisierungsfähigkeit gestoppt werden.

### Underfittig

Beim Underfitting [76], auch Unteranpassung genannt, ist das Modell auf Grund zu weniger Neuronen nicht in der Lage die komplexen, nichtlinearen Strukturen zu erlernen. Dieser Effekt ist in Abbildung 2.14 durch die Gerade dargestellt. Underfitting kann durch eine zu einfache Modellarchitektur oder aber auch unzureichendes Training verursacht werden.

### Probleme mit der Skalierung und Verteilung der Eingabedaten

Ein häufig übersehenes Problem beim Training neuronaler Netze ist eine schlechte Ska-

lierung der Eingabedaten [26]. Wenn die Trainingsdaten ungleichmäßig verteilt sind oder die Eingabedaten nicht richtig skaliert werden, können die Aktivierungen der Neuronen stark variieren. Dies kann dazu führen, dass das Training nur sehr langsam oder gar nicht konvergiert. Um dies zu verhindern, können die Eingabedaten normalisiert werden. Dadurch kann das Training stabiler und effizienter verlaufen.

## 2.4 Reinforcement Learning

Dieses Kapitel bietet einen Überblick über die Grundlagen des Reinforcement Learning und führt in ausgewählte Konzepte und Algorithmen ein. Zunächst werden als Grundlage die prinzipiellen Methoden und Begrifflichkeiten des Reinforcement Learnings vorgestellt. Nachdem die Grundzusammenhänge beschrieben sind, werden verschiedene RL-Algorithmen erläutert. Begonnen wird mit dem Q-Learning und dem daraus entwickelten Deep Q-Learning. Den Abschluss dieses Kapitels bildet der SARSA Algorithmus.

### 2.4.1 Grundlagen

Als Unterklasse des Maschinellen Lernens lernt ein RL-Algorithmus [78] die Belohnung in seiner Umgebung auf lange Sicht zu maximieren. Ein RL-Algorithmus wird immer bei Anwendungen eingesetzt, bei denen es darum geht Entscheidungen zu treffen oder Aktionen durchzuführen. Diese Anwendungsaufgabe wird auch als Steuerungsaufgabe bezeichnet. Ein RL-Algorithmus kann mit dem Trainieren von Hunden verglichen werden. Zum Trainieren neuer Tricks werden oft Leckerlis (Belohnungen) gegeben um gewünschtes Verhalten zu stärken. Bei RL werden Anreize (positive Belohnungen) gegeben ein gesetztes Ziel zu erreichen und Aktionen bestraft, die wir nicht wollen (negative Belohnungen). Es werden bestimmte Verhaltensweisen durch Belohnungssignale verstärkt (to reinforce). Daher auch der Name Reinforcement Learning. Das Grundprinzip ist in Abbildung 2.15 schematisch dargestellt.

In Abbildung 2.16 ist hingegen das RL Standard-Framework mit seinen Komponenten und deren Zusammenspiel dargestellt. Der **Agent** stellt dabei das Herzstück eines jeden RL-Algorithmuses dar. Er verarbeitet die Eingabedaten und bestimmt daraufhin die Aktion, die zu ergreifen ist. Das Umfeld, in dem der Agent agiert und seine

## 2 Grundlagen

---

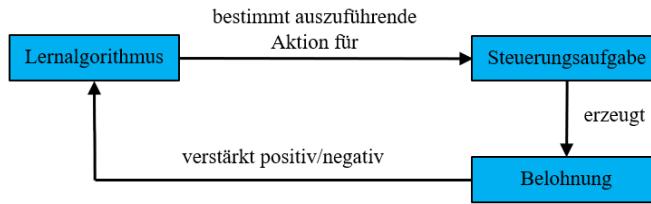


Abbildung 2.15: Grundprinzip RL

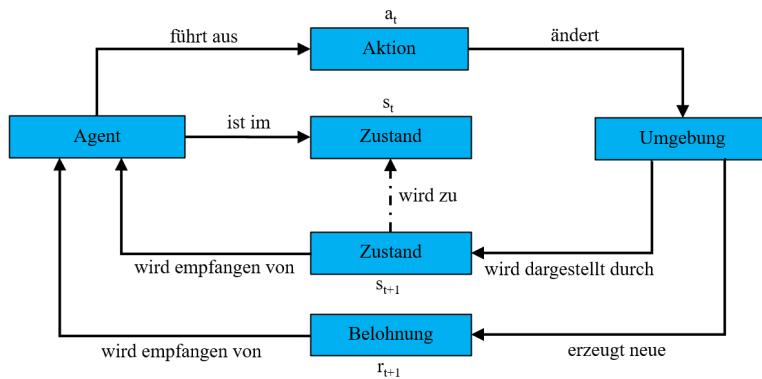


Abbildung 2.16: RL Standard-Framework

Aktionen ausführt, heißt **Umgebung** (environment). Sie stellt einen dynamischen Prozess dar (Funktion der Zeit), in dem sich die Daten kontinuierlich verändern können. Um diese Umgebung in einem Algorithmus abzubilden, werden Momentaufnahmen der Umgebung gebildet. Auf Basis dieses diskreten Zeitschrittes, welcher auch als **Zustand** (state) bezeichnet wird, trifft der Agent seine Entscheidungen. Eine **Aktion** (action) kann zu einer Änderung der Umgebung durch die zuvor getroffene Entscheidung des Agenten führen, beispielsweise das Bewegen eines der Drehräder des BRIO Labyrinthes. Nachdem eine Aktion ausgeführt wurde, wird dem Agenten ein Feedback gegeben, wie gut die ausgeführte Handlung im Bezug auf das Erreichen des globalen Ziels war. Dies geschieht mit Hilfe von positiver oder negativer **Belohnung** (reward). Der Agent hat das primäre Ziel diese Belohnung über die Zeit zu maximieren und Aktionen zu vermeiden, die eine negative Belohnung bringen.

Bei der Umsetzung eines RL-Algorithmus ist zuerst das Gesamtziel zu definieren, dieses wird auch als Zielfunktion bezeichnet. Zudem braucht der Agent Eingabedaten in Form des aktuellen Zustandes (Zustandfunktion). Zu guter Letzt wird noch der RL-Algorithmus selbst benötigt, der aus den gegebenen Daten lernen kann seine Belohnung zu maximieren. Damit wird dann der Lernzyklus des Agenten mit folgenden Schritten wiederholt:

1. Zustandsinformationen beobachten und verarbeiten,
2. Aktion wählen,
3. Aktion ausführen,
4. Belohnung ermitteln

### **Markov Decision Process (MDP)**

Ein Spiel (oder eine andere zu lösende Aufgabe) wird als Markov Decision Process bezeichnet, wenn es die Markov-Eigenschaft aufweist [78]. Das Ziel eines MDP ist es, eine Richtlinie (Policy, wird nachfolgend noch genauer betrachtet) zu finden, die angibt, welche Aktion der Agent in jedem Zustand ausführen sollte, um die Belohnung langfristig zu maximieren. Um diese Richtlinie zu erlernen wird der zuvor beschriebene zyklisch Lernprozess des Agenten durchlaufen. Dabei beschreibt die Markov-Eigenschaft, dass die Aktionsauswahl nur vom aktuellen Zustand abhängt und unabhängig von vergangenen Zuständen getroffen wird. Hierbei beeinflussen die gewählten Aktionen nicht nur die unmittelbare Belohnung, sondern auch die zukünftigen Zustände und dadurch zukünftige Belohnungen. Somit beinhalten MDPs verzögerte Belohnungen und die Notwendigkeit bei der Aktionswahl, zwischen unmittelbaren und verzögerten Belohnungen abzuwagen. MDP-Probleme können in zwei Kategorien, die modellfreie und die modellbasierte Methode, klassifiziert werden [28]. Beim modellbasierten Verstärkungslernen wird versucht Umgebungs- oder Spielregeln zu bestimmten und erst daraus die optimale Handlungsrichtlinie abzuleiten. Hierbei wird sozusagen erst einmal die Umgebung modelliert um daraus Schlussfolgerungen fürs Handeln ziehen zu können. Hingegen versuchen modellfreie Methoden die optimale Handlungsrichtlinie zu bestimmen ohne die Prinzipien der Umgebung oder Spielregeln zu verstehen. Die meisten Standard RL-Algorithmen nutzen die modellfreie Methode, da die modellbasierte Methode sehr viel komplexer ist. Die in den folgenden Unterkapiteln betrachteten Lernalgorithmen wie Q-Learning und SARSA nutzen die modellfreie Methode.

### **Policy**

Die zuvor erwähnte Richtlinie (Policy) [78] entscheidet, welche Aktion bei welchem Zustand ausgeführt werden soll, und stellt somit die Strategie des Agenten dar. Ziel des Agenten ist es eine Richtlinie zu erlernen, die die Gesamtbelohnung maximiert. Daher ist es ein zentrales Konzept des zuvor beschriebenen Markov-Entscheidungsprozesses (MDP). Die Richtlinie kann dabei deterministisch, stochastisch oder zufällig sein [48]. Bei einer deterministischen Richtlinie wird immer die Aktion gewählt, bei der die meiste Belohnung erwartet wird (wird nachfolgend als Ausbeutungsstrategie vorgestellt). Bei einer stochastischen Richtlinie bekommt man eine Wahrscheinlichkeitsverteilung über

jede mögliche Aktion. Entsprechend dieser Wahrscheinlichkeiten wählt dann der Agent eine Aktion aus. Eine zufällige Aktion wird bei der zufälligen Richtlinie gewählt (wird nachfolgend als Erkundungsstrategie vorgestellt).

**Erkundung und Ausnutzung:** Zu Beginn eines Spiels kann der Agent auf keine Erfahrungen zurückgreifen und es muss somit zuerst die Umgebung erkundet werden (Exploration). Bei der Erkundungsstrategie werden die Aktionen im Wesentlichen nach dem Zufallsprinzip gewählt, um die unbekannte Umgebung zu erforschen und im Verlauf möglichst hohe Belohnungen zu finden. Im Gegensatz dazu wird bei der Ausnutzungsstrategie (Exploitation) auf bereits gelerntes Wissen bzw. bereits gemachte Erfahrungen über die Umgebung zurückgegriffen, die die Belohnung bei gegebenen Wissenstand maximiert. Die gesamte Strategie muss eine gute Mischung von Erkundung und Ausnutzung besitzen um die Belohnung zu maximieren [78].

**Epsilon-Greedy-Policy:** Eine weit verbreitete Möglichkeit um die gute Mischung zwischen Erkundung und Ausnutzung zu finden, ist die Epsilon-Greedy-Policy [64]. Dabei wird die Erkundungsrate (exploration rate)  $\epsilon$  eingeführt. Der Wert von  $\epsilon$  liegt dabei im Bereich  $0 < \epsilon < 1$ . Die  $\epsilon$ -greedy Strategie wählt mit einer Wahrscheinlichkeit von  $1 - \epsilon$  die Aktion mit der höchsten bisher bekannten Belohnung aus, während mit einer Wahrscheinlichkeit von  $\epsilon$  eine zufällige Aktion gewählt wird, um die Exploration zu fördern. Bei einem  $\epsilon$ -Wert von null würde deshalb ausschließlich auf bereits gelerntes Wissen zurückgegriffen werden (Ausnutzungsstrategie), bei einem Wert von eins würde ausschließlich die Umgebung erkundet werden. Bei der Anwendung wird eine zufällige Zahl zwischen 0 und 1 generiert. Ist diese Zahl kleiner als  $\epsilon$ , dann wird eine zufällige Aktion gewählt. Wenn die zufällige Zahl größer als oder gleich  $\epsilon$  ist, wird die Aktion mit der größten erwarteten Belohnung gewählt. Oft wird zu Beginn des Trainings der  $\epsilon$ -Wert hoch eingestellt, um eine intensive Erkundung zu ermöglichen. Im Laufe der Zeit wird dieser Wert allmählich reduziert, um eine Konvergenz zu einer optimalen Strategie zu fördern, welche auf den gesammelten Erfahrungen basiert.

### 2.4.2 Q-Learning

Eine Form des Reinforcement Learning ist das Q-Learning (quality learning [28]). Er gehört zu den häufigsten verwendeten RL-Algorithmen [71] und wurde 1989 von Watkins [75] vorgestellt. Der Name „Q-Learning“ bezieht sich auf eine Methode zur Bewertung der Güte von Aktionen, indem eine Aktionswertfunktion (action-value-function)  $Q(s, a)$

verwendet wird. Diese Funktion schätzt die zu erwartende zukünftige Belohnung für eine gegebene Aktion  $a$  in einem Zustand  $s$ , wobei zukünftige Belohnungen über den Faktor  $\gamma$  diskontiert berücksichtigt werden.  $Q(s, a)$  repräsentiert somit den geschätzten langfristigen Nutzen für den Agenten. Beim Training werden Vorhersagen mit der beobachteten Belohnung  $r$  zusammengefasst und die Werte der Funktion wie folgt aktualisiert [28].

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a)] \quad (2.43)$$

Die Funktionsweise wird anschließend anhand der Formel 2.43 erläutert. Alternativ kann die Formel aber auch in folgender Form dargestellt werden:

$$Q(s, a) \leftarrow (1 - \alpha)Q(s, a) + \alpha \cdot [r + \gamma \cdot \max_{a'} Q(s', a')] \quad (2.44)$$

Zu Beginn sind alle Q-Werte mit null initialisiert, da der Agent anfänglich keine Kenntnisse über seine Umwelt hat. Während des Trainings wählt der Agent eine Aktion  $a$  basierend auf einer Policy und führt diese aus. Daraus ergibt sich ein neuer Zustand  $s'$  und der Agent erhält eine Belohnung  $r$ . Mit dem neuen Zustand wird der Algorithmus erneut ausgeführt wobei die Aktion  $a'$  gemäß einer definierten Policy ausgewählt wird. Die Aktualisierung der Q-Werte erfolgt unter Verwendung der temporalen Differenzmethode (TD). Die Differenz besteht dabei aus der beobachteten Belohnung  $r$ , sowie der mit  $\gamma$  diskontierten Schätzung des Folgezustandes  $\max_{a'} Q(s', a')$  (Erkundung) und der bisher gegebenen Bewertungsschätzung  $Q(s, a)$  (Erfahrung). Der Term  $r + \gamma \max_{a'} Q(s', a')$  stellt dabei die neue Schätzung des Q-Werts dar, die auf den neuesten Erfahrungen basiert. Somit beschreibt die Differenz aus Erkundung und Erfahrung den Lernfortschritt (TD-Fehler):

$$TD = [r + \gamma \cdot \max_{a'} Q(s', a')] - Q(s, a) \quad (2.45)$$

Die Q-Werte werden iterativ auf Basis der gesammelten Erfahrungen aktualisiert. Die Parameter  $\alpha$  und  $\gamma$  werden als Hyperparameter bezeichnet [78]. Sie beeinflussen das Lernen, werden aber nicht während des Lernprozesses angepasst, sondern müssen vorher festgelegt werden. Die Lernrate  $\alpha$  beschreibt hierbei die Gewichtung der neuen Information und liegt im Bereich  $0 < \alpha \leq 1$ . Bei einem Wert von 1 würde der Q-Wert durch den neuen ersetzt werden, bei einem Wert von 0 würden nur alte Werte berücksichtigt werden. Somit steuert die Lernrate wie schnell der Algorithmus von jeder Aktion lernt. Der Diskontierungsfaktor  $\gamma$  liegt im Bereich von  $0 < \gamma \leq 1$  und beschreibt, wie weit zukünftige Belohnungen diskontiert (abgewertet) werden. Ein hoher Diskontierungsfaktor bedeutet, dass langfristige Belohnungen stärker berücksichtigt werden, während

ein niedrigerer Faktor zu einer Optimierung der kurzfristigen, unmittelbaren Belohnung führt. Q-Learning eignet sich gut für Anwendungen mit diskreten Aktionsräumen. Beim Q-Learning handelt sich um einen Off-Policy Algorithmus [74]. Bei Off-Policy Methoden geht es darum die optimale Strategie unabhängig davon, welche Strategie für die Auswahl einer neuen Aktion gewählt wurde, zu erlernen. Das bedeutet, dass der Agent von Daten lernen kann (Q-Wert-Aktualisierung), die von einer anderen Richtlinie generiert wurden als der, die er zu optimieren versucht. Off-Policy Methoden eignen sich gut in Umgebungen, bei denen viel Erkundung nötig ist. Es ist dabei außerdem wahrscheinlicher, dass diese Algorithmen eher die optimale Politik finden.

### 2.4.3 Deep Q-Learning

Einer der Hauptnachteile von Q-Learning besteht darin, dass die Größe der Q-Tabelle sich aus dem Produkt der Anzahl der Zustände und der Anzahl der Aktionen ergibt. Der Algorithmus wird somit bei großen Zustands- oder Aktionsräumen sehr rechenaufwändig und benötigt viel Speicherplatz, um die Q-Werte zu speichern. Ein weiteres Problem ist, dass Q-Learning nur bei bekannten Zuständen funktioniert, also bei Zuständen, die der Agent bereits vorher in seinem Training erkundet hat. Wenn der Agent auf neue, ihm unbekannte Zustände trifft, für die er noch keine Q-Werte gespeichert hat, kann er keine fundierte Entscheidung treffen, da ihm die notwendigen Erfahrungswerte fehlen. Der Algorithmus wäre gezwungen, jeden dieser neuen Zustände durch Ausprobieren zu erkunden, was in komplexen Umgebungen oft nicht praktikabel ist. Um diese Nachteile zu beseitigen wurde der Ansatz des Deep Q-Learnings (auch Deep Q-Network, DQN genannt) im Jahr 2013 von Google's DeepMind entwickelt [51]. Dabei wird das Q-Learning mit tiefen neuronalen Netzen kombiniert. Das neuronale Netzwerk erhält den Zustand als Eingabe und gibt die Q-Werte für alle definierten Aktionen als Vektor aus. Dies ermöglicht es, von bereits bekannten Zuständen auf ähnliche, unbekannte Zustände zu generalisieren und somit auch in neuen Situationen bessere Entscheidungen zu treffen. Diese Unterschiede von Q-Learning und DQN sind in der Abbildung 2.17 verdeutlicht.

### 2.4.4 SARSA

Der Algorithmus „Modified Connectionist Q-learning“ wurde 1994 von Rummery und Niranjan [70] eingeführt. Im Jahr 1996 wurde dieser Algorithmus von Sutton umbenannt

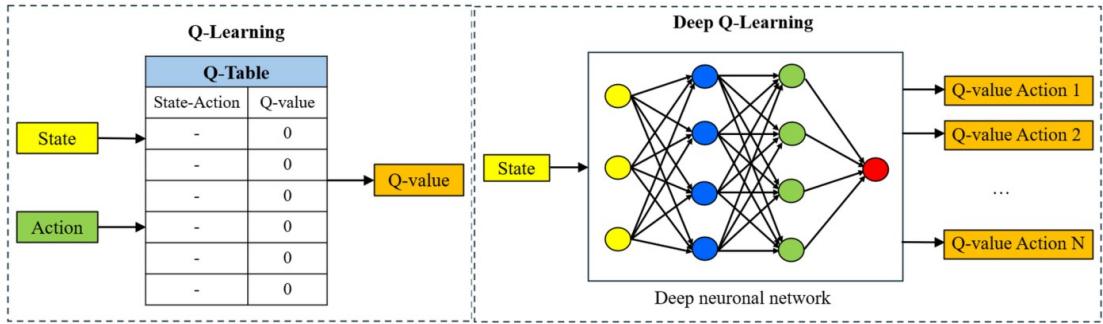


Abbildung 2.17: Vergleich Q-Learning und DQN

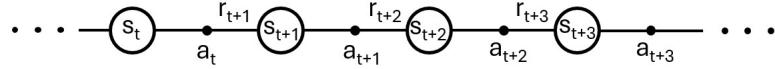


Abbildung 2.18: SARSA

und trägt seitdem den Namen „SARSA“ [74]. Dieses Verfahren ist recht ähnlich zum Q-Learning aufgebaut. Das lässt sich daran erkennen, dass die Formel [74]

$$Q(s, a) \leftarrow Q(s, a) + \alpha \cdot [r + \gamma \cdot Q(s', a') - Q(s, a)] \quad (2.46)$$

zur Aktualisierung der Q-Tabelle beim SARSA Algorithmus, viele Parallelen zur Formel 2.43 des Q-Learning aufweist. Der einzige Unterschied dieser Formeln besteht darin, dass beim Q-Learning die Aktion  $a'$  verwendet wird, die die maximale erwartende Belohnung  $\max_{a'} Q(s', a')$  liefert. Der Lernalgorithmus SARSA hingegen verwendet die Aktion  $a'$ , die auf Grund der Policy gewählt wurde, und nimmt den dazugehörigen Q-Wert  $Q(s', a')$ . Zur Aktualisierung wird ein Vektor mit fünf Elementen ( $s, a, r, s', a'$ ) verwendet. Daher auch der Name des Lernalgorithmus State-Action-Reward-State-Action (SARSA) [28]. Die zeitliche Abfolge der State-Action-Reward-Kette ist zur Verdeutlichung in Abbildung 2.18 dargestellt. SARSA ist ein On-Policy Algorithmus, der die gleiche Policy (typischerweise  $\epsilon$ -greedy) für die Auswahl einer Aktion und die Aktualisierung der Q-Werte verwendet. On-Policy kann deshalb auch als richtlinienkonforme Methode bezeichnet werden. Das bedeutet, dass der Agent Aktionen basierend auf einer Richtlinie auswählt und gleichzeitig versucht, diese Richtlinie zu verbessern. On-Policy Methoden eignen sich gut für Umgebungen, in denen die Sicherheit der Agentenaktionen wichtig ist. Der SARSA-Algorithmus (On-Policy) geht eher einen sichereren Weg und vermeidet mögliche größere Verluste unter Inkaufnahme einer zeitlich nicht optimalen Lösung. Q-Learning (Off-Policy) könnte man als das risikofreudigere Lernen bezeichnen,

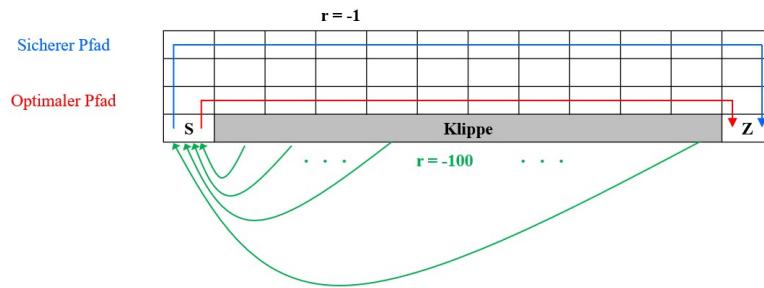


Abbildung 2.19: Vergleich Q-Learning und SARSA

da dieser Algorithmus in der Trainingsphase eher bereit ist auch mal eine Aktion zu wählen, die zu negativen Belohnungen führen kann. Ein Beispiel hierfür ist das in Abbildung 2.19 dargestellte Cliff Walking Beispiel [74]. Der Q-Learning Algorithmus erlernt eher den optimalen Weg (rot) direkt an der Klippe, wohingegen der SARSA Algorithmus eher einen sichereren Weg (blau) weiter weg von der Klippe erlernt. Q-Learning eignet sich eher bei Problemen mit viel Erkundungsbedarf. Beispielsweise wäre es beim Training eines Roboters in der realen Welt sinnvoller den SARSA-Algorithmus zu verwenden, da dieser konservativer vorgeht und somit die Gefahr von Schäden reduziert.

# 3 Stand der Technik

In diesem Kapitel werden ausgewählte öffentlich zugängliche Arbeiten und Forschungsprojekt zur Automatisierung des BRIO Labyrinthes vorgestellt. In einigen dieser Arbeiten wurde künstliche Intelligenz dazu eingesetzt das Labyrinth zu lösen, wobei die Testumgebung (als Simulation oder am realen Spiel) nicht immer dieselbe war. Die verschiedenen Entwicklungen und Lösungen dieser Arbeiten werden nachfolgend genauer dargestellt.

## 3.1 DFKI

Von dem Deutschen Forschungszentrum für Künstliche Intelligenz (DFKI) wurde in mehreren Arbeiten ein BRIO Labyrinth so umgebaut, dass es automatisch gesteuert werden kann [17]. Hierzu wurden die Drehknöpfe ersetzt und jeweils mit einem Servomotor des Typs Dynamixel DX-117 [50] erweitert, wodurch die indirekte Ansteuerung über einen Riemen ermöglicht wird (siehe Abbildung 3.1). Zur Begrenzung des Arbeitsbereiches der Spielfläche in beiden Dimensionen werden zusätzlich vier Schnappschalter [17] eingesetzt. Zudem dienen sie auch zur Kalibrierung der Nullposition der Servomotoren.

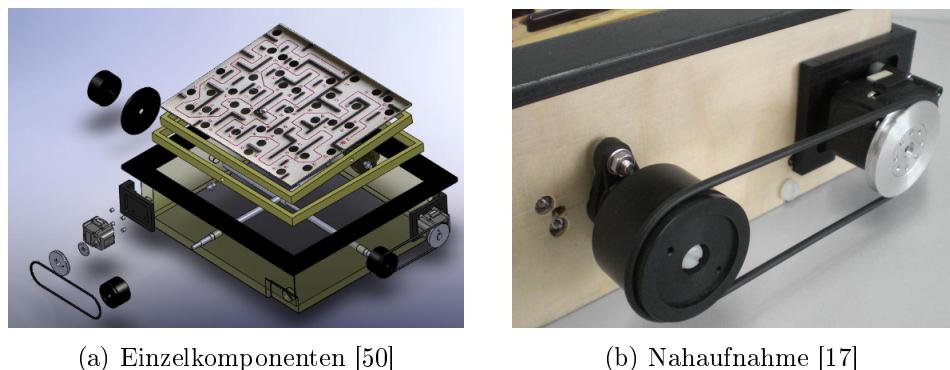


Abbildung 3.1: Motoransteuerung mit Riemen (DFKI)

### *3 Stand der Technik*

---



Abbildung 3.2: Gesamtaufbau (DFKI) [50]

Eine Kamera wurde 1 m [50] über dem Spiel platziert. Mit Hilfe von Bildverarbeitung können die Position und der Geschwindigkeitsvektor der Kugel bestimmt werden. Zu Beginn wird eine automatische Kalibrierung der Kugel relativ zu den Koordinaten des Spielbrettes durchgeführt. Zusätzlich wird die Drehung der Achsen durch zwei Potentiometer bestimmt. Der Gesamtaufbau des vom DFKI erweiterten BRIO Labyrinthes ist in der Abbildung 3.2 dargestellt. Als Erweiterung des physischen Aufbaus wurde noch ein Ballmagazin (siehe Abbildung 3.3) entwickelt. Dieses kann bis zu 14 Kugeln [17] bereitstellen. An der Unterseite des Spiels wurde zudem noch ein Piezo-Sensor [50] angebracht. Dieser erkennt die Erschütterung, wenn eine Kugel durch ein Loch fällt. Durch den Sensor wird somit detektiert, wann eine neue Kugel ins Spiel gebracht werden muss.

Die durch das Training des Agenten erlernte Strategie zur Lösung des Spiels konnte in einem virtuell nachgebauten BRIO Labyrinth gezeigt werden. Die Anwendung der erlernten Strategie auf das reale Spiel ist noch offen [50]. Diese 3D-Simulation (siehe Abbildung 3.4) wurde mit Hilfe der Open Dynamics Engine (ODE) programmiert. Dies ist eine leistungsfähige Bibliothek zur Simulation starrer Körpermodynamiken in Echtzeit[50]. Die für den Zustand des Systems benötigten Parameter Position, Geschwindigkeitsvektor und Drehung der Achsen lassen einen Zustandsraum mit sechs kontinuierlichen Dimensionen [50] entstehen, mit dessen Hilfe der Agent das System kontrollieren kann.

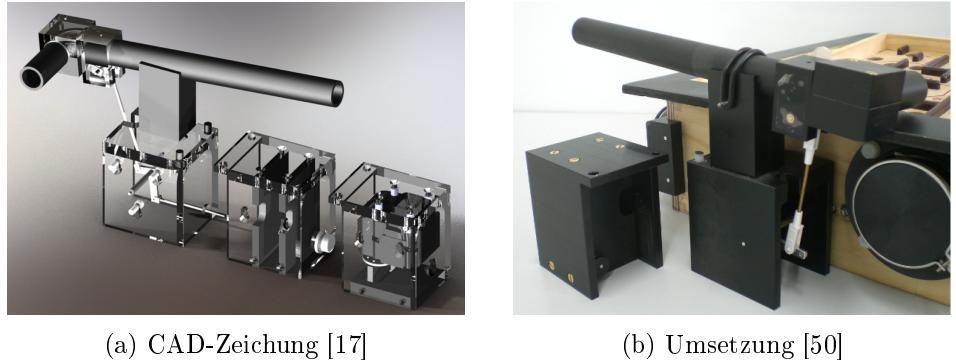


Abbildung 3.3: Ballmagazin (DFKI)

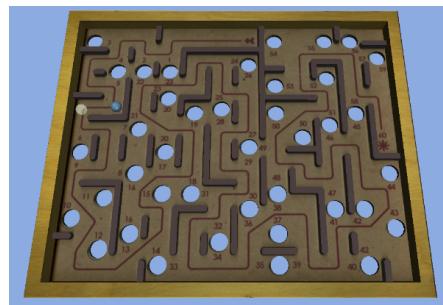


Abbildung 3.4: Simulationsumgebung (DFKI) [50]

Die möglichen Aktionen, die der Agent ausführen kann, beschränken sich auf die Änderung der zwei drehbaren Achsen. Somit kann der Aktionsraum als Menge der erlaubten Positionen für die beiden Achsen  $[-\Phi x, \Phi x] \times [-\Phi y, \Phi y]$  definiert werden [50]. Die Berechnung der Reibung der Kugel basiert in der Simulation auf einer Annäherung an das Coulombsche Reibungsmodell [50]. Nach dem Coulombschen Reibungsmodell wird die Reibung zwischen zwei Festkörpern in Haftreibung (statische Reibungskraft) und Gleitreibung (dynamische Reibungskraft) unterteilt [59]. Im ersten Experiment wurde das SARSA( $\gamma$ ) Lernen mit einem Cerebellar Model Articulation Controller (CMAC) Funktionsapproximator verwendet, um Strategien für die BRIO Labyrinthsimulation zu erlernen. (Die genaue Funktionsweise und mathematischen Grundlagen zu CMAC können in dem Ursprungspaper von Albus nachgelesen werden [1].) Hierzu wurde eine  $\epsilon$ -greedy Erkundung mit einem  $\epsilon = 0,01$  verwendet und die Lernrate wurde auf  $\alpha = 0,1$  gesetzt [50]. Zudem wurde der Diskontierungsfaktor mit  $\gamma = 1,0$  und die Zerfallsrate, mit der die Gewichtung der vergangenen Aktionen und Zustände vergessen werden, mit  $\lambda = 0,9$  festgelegt [50]. Mit diesen Konfigurationen konnte das Ziel nach dem Training von 65000 Spielepisoden erreicht werden [50].



Abbildung 3.5: CyberRunner (ETH Zürich) [15]

### 3.2 ETH Zürich (CyberRunner)

Den Rekord für das Lösen des Labyrinth-Geschicklichkeitsspiels hält der CyberRunner. Wissenschaftler der Eidgenössisch Technischen Hochschule Zürich (ETH Zürich) entwickelten den CyberRunner, welcher das BRIO Labyrinth mit Hilfe von künstlicher Intelligenz innerhalb von 14,48 Sekunden [15] lösen kann. Dafür musste 6,06 Stunden [15] trainiert werden. Der Aufbau des CyberRunners ist in der Abbildung 3.5 dargestellt. Wie auch das zuvor betrachtete Labyrinth des DFKI, wurde das Labyrinth-Spiel für den CyberRunner um eine Kamera, hier eine Weitwinkelkamera See3CAM 24CUG [12] und zwei Servomotoren, hier vom Typ Dynamixel MX-12W [12], erweitert. Hierbei wurden die Motoren direkt auf der Welle befestigt und haben somit die Drehknöpfe komplett ersetzt. Die Motoren wurden mit einem Dynamixel U2D2-USB-Konverter [12] verbunden, welcher es ermöglicht die Motoren über eine einzige USB-Schnittstelle mittels serieller Kommunikation zu steuern. Aus den Kameradaten werden Informationen zur Position der Kugel, des Neigungswinkels der Platte, sowie die Richtungsinformation über den Pfad durch das Labyrinth extrahiert [12]. Für eine bessere Positionsbestimmung der Kugel, wurde diese Blau eingefärbt und somit der Kontrast erhöht. Zusätzlich wurden in die Ecken der Spielfläche blaue Punkte geklebt, um den Neigungswinkel schneller zu erkennen. Die gesamte Software wurde in ROS2 geschrieben. Die Entwickler kombinierten den modellbasierten verstärkenden Lernansatz DreamerV3 (mit Standardparametern) mit einer Datenaugmentationstechnik [12] und nutzen ein Belohnungssystem

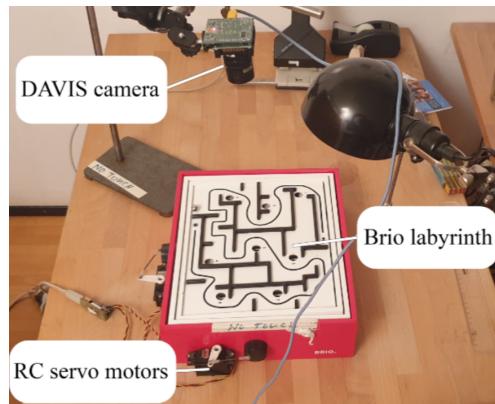


Abbildung 3.6: Prototypischer Aufbau (Kantonsschule Stadelhofen) [2]

basierend auf dem erzielten Fortschritt durch das Labyrinth. (Genauere Informationen zum Lernalgorithmus DreamerV3 sind dem Ursprungspaper zu entnehmen [29].) Vor dem Training wurde eine Sequenz von Wegpunkten festlegen, die den Weg durchs Labyrinth beschreibt, den das System abfahren soll. Während des Lernprozesses muss hierbei die Kugel immer wieder manuell an die Anfangsposition gelegt werden. Die Entwickler planen in der Zukunft das CyberRunner-System zu veröffentlichen und für die Allgemeinheit zur Verfügung zu stellen.

### 3.3 Kantonsschule Stadelhofen

Als weitere Entwicklung im Bereich des BRIO Labyrinthes im Zusammenhang mit künstlicher Intelligenz lässt sich noch ein Poster zu einer Abschlussarbeit [2] finden. Die genutzte prototypische Entwicklung ist in der Abbildung 3.6 dargestellt. Wie in dieser Abbildung zu erkennen ist, wurden die Drehknöpfe um jeweils einen Standard-Servo-Motor erweitert. Zudem wurde oberhalb des Spielfeldes eine Kamera befestigt. Die Hauptaufgabe dieser Arbeit bestand in der praktischen Umsetzung eines Algorithmus zur Objekterkennung und -verfolgung, genauer gesagt der Kugel auf dem Spielfeld. Im Rahmen der Arbeit wurde hierzu ein Fully Convolutional Neural Network CueNet entwickelt.

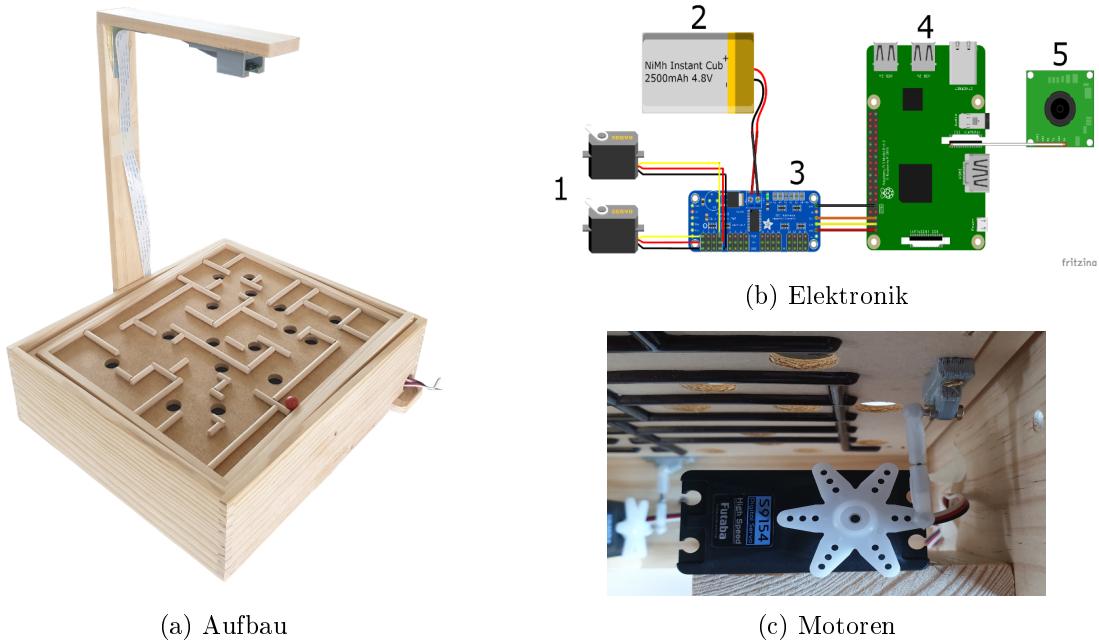


Abbildung 3.7: Labyrinthweiterung der ersten Masterarbeit (Linköping University) [24]

### 3.4 Linköping University

Als letzte recherchierte Arbeiten, die zu dem behandelten Themengebiet in dieser Arbeit passen, sind zwei aufeinander aufbauende Masterarbeiten der Linköping University [24, 23] zu erwähnen. Diese Masterarbeiten wurden jeweils als Gruppenarbeit mit zwei beteiligten Personen erstellt.

Die erste Masterarbeit [24] befasst sich mit dem Hardwareaufbau (siehe Abbildung 3.7), sowie der Lösung des Labyrinthes durch einen Gain-Scheduling LQR Regler mit Hilfe eines Spline-Pfads. Im Vergleich zu den zuvor betrachteten Projekten lässt sich erkennen, dass die Servo Motoren, vom Typ Futaba S9154 Digital High Speed Servo, nicht außerhalb des Gehäuses angebracht wurden, sondern von innen. Als weitere Hardwaredkomponenten wurde ein Adafruit 16-Channel 12-bit PWM/Servo Driver PCA9685, sowie ein Raspberry Pi mit kompatiblem Kameramodul verwendet. Zur Spannungsversorgung wurde ein Vierer-Pack AA Batteriezellen verwendet. Außerdem wurde die Kugel rot eingefärbt, um den Kontrast zu erhöhen. Für die Lösung des Labyrinthes wurde der Spline-Pfadverfolgungsansatz gewählt. Dadurch können glatte Bewegungen durch das Labyrinth erzeugt werden, anstatt diskrete Punkte zu verfolgen. Dieser An-

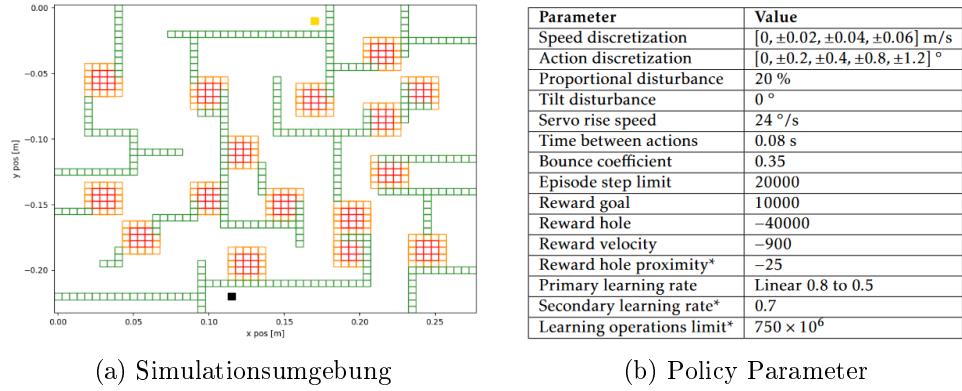


Abbildung 3.8: Simulation und genutzte Policy Parameter der zweiten Masterarbeit (Linköping University) [23]

satz in Kombination mit dem Gain-Scheduling LQR Regler ermöglicht es den Weg durch das Labyrinth zu bestreiten ohne KI zu verwenden.

In der darauf aufbauenden Masterarbeit [23] wurde nun zum Lösen des Labyrinthes KI verwendet. Das Training der KI wurde zum Großteil in einer simulativen Umgebung (siehe Abbildung 3.8(a)) durchgeführt, dabei wurde die Methode Q-Learning verwendet. Die Erfolgsrate das Ziel in der Simulation zu erreichen, lag bei einem Labyrinth mit 16 Löchern mit der besten RL Policy (siehe Abbildung 3.8(b)) bei 66%. Der physikalische Aufbau des erweiterten Labyrinthes wurde von der vorherigen Masterarbeit übernommen, nur dass die Spielfläche grün eingefärbt wurde, um den Farbkontrast zu erhöhen.

### 3.5 Zusammenfassung vorhandener Arbeiten

Alle vorhandenen Arbeiten haben gemeinsam, dass die ursprünglichen Labyrinthe um zwei Servomotoren und eine Kamera zur Zustandserkennung erweitert wurden. Außerdem ist zu keinem der Arbeiten irgendein Programmcode öffentlich verfügbar. Zum Teil sind die Informationen auch nur als Poster oder in einem sehr kurzen Paper veröffentlicht worden und somit ist die Informationslage über die Umsetzung sehr spärlich gehalten. Außerdem wurde jeweils nur eine spezielle Spielplatte, sprich ein ausgewähltes Labyrinth trainiert. Worin sich die vorhandenen Umsetzungen unterschieden ist auch der Umsetzungsumfang. Im Matura Paper [2] wurde bspw. nur die Kugelerkennung mit Hilfe von Bildverarbeitung aufgeführt und nicht die Lösung des Labyrinthes. Die erste

### *3 Stand der Technik*

---

Tabelle 3.1: Kurzzusammenfassung vorhandener Arbeiten

Name	Motoranbringung	KI-Methodik	Testumgebung
DFKI [50]	mit Riemen außen	SARSA( $\gamma$ ) Lernen	Simulation
ETH Zürich [12]	auf der Welle außen	DreamerV3	Realität
Schule Stadelhofen[2]	mit Lenkstange außen	-	-
Linköping University Masterarbeit 1 [24]	mit Lenkstange innen	-	Realität
Linköping University Masterarbeit 2 [23]	mit Lenkstange innen	Q-Learning	Simulation, Realität

Masterarbeit der Linköping University hat zur Lösung des Labyrinthes keine KI verwendet, sondern einen Gain-Scheduling LQR Regler mit Hilfe eines Spline-Pfads [24]. Die vorgestellten Arbeiten mit deren verwendetet KI-Methodiken zur Lösung des Labyrinthes und Testumgebungen der KI sind nochmal als Übersicht in der Tabelle 3.1 dargestellt.

# 4 Anforderungsanalyse

Eine Anforderungsanalyse bildet die Grundlage für ein erfolgreiches Projekt. Sie dient dazu, dass die Anforderungen und das Projektziel klar verstanden und dokumentiert sind. Beginnend mit einer umfassenden Systemübersicht werden die Hauptkomponenten der Systemumgebung identifiziert und deren Zusammenhang verdeutlicht. Anschließend werden verschiedenste Stakeholder analysiert. Denn der Erfolg eines Projekts hängt oft davon ab, ob die Bedürfnisse und Erwartungen aller beteiligten Interessensgruppen berücksichtigt und erfüllt werden. Zu guter Letzt werden die Anwendungsfälle und vor allem Systemanforderungen vorgestellt, dabei wird die virtuelle Umgebung und der physische Demonstrator separat betrachtet. Die Anforderungen legen fest, welche Funktionen das System haben soll und wie oder unter welchen Bedingungen das System seine Funktionen ausführen soll. Dieses Kapitel legt somit den Grundstein für die nachfolgende Entwicklung und die spätere Evaluierung mit entsprechend angepassten Testszenarien. Durch eine sorgfältige und umfassende Anforderungsanalyse kann gewährleistet werden, dass das entwickelte System den Bedürfnissen der Stakeholder entspricht und die gesteckten Anforderungen und Ziele erfolgreich erreicht werden können.

## 4.1 Systemumgebung

Für eine bessere Projektübersicht sind in der Abbildung 4.1 die Hauptkomponenten und deren Zusammenhang dargestellt. Der physische Demonstrator umfasst das BRIO Labyrinth, welches durch Motoren automatisiert wird. Zur Ansteuerung der Motoren benötigt es Steuerungssoftware. Damit der KI-Agent lernen kann das Labyrinth zu lösen, braucht es Daten zum aktuellen Zustand des Labyrinthes. Diese Informationen werden durch ein Kamerasystem und Bildverarbeitungssoftware bereitgestellt. Außerdem wird noch ein Digitaler Zwilling (virtuelles Abbild eines realen Gegenstandes, Systems oder Prozesses [44]) des BRIO Labyrinthes erstellt. In der virtuellen Umgebung sollen verschiedene Methodiken bzw. Parameter im Bereich Reinforcement Learning getestet

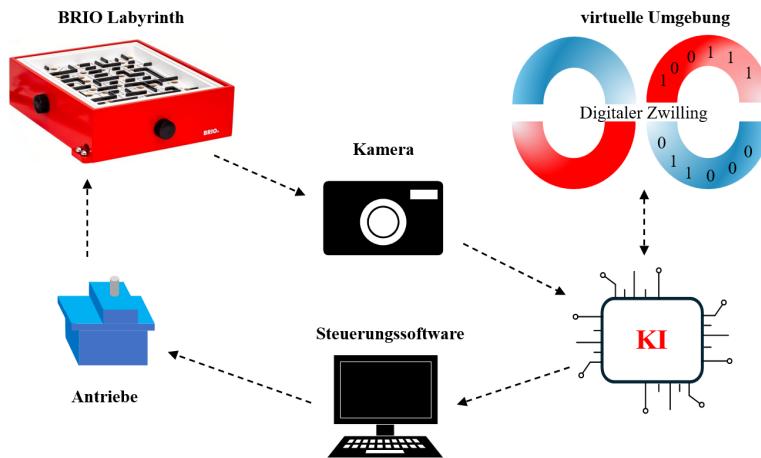


Abbildung 4.1: Systemumgebung [13]

werden. Diese Parameter des RL-Algorithmus können in die reale Umgebung übertragen werden oder auch mit dem realen Systemverhalten verglichen werden.

## 4.2 Stakeholder

Für den Projekterfolg spielen nicht nur die technische Machbarkeit und Umsetzung, sowie die funktionellen Anforderungen eine Rolle. Es ist ebenso wichtig, die Bedürfnisse und Erwartungen sämtlicher involvierter Interessensgruppen zu berücksichtigen und zu erfüllen. Diese Interessensgruppen oder Personen werden Stakeholder genannt. Sie haben einen direkten oder indirekten Einfluss auf das Projekt und dessen Erfolg. Im Falle dieser Abschlussarbeit sind die nachfolgend genauer betrachteten Stakeholder am Gelingen des Projektes interessiert.

### 4.2.1 Auftraggeber

Als Erstprüfer und Betreuer dieser Abschlussarbeit ist Herr Prof. Dr. Hensel ein zentraler Stakeholder, der auch die Rolle des Auftraggebers übernimmt. Hierbei werden verschiedene Ziele und Interessen verfolgt. Zunächst besteht das Interesse darin, ein funktionsfähiges BRIO Labyrinth zu erhalten, das von künstlicher Intelligenz gesteuert wird. Dadurch kann gezeigt werden, wie innovative Technologien und Theorien praktisch

angewendet werden können. Durch die praktische Anwendung von künstlicher Intelligenz im BRIO Labyrinth soll gezeigt werden, wie diese Technologie dazu beitragen kann, komplexe Probleme zu lösen und autonomes Verhalten zu ermöglichen. Darüber hinaus kann das BRIO Labyrinth auch als Werbemittel fungieren, um die Möglichkeiten und Potenziale von künstlicher Intelligenz in virtuellen und realen Umgebungen zu präsentieren. Zudem kann das Projekt auch als Anwendungsbeispiel für Weiterentwicklungen von verschiedenen KI Verfahren und/oder Algorithmen in weiteren studentischen Arbeiten genutzt werden. Aus Sicht des Prüfers liegt das Interesse neben der fachlichen Qualität der Lösung auch auf dem Erkenntnisgewinn und der Wissenserweiterung. Dies umfasst sowohl theoretische Erkenntnisse über die Funktionsweise von KI-Systemen als auch praktische Erfahrungen über deren Implementierung und Anwendung.

#### 4.2.2 Entwicklerin

Die Entwicklerin, als Erstellerin dieser Abschlussarbeit, trägt die maßgebliche Verantwortung für den Erfolg des Projekts. Für diese besteht ein hohes Interesse am erfolgreichen Abschluss dieser Arbeit, bei der die technische Umsetzung realisiert werden muss und der zeitliche Rahmen einer Masterarbeit einhalten werden muss. Dabei liegt der Fokus nicht nur auf der Erfüllung der Anforderungen, sondern auch auf dem Erwerb neuer Fähigkeiten und dem Ausbau des Wissens. Die Anforderungen sind hierbei äußerst vielfältig und anspruchsvoll. Sie reichen von der Entwicklung der Hardware- und Softwarekomponenten des BRIO Labyrinths, einer zu erstellenden Simulationsumgebung bis hin zu komplexen Themen der künstlichen Intelligenz und Bildverarbeitung. Hierbei ist nicht nur technisches Know-how gefragt, sondern auch kreative Problemlösungsfähigkeiten und die Fähigkeit, innovative Lösungen zu entwickeln.

#### 4.2.3 Nutzer

Hier lassen sich mehrere Nutzerkreise identifizieren, deren Interessen und Bedürfnisse eine wichtige Rolle für das Projekt spielen. Zum Einen wären dies Studierende oder Studieninteressierte, die sich anschaulich einen Eindruck von den Ergebnissen studentischer Arbeiten verschaffen wollen. Das Projekt stellt eine gute Plattform dar, um das Interesse an künstlicher Intelligenz und technischer Innovation zu vertiefen. Es bietet Einblicke in die Möglichkeiten der Technologie (KI) und macht sichtbar, wie theoretische Konzepte in der Praxis umgesetzt werden. Zum Anderen ist das der Nutzerkreis

## **4 Anforderungsanalyse**

---

der Spieler, die sich parallel an einem zweiten BRIO Labyrinth messen wollen. Dadurch fungiert das Projekt nicht nur als Bildungs- und Forschungsinstrument, sondern kann auch als eine unterhaltsamen Freizeitaktivität genutzt werden. Spieler können dabei ihre strategischen und problemlösenden Fähigkeiten verbessern, während sie gleichzeitig Spaß haben und sich mit einer KI messen.

### **4.2.4 Personen im Bereich Weiterentwicklung**

Das System und die Ergebnisse dieser Arbeit können vor allem für Studierende als Basis für Weiterentwicklungen bzw. Arbeiten genutzt werden. Insbesondere für Informatik- und Ingenieursstudierende bietet ein automatisiertes BRIO Labyrinth und die Simulation eine spannende Umgebung, theoretisches Wissen in praktischer Anwendung umzusetzen. Beispielsweise bietet es noch einige Möglichkeiten um verschiedenste Bildverarbeitungskonzepte zu testen und zu vergleichen. Neben den Studierenden könnten auch Personen der allgemeinen Öffentlichkeit von dieser Arbeit inspiriert werden. Das automatisierte BRIO Labyrinth kann hier bspw. als Grundlage für Hobbyprojekte oder eigene Entwicklungen dienen.

## **4.3 Virtuelle Umgebung**

In den beiden folgenden Unterkapiteln werden nun die Anwendungsfälle und Anforderungen an die virtuelle Umgebung genauer analysiert und beschrieben. Dieses bietet einen Leitfaden für die Systementwicklung und Entwicklung von Testszenarien in der Evaluierung.

### **4.3.1 Anwendungsfälle**

In diesem Unterkapitel werden die spezifischen Anwendungsfälle der virtuellen Umgebung näher betrachtet. Dazu ist in der Abbildung 4.2 ein Anwendungsfalldiagramm, zur Übersicht der Anwendungsfälle und deren Beziehungen untereinander, dargestellt. Die grafisch dargestellten Anwendungsfälle werden folgend genauer beschrieben.

Da das BRIO Labyrinth präzise simuliert werden soll, müssen die Proportionen und physikalischen Eigenschaften an die reale Umgebung angepasst werden. Da verschiedenste

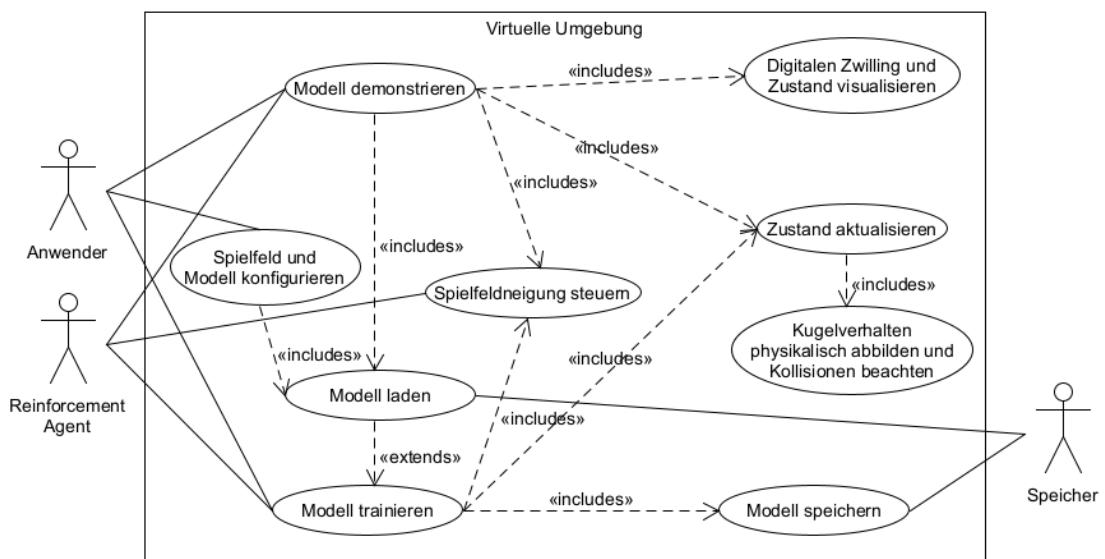


Abbildung 4.2: Anwendungsfalldiagramm der virtuellen Umgebung

RL-Parameterkonfigurationen getestet werden und der Agent verschiedenen Spielplatten und Labyrinthe trainieren soll, müssen diese vom Anwender konfigurierbar sein. Der Anwender oder Benutzer muss zu Beginn die Simulation starten, darauf aufbauend werden verschiedenste weitere Anwendungsfälle aufgerufen und ausgeführt. Dabei kann der Anwender auch die Anzahl der Testdurchläufe angeben. Die grafische Ausgabe der Simulation als Digitaler Zwilling und die Visualisierung des Zustandes sind ein weiterer Anwendungsfall. Der Anwender muss jederzeit diese Visualisierung ein- oder ausschalten können. Sie dient dazu die Funktionsweise der KI oder des RL-Agenten überprüfen und demonstrieren zu können. Zudem können durch die Simulation die Trainingserfolge des Agenten evaluiert werden. Somit muss zu jeder Zeit der aktuelle Spielzustand gezeigt werden können. Nach der gestarteten Simulation kann damit begonnen werden ein KI-Modell zu trainieren, genauer gesagt einen RL-Agenten. Dabei soll der RL-Agent seine Fähigkeiten in Testdurchläufen erweitern und Gelerntes demonstrieren. Die aus dem Training ermittelten Parameter wie die Gewichte des neuronalen Netztes sollen gespeichert werden, um jederzeit auf alte trainierte Fähigkeiten zurückgreifen zu können. Anschließend können die zuvor gespeicherten Trainingsmodelle und Parameter wieder geladen oder eingelesen werden. Dadurch muss nicht bei jedem Trainingsdurchlauf wieder von vorne begonnen werden, sondern es kann auf schon erlernten Fähigkeiten aufgebaut werden. Um ein Modell trainieren zu können muss dem Agenten kontinuierlich

der aktuelle Zustand des Spielgeschehens übermittelt werden. Dafür muss dieser regelmäßig neu berechnet und aktualisiert werden. Dabei ist es wichtig das Rollverhalten physikalisch abzubilden aber auch Kollisionen mit Wänden zu beachten. Da es durch die Kollision zu einer Richtungsänderung der Kugel kommt und somit die Position und Bewegung der Kugel angepasst werden müssen um den Zustand richtig zu ermitteln. Auf Basis des Zustandes kann der Agent eine nächste auszuführende Aktion wählen und somit die Spielfeldneigung steuern. Hierbei ist die Neigung um zwei Achsen möglich, um die Vertikale und die Horizontale. Die ausgeführte Aktion verändert wiederum den Zustand, welcher wieder aktualisiert werden muss.

### 4.3.2 Anforderungen

Bei den Anforderungen unterscheidet man zwei Kategorien, das sind zum einen die funktionalen Anforderungen und zum anderen die nicht funktionalen Anforderungen [43].

**Funktionale Anforderungen (F)** beschreiben dabei spezifisch den Zweck und die Funktionen oder Aufgaben, die das System erfüllen muss. Somit definieren sie im Wesentlichen was das System tun oder können soll. Wohin gegen **nicht funktionale Anforderungen (NF)** Eigenschaften, Qualitätsmerkmale oder Beschränkungen des Systems beschreiben. Hierbei geht es darum wie oder unter welchen Bedingungen das System seine Funktionen ausführen soll.

Nachfolgend werden nun die Anforderungen an die virtuelle Umgebung (**VU**) genauer beschrieben.

**VU-F1:** Das Rollverhalten der Kugel über das Spielfeld und die Kollisionen der Kugel mit den Wänden sollen physikalisch realistisch modelliert werden.

*Dies trägt zur Vergleichbarkeit der virtuellen der realen Umgebung bei und gewährleistet ein authentisches Spielerlebnis.*

**VU-F2:** Wenn die Kugel in ein Loch fällt, soll das Spiel automatisch neu gestartet werden, um den Spielfluss aufrechtzuerhalten und vor allem das Training des Agenten fortzuführen.

**VU-F3:** Als virtuelles Spielfeld sollen verschiedenste Spielumgebungen und Labyrinthe realisiert werden. Eine Labyrinthplatte ist dabei mit unterschiedlich langen Wänden bestückt, wobei alle Wände eine einheitliche Dicke aufweisen. Diese Wände verlaufen ausschließlich horizontal oder vertikal. Zudem gibt es ein Start- und ein Zielpunkt. Außerdem sind Labyrinthplatten mit Löchern versehen. Mindestens eine Labyrinthplatte

soll das Spielfeld mit den 8 Löchern (HOLES\_8) des realen roten BRIO Labyrinths nachbilden. Es können jedoch auch eigene Spielplatten entworfen werden, die keine spezifischen Startpunkte oder Löcher enthalten müssen.

*Zum einen besitzt die rote Variante des in Abbildung 2.1 gezeigten Labyrinthes mehrere Labyrinthplatten und zum anderen sollen eigene Übungsplatten entworfen und trainiert werden. Die Anforderungen an den Spielfeldaufbau ergeben sich aus dem Aufbau der realen Übungsplatten des BRIO Labyrinthes.*

**VU-F4:** An den Wänden des Spielfeldes prallt die Kugel ab und in den Löchern verschwindet die Kugel.

*Dieses Verhalten wird benötigt um die Realität des BRIO Spiels vergleichbar nachzubilden.*

**VU-F5:** Die Drehgeschwindigkeit der realen Motoren bzw. des Spielfeldes soll in der Simulation mit berücksichtigt werden. Die konkret gewählten Motoren sollen in Kapitel 7 vorgestellt werden.

*Die Berücksichtigung der Drehgeschwindigkeit der Motoren bzw. des Spielfeldes ist sinnvoll um ein realistischeres Verhalten des realen Systems nachzubilden.*

**VU-F6:** Mit Hilfe von KI sollen die nach VU-F3 realisierten Spiel- und Labyrinthplatten trainiert und beherrscht werden.

*Labyrinthplatten werden beherrscht, wenn der Agent die Kugel immer von der Startposition zu der Zielposition steuern kann, ohne dass die Kugel in ein Loch fällt.*

**VU-F7:** Die Parameter des KI-Algorithmus aus der virtuellen Umgebung sollen auch auf die physische Demonstratorumgebung übertragbar sein, um dort die Agenten weiter trainieren zu können.

**VU-F8:** Es soll ein Trainingsmodus und ein Evaluations- bzw. Demonstratormodus geben. Im Trainingsmodus soll der Agent trainiert werden können und im Evaluationsmodus soll der Trainingsfortschritt visualisiert werden können.

**VU-NF1:** Das reale Labyrinth soll grafisch als virtuelle Umgebung in 3D modelliert werden.

**VU-NF2:** Die virtuelle Umgebung sollte das reale BRIO Labyrinth proportional abbilden, um eine einfache Vergleichbarkeit zwischen virtueller und realer Spielumgebung zu ermöglichen. Die reale Spielfeldgröße beträgt 27,4 cm x 22,8 cm. Die reale Wanddicke

## 4 Anforderungsanalyse

---

der Labyrinthplatte liegt bei 0,58 cm und die Wandhöhe beträgt 0,6 cm. Zudem haben die Löcher der BRIO Labyrinthplatten einen Durchmesser von 0,75 cm.

**VU-NF3:** Proportional zu den Abmessungen (Radius: 6,35 mm) der Kugel des echten BRIO Labyrinths sollte die virtuell modellierte Kugel modelliert werden, um eine realistische Simulation zu gewährleisten.

*Die Stahlkugel des realen BRIO Labyrinth entspricht der Kugel des GraviTrax Spiels des gleichen Hersteller, welche die oben angegebene Abmessung besitzen [63].*

**VU-NF4:** Das Spielfeld darf sich in x-Richtung um maximal 9° und in y-Richtung um 6° neigen.

*Die Neigung des Spielfeldes wird somit entsprechend der realen Gegebenheiten des BRIO Spiels begrenzt, somit wird die Vergleichbarkeit zum realen Spiel gewahrt.*

**VU-NF5:** Die nächste auszuführende Aktion soll durch einen KI-Agenten, bzw. mit Hilfe eines RL-Algorithmus bestimmt werden.

*Reinforcement Learning stellt im Bereich maschinellem Lernens den vielversprechendsten Allzweck-Lernalgorithmus dar [78].*

**VU-NF6:** Die Kriterien für die Beschreibung des Zustandes und das Erhalten einer Belohnung sollen im Konzept definiert werden. Außerdem ist dort auch der mögliche Aktionsraum zu beschreiben.

*Der Zustand dient als Eingabedaten für den Agenten, auf Basis dessen er eine mögliche Aktion für sein Handeln auswählt (Ausgabedaten). Durch die Belohnung ist es erst möglich, dass der Agent lernen kann.*

**VU-NF7:** Die Implementierung der virtuellen Umgebung soll in Python erfolgen, um eine nahtlose Integration mit gängigen RL-Frameworks zu ermöglichen.

*Die heutzutage beliebtesten Frameworks für Deep-Learning und RL-Algorithmen nutzen die Programmiersprache Python [78].*

**VU-NF8:** Die Programmimplementierung soll strukturiert und gut kommentiert erfolgen, um eine gute Lesbarkeit und eine einfache Wartbarkeit zu ermöglichen.

*Dafür soll mit Klassen gearbeitet werden, um eine bessere Gliederung in zusammenhängende Bereiche zu erhalten, zu dem sollen aussagekräftige Namen für Variablen, Methoden oder auch Klassen verwendet werden.*

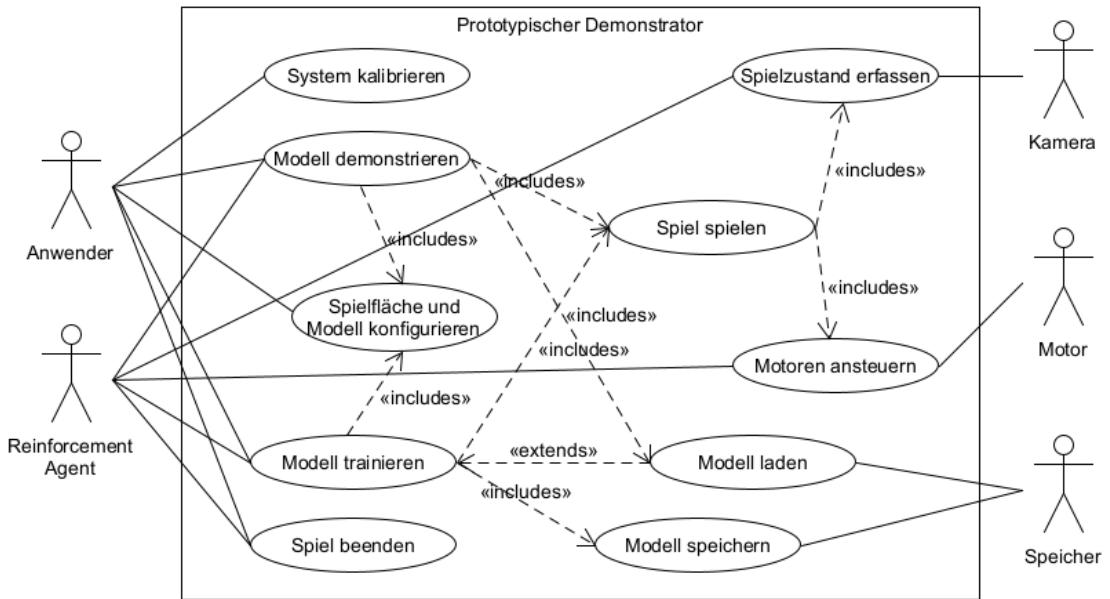


Abbildung 4.3: Anwendungsfalldiagramm des physischen Demonstrators

## 4.4 Physischer Demonstrator

In den beiden folgenden Unterkapiteln werden zum Schluss der Anforderungsanalyse die Anwendungsfälle und Anforderungen an den physischen Demonstrator genauer analysiert und beschrieben. Dieses bietet einen Leitfaden für die Systementwicklung und Entwicklung von Testszenarien in der Evaluierung.

### 4.4.1 Anwendungsfälle

In diesem Unterkapitel werden die spezifischen Anwendungsfälle des physischen Demonstrators näher betrachtet. Hierzu ist in Abbildung 4.3 ein entsprechendes Anwendungsfalldiagramm, zur Übersicht der Anwendungsfälle und deren Beziehungen untereinander, dargestellt. Die grafisch dargestellten Anwendungsfälle werden nachfolgend genauer erläutert.

Bevor das Spiel gestartet werden kann, muss eine Kugel vom Anwender eingelegt werden. Anschließend kann der Anwender das Spiel starten, darauf aufbauend werden verschiedenste weitere Anwendungsfälle aufgerufen und ausgeführt. Zu Beginn eines neuen Trainingsdurchlaufes, welcher mehrere Spiele beinhalten kann, muss eine Kalibrierung

## 4 Anforderungsanalyse

des Kamerasytems und ggf. auch der Motoren durch geführt werden. Nach einem gestarteten Spiel kann damit begonnen werden ein KI-Modell zu trainieren, genauer gesagt einen RL-Agenten. Dabei soll der RL-Agent seine Fähigkeiten in Trainingsdurchläufen erweitern und Gelerntes demonstrieren. Die aus dem Training ermittelten Parameter des Reinforcement Learnings sollen gespeichert werden, um jederzeit auf alte trainierte Fähigkeiten zurückgreifen zu können. Zuvor gespeicherten Trainingsmodelle und Parameter können bei einem späteren Trainingsdurchlauf wieder geladen oder eingelesen werden. Dadurch muss nicht bei jedem Trainingsdurchlauf wieder von vorne begonnen werden, sondern es kann auf schon erlernten Fähigkeiten aufgebaut werden. Um ein Modell trainieren zu können muss dem Agenten kontinuierlich der aktuelle Zustand des Spielgeschehens übermittelt werden. Diese Zustandsinformationen können mit Hilfe eines Kamerasytems und Bildverarbeitungsalgorithmen extrahiert werden. Auf Basis des übermittelten Zustandes kann der Agent eine nächste auszuführende Aktion wählen und somit die Spielfeldneigung über zwei Motoren steuern. Hierbei ist die Neigung um zwei Achsen möglich, um die Vertikale und die Horizontale. Die ausgeführte Aktion verändert wiederum den Zustand, welcher wieder aktualisiert werden muss. Aus Sicherheitsgründen muss der Anwender jederzeit in der Lage sein das Spiel zu beenden und somit die Motoren zu stoppen. Außerdem wird das Spiel von dem Reinforcement Agenten beendet, wenn eine Kugel in ein Loch fällt.

### **4.4.2 Anforderungen**

Wie schon in der virtuellen Umgebung werden auch beim physischen Demonstrator (**PD**) funktionale (**F**) und nicht funktionale Anforderungen (**NF**) unterschieden, die im Folgenden genauer erläutert werden.

**PD-F1:** Es muss eine Steuerungssoftware geben, die es ermöglicht die Motoren gezielt anzusteuern, wodurch das Spielfeld dann horizontal oder vertikal geneigt werden kann.

**PD-F2:** Das Stoppen der Motoren muss bei Bedarf jeder Zeit möglich sein, um die Sicherheit der Benutzer zu gewährleisten.

**PD-N3:** Vor dem Beginn jedes Spieles soll die Spielplatte zurück in die 0° Ausgangslage gedreht werden.

**PD-F4:** Ein manuelles Starten des Spiels soll ermöglicht werden.

*Dadurch wird ermöglicht, dass die Kugel zurück an eine Startposition gelegt werden kann und sichergestellt ist, dass der Vorgang abgeschlossen ist.*

**PD-F5:** Für die Ermittlung des aktuellen Zustandes soll ein Kamerasystem eingesetzt werden sowie OpenCV (Open Computer Vision Library) für die Bildverarbeitung.

**PD-F6:** Ein aus der Simulation trainierter Agent soll auf eine reale Übungsplatte angewendet und weiter trainiert werden.

*Die Funktionsweise der Automatisierung des realen Labyrinthes und des Agenten sollen an einer realen Spielplatte gezeigt werden.*

**PD-F7:** Es soll ein Trainingsmodus und ein Evaluations- bzw. Demonstratormodus geben. Im Trainingsmodus soll der Agent trainiert werden können und im Evaluationsmodus soll der Trainingsfortschritt gezeigt werden können.

**PD-NF1:** Im Rahmen dieses Projektes dürfen Neuanschaffungen die Kosten von 250 € nicht überschreiten.

*Diese Obergrenze der Projektkosten ergibt sich aus den Vorschriften des Departments Informations- und Elektrotechnik.*

**PD-NF2:** Die Spielfeldgröße beträgt 27,4 cm x 22,8 cm.

*Die Abmessungen ergeben sich aus dem eingesetzten BRIO Labyrinth.*

**PD-NF3:** Es soll mit einer Stahlkugel gespielt werden. Diese hat einen Radius von 6,35 mm und wiegt 9 g.

*Die Stahlkugel ist bei dem eingesetzten BRIO Spiel enthalten, sie entspricht zudem den Kugeln des GraviTrax-Spiels vom gleichen Hersteller und hat die oben angegebenen Abmessungen [63].*

**PD-NF4:** Das Drehrad, dass den inneren Rahmen des Spielfeldes kippt darf eine maximale Auslenkung von 6° erreichen. Das zweite Drehrad, welches den äußeren Rahmen des Spielfeldes neigt, darf eine maximale Auslenkung von 9° erreichen. Diese beiden maximalen Auslenkungen müssen bei der Motorsteuerung berücksichtigt sein und dürfen nicht überschritten werden. Beim Überschreiten dieser Spielfeldneigungen, könnte es ansonsten zu irreparablen Schäden an der Spielumgebung oder den Motoren kommen.  
*Die beiden maximalen Auslenkungen ergeben sich durch die mechanischen Möglichkeiten des existierenden BRIO Labyrinthes.*

**PD-NF5:** Die Motorenanbringung für die Automatisierung des Spiels soll ohne invasiven Eingriff erfolgen.

*Das Spiel soll anschließend noch einfach in seiner ursprünglichen Form gespielt werden können.*

**PD-NF6:** Die nächste auszuführende Aktion soll mit Hilfe eines RL-Algorithmus bestimmt werden.

*Reinforcement Learning stellt im Bereich maschinellem Lernens den vielversprechendsten Allzweck-Lernalgorithmus dar [78].*

**PD-NF7:** Die Kriterien für die Beschreibung des Zustandes und das Erhalten einer Belohnung sollen im Konzept definiert werden. Außerdem ist dort der mögliche Aktionsraum zu beschreiben.

**PD-NF8:** Bei der Umsetzung eigener einfacherer Labyrinthspielplatten müssen die Löcherpositionen mit Löchern der fest eingesetzten BRIO-Grundspielplatten übereinstimmen.

*Die Übereinstimmung der Löcher ist wichtig, damit die Kugel auch hineinfallen kann und nicht auf der Grundplatte liegen bleibt.*

**PD-NF9:** Der physische Demonstrator soll gut transportierbar sein. Er soll deshalb kompakt zusammenzupacken sein.

*Der Demonstrator soll an verschiedensten Orten verwendbar sein, beispielsweise in der Hochschule oder zu Hause.*

**PD-NF10:** Als Programmiersprache fürs Reinforcement Learning soll Python verwendet werden.

*Die heutzutage beliebtesten Frameworks für Deep-Learning und RL-Algorithmen nutzen die Programmiersprache Python [78].*

**PD-NF11:** Die Programmimplementierung soll strukturiert und gut kommentiert erfolgen, um eine gute Lesbarkeit und einfache Wartbarkeit zu ermöglichen.

# 5 Konzept

In diesem Kapitel werden verschiedene Aspekte bezüglich zu nutzender Frameworks, Definitionen von Aktions- und Zustandsräumen, mögliche Belohnungsstrategien und ein Grundkonzept des Agentenaufbaus vorgestellt. Außerdem werden Grundkomponenten und Designüberlegungen für den physischen Demonstrators detailliert beschrieben.

## 5.1 Simulation und Software

Dieses Unterkapitel beschreibt verschiedene Konzeptaspekte der Softwarearchitektur, Simulation und vor allem des KI-Lernens detailliert. Dazu gehören die Wahl der Programmiersprache, der Simulationsumgebung sowie des Machine Learning Frameworks. Für das Training und die Performance des Agenten zudem sind die Zustandsrepräsentation und die Aktionsauswahl entscheidend. Mögliche Umsetzungen werden deshalb genauer erläutert. Ebenso wird auf die Definition und Implementierung des Belohnungssystems eingegangen, welches den Lernprozess des Agenten maßgeblich beeinflusst. Abschließend wird der Agent selbst betrachtet, einschließlich seiner Struktur, seiner Lernmechanismen und seiner Interaktionen innerhalb der Simulationsumgebung.

### 5.1.1 Programmiersprache

Die Hauptaufgabe dieser Arbeit besteht darin, das BRIO Labyrinth mit Hilfe von künstlicher Intelligenz zu lösen. Die heutzutage beliebtesten Frameworks für Deep-Learning und RL-Algorithmen nutzen die Programmiersprache Python [78]. Auch in dieser Arbeit wird auf diese Programmiersprache zurückgegriffen. Python zeichnet sich durch eine einfache Syntax aus, die zudem eine gut strukturierte und objektorientierte Programmierung ermöglicht. Des Weiteren bietet Python eine Vielzahl von Bibliotheken an, die speziell für Reinforcement Learning oder auch für 3D-Simulationen entwickelt

## 5 Konzept

---

wurden. Die gängigsten Entwicklungsumgebungen (IDE = Integrated Development Environment) für Python sind dabei unter anderem PyCharm, Visual Studio Code, Anaconda oder Jupyter Notebook. In dieser Abschlussarbeit wird PyCharm verwendet, damit dieser IDE bereits die meisten persönlichen Vorerfahrungen vorlagen.

### 5.1.2 Simulationsumgebung

Mit Hilfe einer Simulationsumgebung können Agenten unter leicht reproduzierbaren und kontrollierten Bedingungen getestet werden. Für die 3D-Simulation wird **VPython** genutzt. Diese Python-Bibliothek ist auf die 3D-Darstellung spezialisiert und ermöglicht es einfach Visualisierungen und Animationen zu erstellen. VPython bietet die Möglichkeit 3D-Objekte wie beispielsweise Kugeln, Boxen oder auch eigene Formen zu erzeugen und diese Objekte hinsichtlich ihrer Position, Rotation oder Skalierung zu manipulieren. Es bietet zudem eine perfekte Plattform um Konzepte der Mathematik und Physik durch interaktive 3D-Visualisierungen zu veranschaulichen. Für die Entwicklung und das Training von KI-Modellen ist **OpenAI Gym** ein äußerst nützliches Werkzeug. OpenAI Gym wurde im April 2016 von der Firma OpenAI veröffentlicht und wird seitdem stetig weiterentwickelt. Es bietet eine Vielzahl von standardisierten Umgebungen, Algorithmen und Evaluierungsverfahren, was die Entwicklung und den Vergleich von KI-Modellen erleichtert. OpenAI Gym dient als standardisierte Schnittstelle. Sie ermöglicht es RL-Agenten mit verschiedenen Umgebungen, in vordefinierten oder eigenen, zu interagieren. Das Kernstück dieser Python-Bibliothek ermöglicht es einem Agenten mit einer Umgebung zu interagieren.

### 5.1.3 Machine Learning Framework

Die beliebtesten Deep-Learning-Frameworks sind Keras, TensorFlow und PyTorch [28]. Diese Frameworks können in der Programmiersprache Python verwendet werden und bieten umfangreiche Funktionen zur Entwicklung von Machine-Learning Modellen und Deep-Learning Modellen. TensorFlow [61] wurde 2015 von Google und PyTorch im Jahr 2017 von Facebook AI Research veröffentlicht. PyTorch hat sich zum beliebtesten Machine-Learning-Framework im Bereich der Forschung entwickelt [62]. TensorFlow hingegen ist das beliebtere Framework in der Industrie [62]. Die Keras-Schnittstelle<sup>1</sup> wurde mittlerweile in Tensorflow (2.0) integriert, ist aber auch mit PyTorch nutzbar.

---

<sup>1</sup>[https://keras.io/keras\\_3/](https://keras.io/keras_3/) - Zugriffssdatum: 15.06.2024

Die Wahl zwischen den einzelnen Frameworks hängt von den spezifischen Anforderungen und vor allem von den Vorlieben des Benutzers ab [28].

Zu Beginn dieser Arbeit wurde ein erster Entwicklungsansatz in Tensorflow mit der Keras-Schnittstelle unternommen, bei dem ein einfaches neuronales Netz mit Agenten entstand. Diese ersten Versuche führten jedoch nicht zu den gewünschten Lernerfolgen. Daraufhin wurde theoretisches Wissen über Konzepte wie beispielsweise Deep Q-Learning, Batchnormalisierung und Replay-Buffer (der Replay-Buffer wird im Kapitel 5.1.6 erläutert) vertieft oder aufgebaut. Dieses neue Verständnis machte deutlich, dass eine angepasste und erweiterte Programmstruktur sinnvoll ist, um die gewünschten Ergebnisse zu erzielen. Um einfacher von den ersten Entwicklungsstrukturen wegzukommen und die neuen theoretischen Erkenntnisse effizient umzusetzen, wurde sich entschieden, einen neuen Entwicklungsansatz in PyTorch zu verfolgen. Dieser ermöglichte erste kleine Lernerfolge und wurde somit weiter verfolgt und verfeinert. Im Rahmen dieser Arbeit wurden somit einfache neuronale Netze und Agenten in Tensorflow mit der Keras-Schnittstelle und in PyTorch getestet. Schlussendlich überzeugte **PyTorch** und der dortige Ansatz wurde weiter vertieft.

#### **5.1.4 Zustandsrepräsentation und Aktionsauswahl**

##### **Zustandsraum**

Die Zustandsrepräsentation spielt beim Reinforcement Learning eine zentrale Rolle, da sie die Grundlage für die Entscheidungsfindung des Agenten bildet. Ein Zustand stellt dabei eine Zusammenfassung der relevanten Informationen über die Umgebung dar. Um den Zustandsraum zu definieren sind verschiedenste Lösungsansätze denkbar. In den Forschungsarbeiten des DFKI, die im Kapitel 3.1 vorgestellt wurden, wurde für den Zustand die Position, der Geschwindigkeitsvektor und die Drehung der Spielplattenachsen verwendet. Beim Testen am realen Demonstrator ist es einfacher aus der Kamera die Positionen des Balls zu bestimmen als einen Geschwindigkeitsvektor zu berechnen, der sich beispielsweise bei einer Wandkollision auch zwischendurch noch verändern kann. Somit kann der Zustand auch die letzten Ballpositionen, anstatt des Geschwindigkeitsvektors enthalten. Als weitere Möglichkeit für den Zustandsraum kann auch eine Bildaufnahme verwendet werden, welche dann als Netzinput für ein Faltungsnetz (siehe Kapitel 2.3.2) dienen kann. Dadurch könnte nicht nur eine ganz spezifische Spielplatte erlernt werden, sondern allgemein das Prinzip von Wänden, Löchern und der Kugelbewegung durch

## 5 Konzept

---

ein Labyrinth. Das komplexere Konzept der Bildaufnahme als Netzinput wird in dieser Arbeit aufgrund der begrenzten Bearbeitungszeit zurückgestellt.

### Aktionsraum

Der Aktionsraum ist ein weiterer wichtiger Aspekt neben der Zustandsrepräsentation. Der Aktionsraum definiert die Menge aller möglichen Aktionen, die ein Agent in einem bestimmten Zustand ausführen kann. Dabei gibt es verschiedenste Konzepte. Man unterscheidet diskrete und kontinuierliche Aktionsräume. In diskreten Aktionsräumen gibt es eine endliche Anzahl von möglichen Aktionen, die der Agent auswählen kann. Dabei würden beispielsweise die Drehachsen auf konkrete ausgewählte Winkelstellungen gedreht werden können. Kontinuierliche Aktionsräume erlauben eine unendliche Anzahl von möglichen Aktionen. Dabei würden beispielsweise die Drehachsen um kontinuierliche Werte wie den Drehwinkel gedreht werden können. Beim kontinuierlichen Ansatz muss berücksichtigt werden, dass keine maximalen Winkelgrenzen überschritten werden, da sonst Schäden am realen Demonstrator die Folge sein könnten. In dieser Arbeit wird der diskrete Ansatz des Drehens der Drehachsen auf konkrete ausgewählte Winkelstellungen verfolgt. Bei der Drehung auf konkrete Winkelstellungen gibt es zwei verschiedene Ansätze in der Umsetzung:

1. Gleichzeitige Drehung beider Achsen (x- und y-Achse)
2. Drehung jeweils nur einer Achse pro Aktion

Bei gleichzeitiger Einstellung der beiden Drehachsen ergibt sich ein Aktionsraum der Größe

$$n_{actions} = n_x \cdot n_y , \quad (5.1)$$

wobei  $n_x$  bzw.  $n_y$  die Anzahl der möglichen Winkelstellungen der x- bzw. y-Achse beschreibt. Bei der Einstellung nur einer Drehachse zur Zeit, ergibt sich ein kleinerer Aktionsraum der Größe

$$n_{actions} = n_x + n_y . \quad (5.2)$$

Nach anfänglichen Versuchen mit dem gleichzeitigen Einstellen beider Drehachsen wurde sich dafür entschieden, nur eine Achse pro Aktion zu wählen. Die Einstellung nur einer Drehachse zur Zeit hat mehrere Vorteile:

#### 1. Reduzierung des Aktionsraums

Ein kleinerer Aktionsraum hat zur Folge, dass der Agent weniger verschiedene Aktionen berücksichtigen muss. Dies vereinfacht das Training und reduziert die benötigte Rechenzeit.

## 2. Vermeidung von Fehlentscheidungen durch unabhängige Netze

Eine Möglichkeit wäre es zwei separate Netzwerke für die x- und y-Achse zu verwenden. Allerdings könnten sie nicht effektiv kommunizieren. Eine richtige Entscheidung in einer Achse könnte durch eine falsche Entscheidung in der anderen Achse negiert werden bzw. zu einer negativen Belohnung beider Netze führen. Es könnte zudem dazu kommen, dass der Agent bzw. ein neuronales Netz für die gleiche oder eine sehr ähnliche Aktion zwei verschiedene Rückmeldungen (Belohnungen) erhält, da diese immer an die Aktion des anderen Netzes gekoppelt ist. Dies führt dazu, dass der Agent unter Umständen keine plausiblen Schlüsse mehr ziehen kann und somit nichts mehr lernt, was auch als katastrophales Vergessen (siehe [78]) bezeichnet wird. Dies würde die Lernkurve sehr erschweren oder sogar das Lernen unmöglich machen. Alternativ wäre es sehr aufwendig die Belohnung für jede einzelne Achse separat zu ermitteln. Aufgrund dieser Probleme stellt dies keine sinnvolle Umsetzung dar. Das Kommunikationsproblem wie es zwischen unabhängigen Netzwerken auftreten kann, kann bei der Einstellung einer Achse pro Aktion nicht entstehen.

## 3. Erlernt die wichtigere Handlungsrichtlinie

Durch die Beschränkung auf eine Achse pro Aktion kann der Agent präziser lernen, welche Rotationsrichtung in einem gewissen Zustand am sinnvollsten ist. Dies fördert ein fokussierteres und effektiveres Lernverhalten.

## 4. Analogie zum menschlichen Verhalten

Menschen sind in der Regel nicht gut im echten Multitasking. Auch wenn sie Aufgaben parallel bearbeiten, betreiben sie eher „Task-Switching“, das heißt sie wechseln schnell zwischen zwei Aufgaben hin und her. Der Mensch dreht somit an den Drehknöpfen des BRIO Spiels oft zeitlich leicht versetzt. Ähnlich wie ein Mensch, der sich besser auf eine Aufgabe zur Zeit fokussieren kann, ist es für den Agenten effizienter, jeweils nur eine Achse zu drehen. Dies minimiert die Komplexität der Entscheidung und verbessert die Lernleistung.

### 5.1.5 Belohnung

Belohnungen (rewards) stellen ein zentrales Element im Reinforcement Learning dar, da der Agent lernen soll, seine Aktionen in einer gegebenen Umgebung zu optimieren, bzw. die Belohnung zu maximieren. Für die Belohnungen gibt es zahlreiche Konzepte, die nachfolgend erläutert werden.

## Ziel

Ziel des BRIO-Labyrinthes ist es die Kugel durch ein Labyrinth ans Ziel zu manövrieren. Daraus ergibt sich eine grundsätzliche Belohnung. Das Ziel ist mit einer großen positiven Belohnung zu versehen. Alle weiteren Belohnungsstrategien sind mögliche erweiterte Ansätze.

## Loch und Lochnähe

Das Spiel ist verloren, wenn die Kugel in ein Loch fällt, dies muss auf jeden Fall verhindert werden. Deshalb werden Löcher mit einer negativen Belohnung ausgestattet. Da die Nähe zu einem Loch bereits ein gewisse Risiko darstellt wird die Nähe zu einem Loch auch mit einer negativen Belohnung bewertet. Bei Lochnähe fällt die Belohnung jedoch betragsmäßig geringer aus, als der Fall in das Loch selbst, bei dem das Spiel verloren geht.

## Labyrinthfortschritt

Um dem Problem der spärlichen Belohnung entgegen zu wirken, ist es sinnvoll den Labyrinthfortschritt positiv zu belohnen. Das Problem der spärlichen Belohnung wird in [78] genauer beschrieben. Verschiedenste Fortschrittsbelohnungen werden nachfolgend beschrieben.

*Schwellenbelohnung:* Wenn die Kugel gewisse Streckenabschnitte bzw. Schwellen überwunden hat, gibt es dafür eine positive Belohnung. Die Herausforderung dabei ist, dass der Agent unter Umständen lernt, die Kugel an diesen Schwellen hin und her zu bewegen und gegebenenfalls nicht den Rest des Labyrinth erkundet. Wenn die Belohnung nur beim ersten Überschreiten gegeben wird, beim zweiten mal aber nicht kann es allerdings zum katastrophalen Vergessen (wurde im vorherigen Unterkapitel erläutert, siehe auch [78]) kommen. Eine andere Möglichkeit wäre es die Überschreitung der Schwelle in richtiger Richtung positiv zu bewerten und die Überschreitung der Schwelle in falscher Richtung negativ zu belohnen. So dass sich das Hin- und Herbewegen bei einer Schwelle nicht rechnet.

*Kacheln und Zwischenziele:* Bei diesem Ansatz wird das Labyrinth in verschiedene Teilabschnitte (Kacheln) geteilt. Jeder Teilabschnitt erhält zudem ein Zwischenziel. Wenn die Kugel den Abstand zwischen sich und dem Zwischenziel durch die gewählte Aktion verkleinern konnte wird die Aktion etwas positiv belohnt, wobei die Belohnung gleichbleibend hoch ist. Die falsche Bewegungsrichtung wird leicht negativ belohnt. Am einfachsten ist es jede Kachel als ein Rechteck zu definieren und nicht andere Formen zu verwenden. Zudem ist es sinnvoll die Rechtecke und Zwischenziele so zu platzieren,

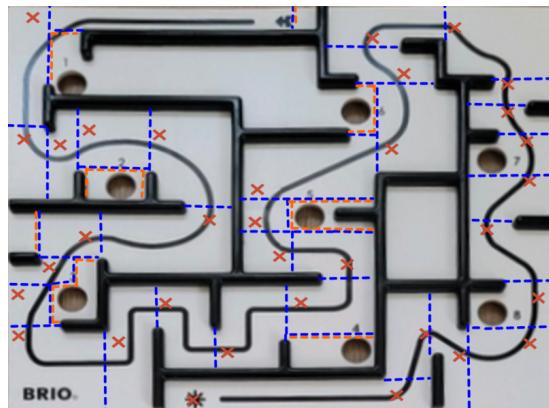


Abbildung 5.1: Fortschritt durch Kacheln und Zwischenziele

dass der direkte Weg zum nächsten Zwischenzielpunkt nicht durch eine Wand versperrt wird oder in ein Loch führt. Außerdem ist es wichtig, dass die Zwischenzielpunkte mit einem gewissen Abstand in Richtung des Ziels hinter ihrer Kachel liegen, so dass die Kugel nicht abgebremst wird, wenn sie von einer Kachel in die nächste übergeht. Die richtige Bewegungsrichtung ist gegeben wenn folgende Bedingung erfüllt ist:

$$\sqrt{(x_k - x_z)^2 + (y_k - y_z)^2} < \sqrt{(x_{k-1} - x_z)^2 + (y_{k-1} - y_z)^2} \quad (5.3)$$

Dabei entsprechen  $x$  und  $y$  den Koordinatenachsen, der Index  $k - 1$  steht für die vorangegangene Kugelposition,  $k$  entspricht der aktuellen Kugelposition und der Index  $z$  beschreibt die nächste Zwischenbelohnungsposition. Schematisch ist der allererste Entwurf so einer Einteilung anhand der 8 Loch Spielplatte in Abbildung 5.1 dargestellt. Dabei sind in blau die Kacheln eingezeichnet, in denen es eine positive Belohnung für die richtige Richtung gibt, und die roten Kreuze zeigen die zu erreichenden Zwischenziele an. Die orangenen Bereiche symbolisieren Bereiche, in denen es keinerlei positive Richtungsbewertung gibt. Im Verlauf des Testens ergaben sich verschiedenste Herausforderungen. Zum Einen kam das Problem des Hin- und Herbewegens an einer Graden auf. Dabei wurde in positiver Richtung langsam vorgerollt und anschließend ging es schnell in falscher Bewegungsrichtung zurück. Zum Anderen bekam die Kugel auch für sehr kleine Positionsänderungen der beispielsweise vierten Nachkommastelle eine positive Belohnung, obwohl die Kugel für den Betrachter an einer Stelle ruhte. Beide Verhalten sind unerwünscht und daraus ergeben sich weitere Belohnungskonzeptideen.

*Belohnungserhöhung anhand des Labyrinthfortschritts:* Der Agent bekommt für jede Kachel, die er näher am Ziel ist, eine höhere Belohnung für die richtige Richtungsbe-

wegung. Das hat zur Folge, dass es sich mehr lohnt weiter um Kurven oder an Löchern vorbei zu gehen, so dass der Agent nicht in definitiv sichereren Bereichen verweilt. Das verbessert das Problem des Hin- und Herbewegens an einer Graden.

*Größere Positionsänderung:* Um das Problem der Belohnung für eine minimale Positionsänderung der vierten Nachkommastelle zu verringern und das schnelle Lösen des Labyrinthes zu bestärken kann die Belohnungsstrategie angepasst werden. Die Positionsänderung muss dabei einen bestimmten Wert pro Aktion in positiver Wegrichtung überschreiten, damit es eine positive Belohnung gibt. Liegt eine Positionsänderung in richtiger Richtung vor, die aber den Schwellwert der Wegänderung nicht übersteigt, so kann die Aktion entweder gleich negativ wie eine falsche Richtung, weniger negativ als die falsche Richtung, mit null (weder positiv noch negativ) oder ganz minimal positiv bewertet werden. Als weitere Idee kann die Belohnung proportional von der Positionsänderung abhängig gemacht werden. Dabei könnten sehr kleine Änderungen nur verschwindend kleine positive Belohnungen nach sich ziehen. Aber auch beispielsweise zu große Positionsänderungen bzw. zu hohe Geschwindigkeiten könnten mit null bewertet werden.

*Vollständig definierter Pfad:* Als Alternative zur Belohnung mit den Kacheln und Zwischenbelohnungen kann auch ein komplett durchgängiger Pfad vordefiniert werden, welcher vom Agenten abgefahren werden muss. Um den dafür benötigten Pfad einzulesen gibt es verschiedenste Ansätze. Eine Möglichkeit besteht darin, dass einzelne Punkte eingelesen werden, die dann mit Hilfe von Interpolation zu einem Pfad verbunden werden. Beispielsweise könnte dabei lineare Interpolation [72] eingesetzt werden. Lineare Interpolation verbindet die einzelnen Punkte durch gerade Linien, dies stellt einen einfachen Ansatz dar. Eine weitere Möglichkeit wäre die Polynominterpolation [66]. Dabei wird ein Polynom

$$f(x) = a_n x^n + a_{n-1} x^{n-1} + \dots + a_1 x + a_0 \quad (5.4)$$

verwendet, welches durch alle gegebenen Punkte verläuft. Bei der Polynominterpolationen kann es zu instabilen und oszillierenden Kurven kommen, was für den Anwendungsfall nachteilig wäre. Außerdem könnte eine Spline-Interpolation [72] verwendet werden. Dabei wird die Gesamtkurve aus Segmenten von mehreren polynomischen Funktionen zusammengesetzt, so dass die gesamte Kurve glatt erscheint. Die drei beschriebenen Interpolationsarten sind beispielhaft in Abbildung 5.2 dargestellt. Eine Alternative zur Interpolation wäre es mit Hilfe von Bildverarbeitung die Wegpunkte aus einem Bild der Spielplatten zu generieren. Dieser Ansatz wird zunächst nicht weiter verfolgt um

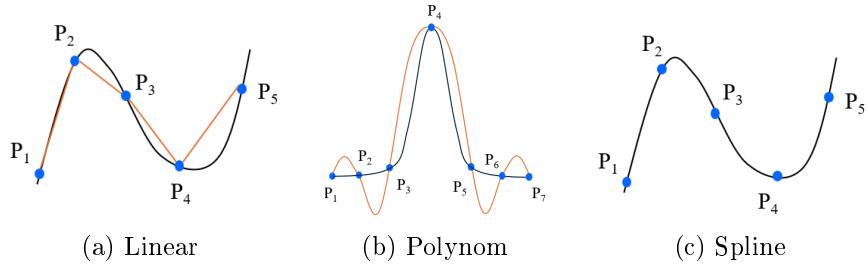


Abbildung 5.2: Interpolationsarten

sich auf die Hauptthemenfelder dieser Arbeit zu konzentrieren. Beim vollständigen Pfad muss die Belohnungsauslegung noch genauer betrachtet werden. Zum einen kann der Fortschritt auf dem exakten Pfad belohnt werden. Dieses erhöht allerdings den Schwierigkeitsgrad erheblich. Da dieser Pfad sehr schmal ist und somit voraussichtlich sehr selten positive Belohnungen erzielt werden. Außerdem ist das Benutzen der Wände in diesem Spiel recht nützlich, da dadurch die Geschwindigkeit erhöht werden kann und es einfacher wird die Kugel durch das Labyrinth zu manövrieren. Der Agent sollte somit auch eine positive Belohnung erhalten, wenn sich die Kugel in der Nähe zu diesem Pfad in die richtige Richtung bewegt.

### Ecken bestrafen

Eine beim Testen öfters aufgetretene Herausforderung ist, dass die Kugel in Ecken hängen und liegen bleibt. Dadurch entstand die Idee, die Ecken (gleichzeitige Kollision mit 2 Wänden) leicht mehr zu bestrafen als eine falsche Richtungsbewegung, wenn sich die Position der Kugel nicht mehr ändert.

### Höhe der jeweiligen Belohnungen

Die Festlegung der expliziten Belohnungen und deren Skalierung sind entscheidend für den Erfolg des Trainings. Die Höhen der Belohnungen müssen die Bedeutung der jeweiligen Aktionen klar widerspiegeln. Sie dürfen nicht zu nah beieinander liegen, sonst sind die Unterschiede zwischen den Aktionen für den Agenten nicht signifikant genug. Sie dürfen aber auch nicht zu extreme Werte annehmen, da es sonst zur Destabilisierung des Lernprozesses kommt. Auch ein zu großes Ungleichgewicht zwischen positiven und negativen Belohnungen kann den Lernprozess behindern.

### 5.1.6 Agent

#### DQN

Der am weitesten verbreitete Deep Reinforcement Learning Algorithmus ist Deep Q-Learning [28]. Er ermöglicht es mit weniger Speicherbedarf und einem geringeren Zeitaufwand Zusammenhänge zu erlernen als beispielsweise Q-Learning. Außerdem erfordert die extrem große Anzahl an Zuständen den Einsatz eines neuronalen Netzes. Allein die Positionen der Kugel (aktuelle oder auch letzte) ließen sich nicht in eine Tabelle fassen, geschweige trainieren. Aus diesen Gründen wird in dieser Arbeit mit Deep Q-Learning begonnen.

#### Neuronales Netz

Für DQN wird ein neuronales Netz benötigt. Bei neuronalen Netzen muss nicht nur die Schichtenanzahl festgelegt werden, sondern auch die Anzahl der Neuronen pro Schicht. Es gibt bisher kein allgemeingültiges Konzept, wie die Anzahl der verdeckten Schichten oder Neuronen in einem neuronalen Netz exakt zu wählen ist. Es sind immer Experimente nötig um eine optimale Struktur zu finden. In der Theorie führt eine Erhöhung der Anzahl der verdeckten Schichten zu einer besseren bzw. komplexeren Abstraktionsfähigkeit des Modells [54]. Allerdings führt eine Erhöhung der Schichtenanzahl auch zu Herausforderungen. Beispielsweise kann es bei tiefen Netzen mit vielen Neuronen schneller zum Overfitting kommen. Dieses Problem wurde in Kapitel 2.3.10 genauer erläutert. Werden zu wenige Neuronen verwendet, kommt es zum Problem des Underfitting (siehe Kapitel 2.3.10). Sind zu viele Schichten in einem Netz vorhanden, kann es auch zum Problem der verschwindenden Gradienten kommen (siehe Kapitel 2.3.3), sodass die ersten Schichten nicht mehr viel lernen. Generell lässt sich sagen, je größer ein neuronales Netz ist, desto mehr Trainingsdaten werden benötigt um alle Neuronen zu trainieren und desto zeitaufwendiger wird das Training. Es muss somit ein guter Kompromiss bei der Schichten- und Neuronenanzahl gefunden werden. Zu Beginn ist es sinnvoll mit kleineren Modellen zu beginnen und diese langsam zu vergrößern. Kleinere Netze benötigen weniger Trainingszeit und Speicherplatz, außerdem kann das Problem der Überanpassung vermieden werden. Als Neuronenanzahl wird oft eine Zweierpotenz gewählt [54]. Nicht nur die Größe des Netzes hat Einfluss auf das Training, sondern auch die gewählte Aktivierungsfunktion und die Gewichtsinitialisierung.

#### Policy

Bei der ursprünglichen Implementierung des DQN-Algorithmus wurde die Epsilon-Greedy Policy (siehe Kapitel 2.4.1) für die Aktionsauswahl verwendet [51]. Dabei wur-

de  $\epsilon$  dynamisch im Trainingsprozess angepasst [51], um einen guten Kompromiss zwischen Exploration und Exploitation zu finden. Diese Strategie wird als decaying epsilon-greedy-policy (Epsilon-Reduzierungs-Strategie) [28] bezeichnet. Dabei ist zu Beginn des Trainings  $\epsilon$  hoch (beispielsweise  $\epsilon = 1$ ), sodass viele zufällige Aktionen ausgewählt werden, um die Umgebung zu erkunden. Mit fortschreitendem Training wird  $\epsilon$  schrittweise reduziert und auf erlerntes Wissen zurückgegriffen. Das heißt über einen definierten Zeitraum wird  $\epsilon$  linear oder exponentiell bis zu einem niedrigen Wert reduziert. Einen guten Wert für die Reduzierungsrate und des minimalen Epsilon-Wertes zu finden, sind entscheidend für ein erfolgreiches Training. Es sind aber auch andere Varianten der Epsilon-Greedy Policy denkbar.  $\epsilon$  kann basierend auf der Performance angepasst werden, zum Beispiel wenn die Belohnung stagniert oder sich verringert wird  $\epsilon$  wieder erhöht. Es kann aber auch mit einem konstanten  $\epsilon$  trainiert werden, wie es bei den Forschungsarbeiten des DFKI getan wurde (siehe Kapitel 3.1). Außerdem könnte  $\epsilon$  beispielsweise je nach Kachelfortschritt angepasst werden. Wenn die ersten Kacheln schon gut beherrscht werden, wird dort Epsilon verringert und bei Kacheln, die weiter weg von der Startposition liegen und somit noch nicht so viel trainiert wurden, bleibt Epsilon höher. Epsilon anhand des Labyrinthkachelfortschrittes anzupassen erfordert mehr Einstellparameter, die aufeinander abgestimmt werden müssen, als die anderen Varianten. Dieses Vorgehen erfordert die Speicherung verschiedener Epsilon Werte und Anwendung der verschiedenen Werte je nach Fortschritt. Zudem sind die Reduzierungsbedingungen, Reduzierungsrate und der minimale Epsilon-Wert aufeinander abzustimmen. Eine Reduzierungsbedingung könnte beispielsweise sein, dass die Kacheln von der Kugel passiert wurden. In dieser Arbeit wird wegen der Einfachheit mit der klassischen Epsilon-Decaying-Strategie begonnen.

### **Experience Replay**

Ein Replay-Buffer [28, 54] wird oft beim Training von DQN eingesetzt, um die Effizienz und Stabilität des Lernprozesses zu verbessern. In einem Replay-Buffer werden vergangene Erfahrungen (Zustand, Aktion, Belohnung, Folgezustand) über mehrere Episoden gespeichert. Aus diesen Erfahrungen werden anschließend zufällige Stichproben (auch als Mini-Batch bezeichnet) ausgewählt, um das neuronale Netz zu trainieren. Zeitliche Abhängigkeiten der einzelnen Datenpunkte werden somit reduziert oder eliminiert. Durch die zufällige Auswahl der Trainingsdaten werden die Gewichtsaktualisierungen stabiler, da sehr ähnliche Datenpunkte nicht direkt aufeinander folgen. Da das Netz mit einer breiten Vielfalt von Zuständen und Aktionen umzugehen lernt, hilft dies dabei besser generalisieren zu können. Ein weiterer Vorteil ist die effizientere Nutzung

## 5 Konzept

---

der Erfahrungen, da diese mehrfach genutzt werden können. Typischerweise besitzt der Replay-Buffer eine begrenzte Größe. Wenn diese Grenze erreicht ist, werden die ältesten Erfahrungen entfernt, um Platz für die Neuen zu schaffen. Dies stellt sicher, dass der Puffer auf dem aktuellen Stand bleibt und relevante Erfahrungen trainiert werden. Durch die Verwendung eines Replay-Buffers kann der Agent effektiver lernen und die Verlustfunktion schneller konvergieren, was letztendlich zu einer besseren Leistung und Generalisierung führt. Für die Anwendung des Replay-Buffer ist die Puffergröße und die Größe der Mini-Batches zu wählen. Die typische Größe von Mini-Batches beim Training von DQN liegt oft im Bereich von 32 bis 256. Kleinere Mini-Batches führen zu schnelleren Gewichtsupdates und sind vorteilhaft bei eingeschränktem Speicher. Sie führen aber auch zu größeren Schwankungen in den Gradienten beim Backpropagation. Größere Batches erhöhen den Rechenaufwand, stabilisieren aber die Gradientenabschätzung. Das Konzept des Replay-Buffers wird auch in dieser Arbeit angewendet.

### Episoden

Eine Episode repräsentiert eine Sequenz von Zuständen, Aktionen und Belohnungen bis ein Terminalzustand erreicht wird. Der Trainingsprozess eines Agents verläuft über viele Episoden hinweg. Der Terminalzustand kann entweder sein, die Kugel ist in ein Loch gefallen oder sie hat das Ziel erreicht. Außerdem ist es sinnvoll ein Abbruchkriterium zu definieren, wenn eine bestimmte maximal Anzahl an Schritten erreicht ist. Dadurch bleibt die Episode endlich, auch wenn die Kugel beispielsweise in einer Ecke hängen geblieben ist oder nur an einer Wand hin und her rollt. Außerdem werden dann nicht zu viele gleiche Erfahrungen im Replay-Buffer gespeichert. Je schwieriger die zu lösende Aufgabe oder je länger das Labyrinth, desto größer sollte die maximale Anzahl an Schritten für das Abbruchkriterium und die Episodenanzahl des Trainingsprozesses sein. Des Weiteren könnten auch die gesammelten Belohnungen ein Abbruchkriterium der Episode darstellen. Die Episode kann beispielsweise abgebrochen werden, wenn eine gewisse Belohnungshöhe (positiv oder negativ) erreicht wurde.

### Startpositionen

Bei der Wahl der Startpositionen für das Training gibt es drei verschiedene Ansätze. Bei jeder Trainingsepisode wird von der ursprünglichen Startposition aus gestartet. Vom richtigen Startpunkt zu beginnen wäre vermutlich das Vorgehen, wie ein Mensch das Spiel üben würde. Es kann aber auch näher am Ziel begonnen werden, bis der Agent gelernt hat zum Ziel zu gelangen. Die Startposition wird dann anschließend langsam immer weiter weg vom Ziel platziert. Dabei wird der Labyrinthweg vom Ziel aus langsam aufgebaut, da es dort eine höhere positive Belohnung gibt. Als dritte Variante können

verschiedenste Startpositionen über das gesamte Spielfeld verteilt zufällig ausgewählt werden. Bei einer breiten Streuung der Startpositionen werden die Spielfeldbereiche von Anfang an gleichmäßiger erkundet.

### Training und Evaluierung

Es sind zwei Bereiche in dieser Arbeit zu unterscheiden, zum einen das Trainieren des neuronalen Netzes und zum anderen das Demonstrieren des Erlernten. Beim Trainieren ist es sinnvoll die Gewichte des neuronalen Netzes in gewissen Episodenabständen (beispielsweise alle 50 oder 100 Episoden) in unterschiedlichen Dateien zu speichern und nicht immer nur eine einzelne Datei zu überschreiben. Zum einen kann dadurch der Trainingsfortschritt im Nachhinein nachvollzogen werden und besser eingeschätzt werden an welchen Stellen gegebenenfalls Probleme aufgetreten sind. Zum anderen ist die benötigte Episodenanzahl für das Erlernen unterschiedlicher Aufgaben verschieden, sodass es im Vorhinein schwierig ist die perfekte Episodenanzahl abzuschätzen. Wenn beispielsweise zu viel trainiert wurde, kommt irgendwann der Punkt, an dem schon Erlerntes vergessen wird und sich das Verhalten verschlechtert. Somit ist es gut auf Zwischentrainingsstände zurückgreifen zu können, um sich die besten Ergebnisse heraussuchen zu können. Der gesamte Trainingserfolg hängt davon ab wie die in diesem Unterkapitel beschriebenen Parameter aufeinander abgestimmt sind. Nur ein gutes Zusammenspiel führt zum Erfolg. Zudem sind noch weitere Parameter anzupassen, wie beispielsweise die Lernrate  $\alpha$ , der Diskontierungsfaktor  $\gamma$ , wie oft aus dem Replay-Buffer gelernt werden soll (Lernperiode) oder auch die Verlustfunktion, um nur einige zu nennen. Beim Trainieren in der simulativen Umgebung ist es sinnvoll ohne die 3D-Darstellung zu arbeiten, um das Training zu beschleunigen. Bei der Evaluierung des Erlernten hilft es bei der Aktionsauswahl nicht ausschließlich den maximalen Q-Wert zu nehmen, sondern eine kleine Zufälligkeit (beispielsweise 1% bis 5%) beizubehalten um das Verhalten zu verbessern.

## 5.2 Hardware

In den nachfolgenden Unterkapiteln werden zum einen verschiedene Konzeptmöglichkeiten zur Motoranbringung am realen Spiel vorgestellt und qualitativ bewertet und zum anderen die Motorauswahl erläutert.



Abbildung 5.3: Direkte Anbringung auf der Welle [12]

### 5.2.1 Mechanik

Für die Anbringung der Motoren zur Automatisierung des BRIO Labyrinthes gibt es verschiedene Umsetzungsideen, die nachfolgend genauer betrachtet werden.

#### Direkte Anbringung auf der Welle

Bei dem CyberRunner System [12] wurde der Motor direkt auf der Welle befestigt (siehe Abbildung 5.3). Dieses könnte bspw. durch eine Wellenkupplung realisiert werden, welche die Welle des Spiels mit der Welle des Motors verbindet. Des Weiteren wird eine Halterung benötigt, die den Motor an der richtigen Position am Spiel befestigt. Ein Vorteil dieser Umsetzung ist ein sehr kompakter, transportabler Aufbau. Jedoch sind dafür grobe Eingriffe in das ursprüngliche Spiel vorzunehmen (Drehknopf entfernen, Halterungen in den Seitenwänden befestigen). Es kann anschließend nicht mehr einfach als unautomatisiertes Spiel verwendet werden.

#### Anbringung über ein Verbindungsstange

Eine weitere Möglichkeit wäre es den Motor über ein Verbindungsstange an dem Drehrad des Spiels anzubringen. Hierbei sind zwei Realisierungen möglich (siehe Abbildung 5.4). Zum einen könnte ein stehender Motor vorne am Spiel, wie in dem Matura Paper zur Objekterkennung anhand eines BRIO Labyrinths (siehe Kapitel 3.3) dargestellt ist, über eine Verbindungsstange am Drehrad befestigt werden. Zum anderen könnte der Motor auch nebenstehend auf gleicher Höhe zum Drehrad befestigt werden. Als Verbindungsstange kann bspw. eine Gelenkstange, RC Lenkstange oder eine Koppelstange eingesetzt werden. Es ist sinnvoll ein Kugel- oder Kreuzgelenk in den Befestigungsenden vorzusehen, da sich der Winkel der Verbindung beim Auslenken verändert. Der Vorteil der stehenden Motoren ist, dass nur ein kleiner Eingriff in das Drehrad zur Befestigung der Gelenkstange erforderlich wird. Bei dem vorne am Spiel befestigten Motor kann voraussichtlich nicht der gesamte Drehwinkelbereich des Drehrades ausgenutzt werden.

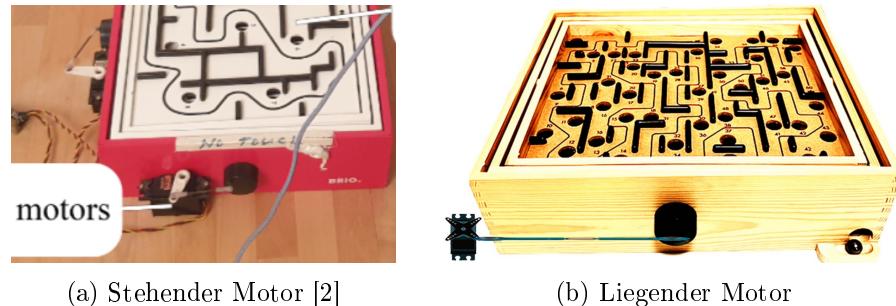


Abbildung 5.4: Anbringung über ein Verbindungsstange



Abbildung 5.5: Innen befestigter Motor [24]

Weiterhin ist bei dieser Anordnung zu beachten, dass die Antriebsachse und die Achse des Drehrades orthogonal zueinander liegen und somit die Winkelübertragung komplexer wird. Die beiden oben beschriebenen Nachteile bei der Anbringung gibt es bei dem neben dem Spiel stehenden Motor nicht, so dass die Rotationsachse des Motors in die gleiche Richtung wie die Rotationsachse des Drehrades weist. Hierbei wäre eine Servohalterung zu entwerfen, um den Motor in dieser Position zum Stehen zu bringen.

### Innen befestigter Motor

Wie in den Masterarbeiten der Linköping University (siehe Kapitel 3.4) verwendet, besteht eine weitere Möglichkeit darin, die Motoren innen im Gehäuse des Spieles zu befestigen (siehe Abbildung 5.5). Als Alternative zu einem drehenden Motor könnte gegebenenfalls auch ein kleiner Linearantrieb eingesetzt werden. Diese Anordnung erfordert ebenfalls eine Lenkstange. Weiterhin werden eine Befestigungshalterung am Spielfeld und eine Befestigung für den Motor am Spiel benötigt. Die Motoren ermöglichen eine Auf- und Abbewegung der Platte. Der größte Vorteil dieser Lösung ist die Optik, da die Motoren im Gehäuse versteckt sind. Nachteilig ist, dass man die Bodenplatte des Spiels lösen müsste um die Motoren ins Gehäuse zu bringen, was einen groben Eingriff darstellt und die Wartung aufwendiger macht. Zum anderen wird durch

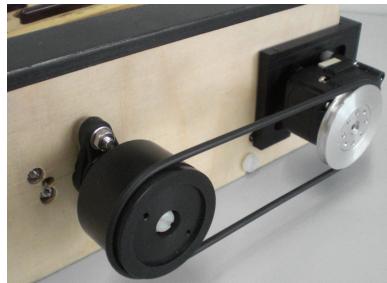


Abbildung 5.6: Nutzung von Riemen [17]

die feste Verbindung der Spielfeldplatte mit den Motoren das Spiel nur noch mit den Motoren nutzbar und kann nicht mehr per Hand gespielt werden.

### Nutzung von Riemen

In verschiedenen Projektarbeiten des DFKI [17] wurden Riemen verwendet (siehe Abbildung 5.6). Der Motor kann dabei an verschiedenensten Positionen angebracht oder hingestellt werden. Er kann, wie in der Abbildung 5.6 zu sehen (vorne am Gehäuse), neben dem Gehäuse auf einer Platte, oder mit einer eigens designten Servoklemmhalterung am Spiel befestigt, bzw. geklemmt werden. Es wird ein Riemen sowie eine Riemenscheibe benötigt. Vorteil dieser Anordnung ist, dass sich ein Übersetzungsverhältnis realisieren lässt und so die Empfindlichkeit eingestellt werden kann. Das Übersetzungsverhältnis

$$i = \frac{d_{Drehknopf}}{d_{Riemenscheibe}} \quad (5.5)$$

kann durch das Verhältnis der Durchmesser  $d$  bestimmt werden. Der Drehknopf kann entweder ersetzt werden oder der originale Drehknopf genutzt werden. Bei der Verwendung des Originaldrehknopfes sollte dieser um eine Halterung erweitert werden, sodass sichergestellt ist, dass der Riemen nicht herunter rutschen kann. Bei dieser Variante ist es von Vorteil, dass die Drehbewegung vom Motor exakt auf das Drehrad übergeben wird. Zudem sind bspw. bei der seitlichen Anbringung des Motors kaum Anpassungen an dem Spiel notwendig, sodass dieses auch ohne großen Aufwand händisch spielbar bleibt. Ein Nachteil könnte sein, dass man nach einem Transport das System neu justieren und kalibrieren muss.

### Nutzung von Zahnrädern

Als weitere Alternative ist die Nutzung von Zahnrädern möglich, wie es in dem beispielhaften Schaubild in Abbildung 5.7 gezeigt wird. Auch diese Lösung bietet die Möglichkeit, dass das Übersetzungsverhältnis durch verschieden große Zahnräder einfach ange-

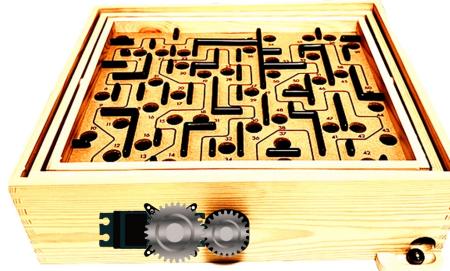


Abbildung 5.7: Nutzung von Zahnrädern

Tabelle 5.1: Motor Anbringungsübersicht

Anbringungsart	Spieleingriff	Klemmprinzip
Welle	-	-
Verbindungsstange im Gehäuse	o	+
Riemen	+	+
Zahnräder	-	-

passt werden kann. Ein Nachteil der Nutzung von Zahnräder ist, dass es eventuell zum Spiel zwischen den Rädern bzw. Zähnen kommt, falls diese nicht perfekt aufeinander abgestimmt sind. Zum anderen sind hier wieder Halterungen am Gehäuse zu befestigen und Anpassungen an den Drehräder zu machen, sodass das Spiel nicht mehr so einfach händisch spielbar ist.

### Auswahl

In Tabelle 5.1 ist eine Übersicht der Anbringungsmöglichkeiten dargestellt. Dabei wird die qualitative Größe der Eingriffe (Modifikationen) am Spiel bewertet. Hier wird auch indirekt deutlich, ob das Spiel weiterhin händisch spielbar bleibt. Dabei beschreibt „-“ einen umfangreichen Eingriff und „o“ einen leichten Eingriff ins Originalspiel. Das „+“ beschreibt keinen Eingriff sondern nur eine leicht reversible Erweiterung. Außerdem wird eine Einschätzung gegeben, ob ein Klemmprinzip oder Steckprinzip möglich ist. An der Spielgehäuseaußenwand ist bis zum Spielboden etwas Platz, so dass eine Motorhalterung angeklemmt werden könnte. Dieses Prinzip ist sinnvoll, da kein Eingriff ins Gehäuse notwendig wird und der Motor dabei dennoch an Ort und Stelle bleibt. Das Kennzeichen „+“ beschreibt, dass das Klemmprinzip einsetzbar ist, und „-“, dass es nicht einsetzbar ist. Auf Grund der zuvor beschriebenen Eigenschaften, sowie der Vor- und Nachteile der einzelnen Anbringungsarten, wird in dieser Arbeit das Riemenprinzip verwendet. Dabei ist kein Eingriff ins Original-Spiel nötig, sodass die reversible Erweiterung leicht entfernt

werden kann. Dies ist beim Transport von Vorteil, zudem bleibt auch das händische Spielen weiterhin möglich.

### 5.2.2 Motorenauswahl

#### Schrittmotor

Wie der Name Schrittmotor [45] schon verrät, dreht sich der Motor in diskreten Schritten. Dies geschieht, indem eine Magnetspule einen Magneten Schritt für Schritt zur nächsten Position bewegt. Bei genauerer Betrachtung handelt es sich bei einem Schrittmotor um einen zweiphasigen Synchronmotor, der aus zwei Wicklungen besteht, die mit Gleichstrom versorgt werden. Durch die unterschiedlich ausgerichteten Felder im feststehenden Stator und im drehenden Rotor sowie dem gezielten Ein- und Ausschalten einzelner Wicklungen des Stators kommt es zur Drehbewegung des Rotors. Die Position und Geschwindigkeit des Motors können durch Umkehrung der Stromrichtung gesteuert werden.

Schrittmotoren benötigen keinen separaten Lagesensor, jedoch können aufgrund ihres gesteuerten Betriebs Schrittfehler [42] auftreten. Daher ist es wichtig zu beachten, dass Schrittmotoren von Natur aus keine Rückmeldung über ihre aktuelle Position geben. In Anwendungen, in denen es entscheidend ist, jederzeit die genaue Position des Motors zu kennen, können Probleme auftreten. Dies ist beispielsweise der Fall, wenn der Motor blockiert ist, sich nicht ordnungsgemäß bewegen kann oder überlastet ist, was dazu führen kann, dass die korrekte Positionsinformation verloren geht. Die Vorteile des Schrittmotors sind die hohe Genauigkeit und die einfache Kontrolle der Rotorposition. Darüber hinaus ermöglicht die einfache Motorsteuerung ein einfaches Starten und Stoppen sowie schnelle Änderungen der Drehrichtung. Im Vergleich zu anderen Motoren sind Schrittmotoren oft kostengünstiger.

Bei Anwendungen mit geringeren Beschleunigungen und großem Haltedrehmoment oder auch hohen Anforderungen an Präzision werden Schrittmotoren eingesetzt. Beispiele dafür sind 3D-Drucker oder Förderanlagen.

#### Servomotor

Servomotoren [46] sind Elektromotoren, die die Winkelposition ihrer Motorwelle präzise kontrollieren können. Diese Motoren zeichnen sich durch die Fähigkeit aus, basierend auf der angelegten Spannung und dem zugeführten Strom ein Drehmoment und eine Beschleunigung zu erzeugen. Im Kern eines Servomotors befindet sich ein magnetischer

Rotor, der mit einem Positionssensor verbunden ist. Dieser Sensor erfasst kontinuierlich die Position des Rotors und übermittelt sie an einen Servoregler, der in der Regel außerhalb des Motors angeordnet ist. Durch fortlaufende Berechnungen der Differenz zwischen der geforderten und der aktuellen Position kann der Servomotor präzise seine Position regulieren. Auf diese Weise bildet der Servomotor ein geschlossenes Regelkreissystem.

Trotz des Vorteils des Feedback-Mechanismus besteht jedoch die Möglichkeit, dass durch ständige Anpassungen in seltenen Fällen leichte Vibrationen auftreten können, obwohl eine Position beibehalten werden soll. Im Vergleich zu anderen Motortypen sind Servomotoren bei gleicher Leistung in der Regel deutlich kompakter und reaktionsschneller. Sie lassen sich äußerst präzise steuern und bieten eine Rückmeldung, mit der Umdrehungen und Winkelstellungen sehr genau erfasst werden können. Somit eignen sie sich hervorragend für Anwendungen, die eine hohe Präzision und Flexibilität erfordern.

Servomotoren werden bevorzugt in Anwendungen mit hohen Drehzahlen oder hochdynamischen Anforderungen eingesetzt, wie beispielsweise in der Robotik oder in Hobbyanwendungen wie ferngesteuerten Autos.

### **Motorauswahl**

Die Entscheidung fiel auf einen Servomotor anstelle eines Schrittmotors. Das Ziel der Anwendung besteht darin, die Kugel durch ein Labyrinth bzw. eine Spielplatte zu steuern. Dabei ist eine präzise Kontrolle der Position entscheidend, welche der Servo bietet. Des Weiteren verfügt ein Servomotor über einen integrierten Feedback-Mechanismus, der kontinuierlich die Position des Rotors überwacht und entsprechend angepasst. Durch den möglichen Positionsinformationsverlust bei Schrittmotoren müssten öfters neue Kalibration durchgeführt werden, was erhöhten Aufwand bedeutet. Die maximale Winkelneigung des Spielfeldes, die durch die physischen Gegebenheiten wie die Mechanik vorgegeben sind, darf nicht überschritten werden. Eine Überschreitung muss zu jeder Zeit ausgeschlossen sein, da es anderenfalls zu irreversiblen Schäden kommen kann. Im Allgemeinen sind Servomotoren im Vergleich zu Schrittmotoren reaktionsschneller, was bei einem dynamischen Labyrinthspiel von Vorteil ist. Das Risiko, dass durch das kontinuierliche Anpassungen der Position durch den Servomotor leichte Vibrationen auftreten, wird als sehr gering eingeschätzt, sodass die Vorteile des Feedback-Mechanismus überwiegen. Wie in Kapitel 5.1.4 beschrieben wird eine Einstellung auf einen bestimmten Winkel als Aktion verwendet. Dies lässt sich mit einem Servomotor einfacher realisieren. Servomotoren wurden auch bei allen schon vorhanden Arbeiten zur Automatisierung des BRIO Labyrinthes eingesetzt (siehe Kapitel 3). Der Rekord im Lösen des Labyrinthes

## *5 Konzept*

---

wurde unter dem Einsatz von Servomotoren erzielt. Dies spricht für diese Technologie und seine Vorteile bezogen auf die Systemanforderungen.

# 6 Entwicklung der virtuellen Umgebung

In diesem Kapitel wird die Entwicklung der virtuellen Umgebung detailliert erläutert. Zunächst wird auf die Trainingsumgebung eingegangen. Anschließend wird der Agent vorgestellt, welcher mit der Umgebung interagiert und dort seine Aktionen ausführt. Die Installationsanleitungen und benötigten Python-Bibliothekspakete, die im Rahmen dieser Arbeit verwendet wurden, sind im Anhang A näher beschrieben.

## 6.1 Übersicht der virtuellen Trainingsumgebung

In Abbildung 6.1 sind die realisierten Klassen der virtuellen Trainingsumgebung in Form einer Klassenübersicht dargestellt. In den nachfolgenden Kapiteln werden die einzelnen Klassen detaillierter betrachtet und erläutert. Dabei wird zuerst das 3D-Modell des Labyrinth Geschicklichkeitsspiels vorgestellt. Anschließend wird die Umsetzung der benötigte Ballphysik für eine realistische Kugelsimulation erläutert. Zum Schluss werden die daraus erstellte OpenAI Gym-Environment und die Rewardklassen vorgestellt.

## 6.2 3D-Modell

Die mit Hilfe der Bibliothek VPython erstellte 3D-Simulationsumgebung ist mit einer exemplarischen Spielplatte in Abbildung 6.2 dargestellt. Diese Simulationsumgebung wird bei entsprechender Programmausführung im Webbrowser wiedergegeben. Insgesamt wurden sechs verschiedene Spielplatten realisiert, die in Abbildung 6.3 dargestellt sind. Es wurden zwei Labyrinthplatten des BRIO Labyrinths realisiert (siehe Abbildung 6.3a und 6.3b), sowie vier selbst designte Spielplatten (siehe Abbildung 6.3c, 6.3d, 6.3e und 6.3f). Die designten Spielplatten wurden erstellt um den Einstieg in das KI Training zu vereinfachen, so dass der Schwierigkeitsgrad mit der dazugewonnenen Erfahrung im Laufe der Zeit schrittweise erhöht werden kann. Es gibt zwei 2 Loch Spielplatten,

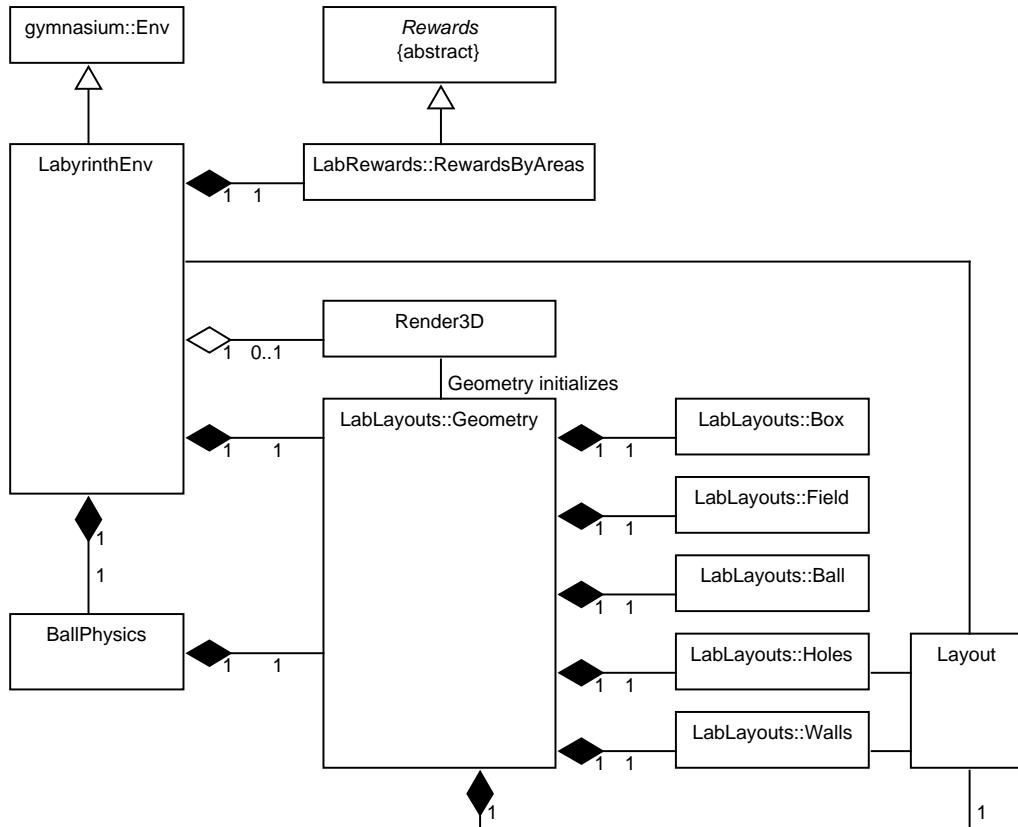


Abbildung 6.1: Klassenübersicht der virtuellen Umgebung

beim ersten Entwurf (siehe Abbildung 6.3c) stimmen die Löcher nicht mit real existierenden Löcherpositionen der BRIO Labyrinth Grundplatte überein. Diese ist somit nicht real umsetzbar, sondern nur in der Simulation spielbar. Bei dem zweiten Entwurf (siehe Abbildung 6.3d) wurde darauf geachtet, dass die Löcherpositionen mit real zur Verfügung stehenden Löchern übereinstimmen und somit die Umsetzung auch in Realität möglich ist.

Für die 3D-Darstellung wurden zwei Pythondateien implementiert, *LabLayouts* und *LabRender3D*, die nachfolgend erläutert werden.

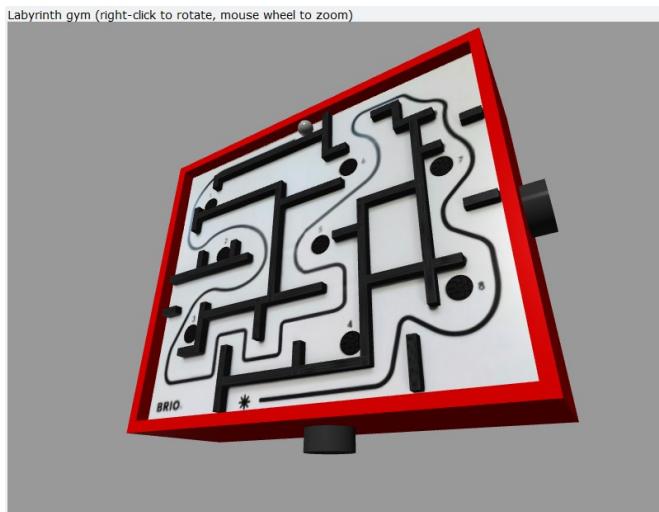
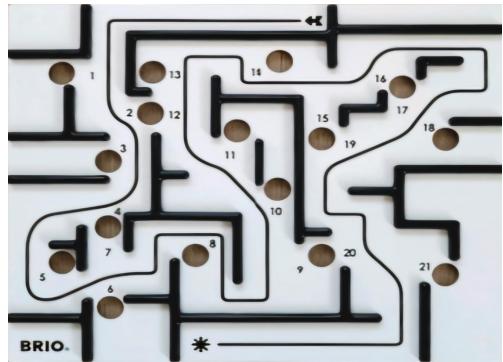


Abbildung 6.2: 3D-Simulation

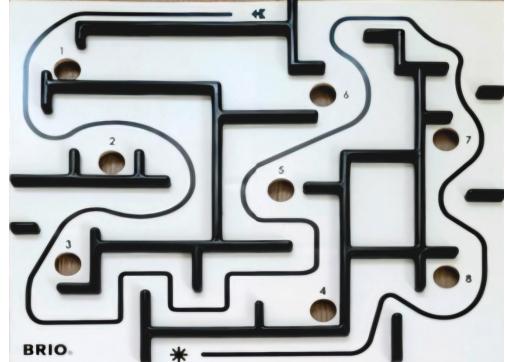
### 6.2.1 Layouts und Geometrie

In der Pythondatei LabLayouts sind die verschiedenen Spielplatten bzw. Layouts und deren geometrischen Abmessungen definiert. Dabei sind die einzelnen geometrischen Abmessungen ohne Einheit definiert. Diese werden aber als Angaben in cm interpretiert. Das detaillierte Klassendiagramm ist in Abbildung 6.4 dargestellt. Die geometrischen Zusammenhänge sind in mehrere Klassen unterteilt, welche in der Klasse LabyrinthGeometry zusammengeführt werden. Die einzelnen Klassen werden nach folgenden Objekten unterschieden: Gehäuse (Box), Spielfeld (Field), Wände (Walls), Löcher (Holes) und Kugel (Ball). Die Abmessungen des Gehäuses, des Spielfeldes und der Kugel wurden messtechnisch anhand des realen BRIO Labyrinths ermittelt. Um die Maße der einzelnen Wände und Löcher zu bestimmen, wurden die realen Spielplatten abfotografiert. Anschließend wurden diese Bilder mit Hilfe von Bildbearbeitung fluchtend ausgerichtet. Anhand dieser Bilder wurden die Abmessungen mit der Linealfunktion einer PDF Software bestimmt. Da diese Bilder nicht das richtige Seitenverhältnis, Größe und den gleichen Koordinatenursprung haben, wurden die einzelnen Bildmaße mit Umrechnungsfaktoren skaliert. Für die Skalierung der Größe der x-Achse  $s_x$  wurde der Umrechnungsfaktor wie folgt bestimmt:

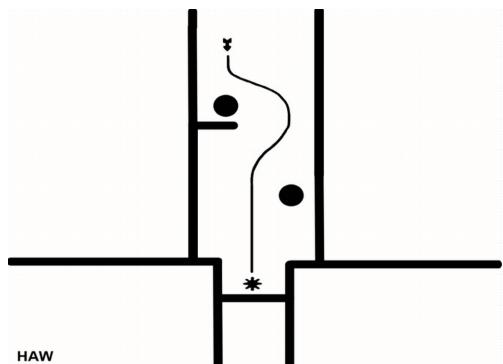
$$s_x = \frac{x_{rg}}{x_{bg}} \quad (6.1)$$



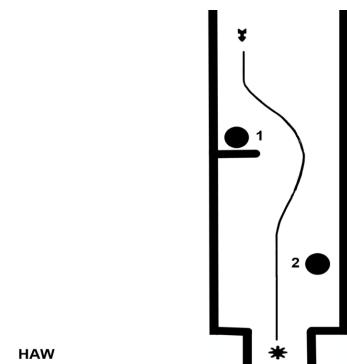
(a)



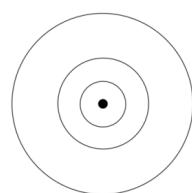
(b)



(c)

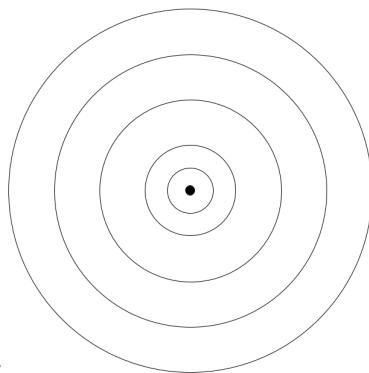


(d)



HAW

(e)



(f)

Abbildung 6.3: Labyrinthe und Spielplatten

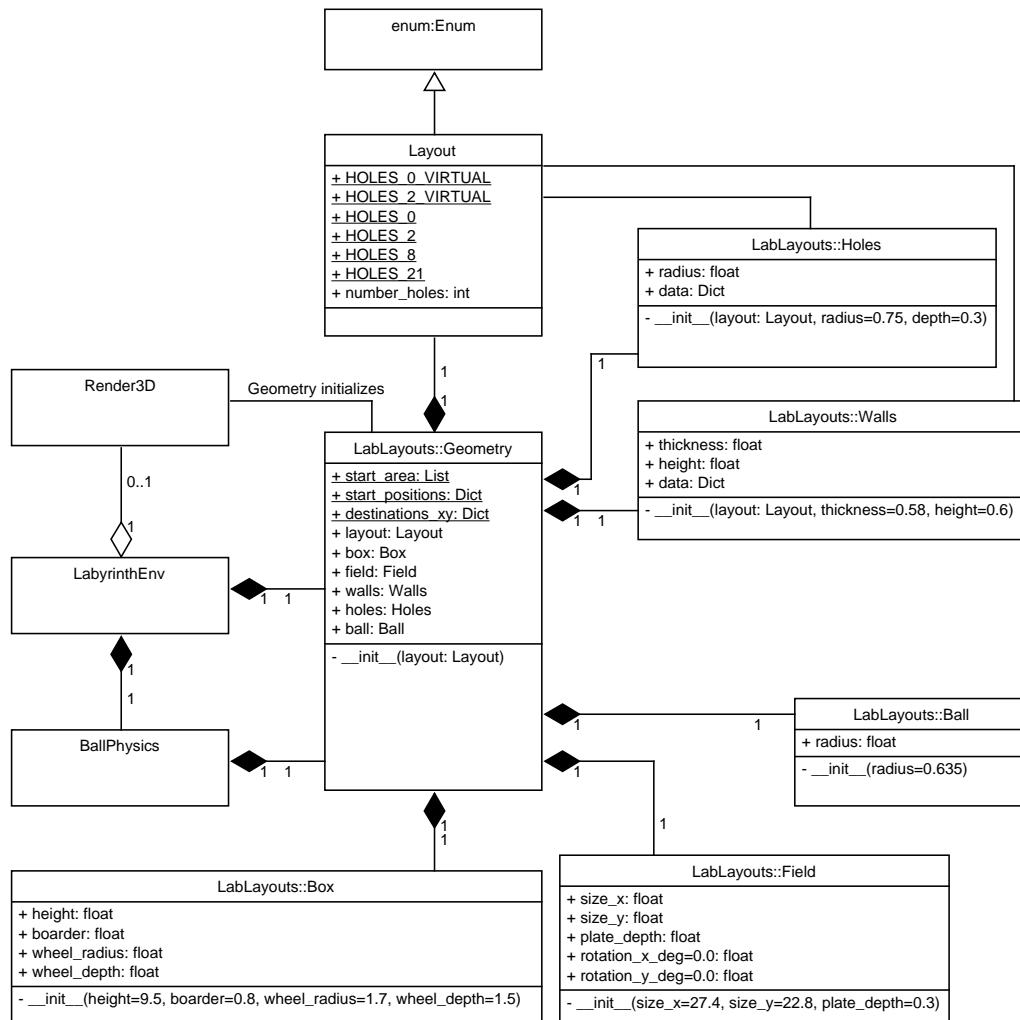


Abbildung 6.4: Klassendiagramm der Layouts und geometrischen Abmessungen

Dabei beschreibt  $x_{bg}$  die aus dem Bild (Index  $b$ ) bestimmte Gesamtlänge der Spielplatte in x-Richtung und  $x_{rg}$  die reale Gesamtlänge. Die Skalierung der Größe der y-Achse  $s_y$  wurde nach dem gleichen Prinzip berechnet. Der Koordinatenursprung der 3D-Simulation liegt in der Mitte des Spielfeldes. Um das Abmessen im Bild zu vereinfachen, wurde dessen Koordinatenursprung in die linke obere Ecke gelegt. Die Umrechnung erfolgte daraufhin für die x-Komponente wie folgt:

$$x = x_b \cdot s_x - \frac{x_{rg}}{2} \quad (6.2)$$

Für die y-Komponente ergibt sich:

$$y = -y_b \cdot s_y + \frac{y_{rg}}{2} \quad (6.3)$$

Die Abmessungen der designten Spielplatten wurden auf gleiche Weise bestimmt. Die erstellte Excel-Datei mit den gesamten Abmessungen, wie auch den bestimmten Start- und Zielpositionen ist auf der CD einsehbar.

### 6.2.2 Render3D

Die Klasse Render3D, realisiert die 3D-Darstellung der einzelnen Objekte. Das konkrete Klassendiagramm ist in Abbildung 6.5 visualisiert. Zu Beginn muss eine *scene* definiert werden. Diese repräsentiert das Fenster, in dem die 3D-Darstellung der Objekte stattfindet. Dabei handelt es sich um eine vordefinierte Instanz der Klasse canvas (Leinwand), welche Methoden zur Verfügung stellt um die Darstellung anzupassen oder mit der 3D-Umgebung zu interagieren. Die *scene* besitzt verschiedenste Einstellmöglichkeiten, wie beispielsweise Breite, Höhe, Hintergrundfarbe, Lichtquelle, Titel und vieles mehr. Für die 3D-Objekte werden in dieser Arbeit vordefinierte Formen verwendet. Das Gehäuse, das Spielfeld und die Wände sind als *Box* definiert. Die realen Spielfeldwände des BRIO Labyrinths sind minimal abgerundet. Dies wird in der Simulation allerdings vernachlässigt, um auf eine vordefinierte Formen zurückgreifen zu können. Der Koordinatenursprung der gesamten 3D-Objekte liegt zentriert auf der Oberseite des Spielfeldes (siehe Abbildung 6.6). Das Box-Objekt ist durch zwei Vektoren definiert, seine Position im Objektmittelpunkt (x, y, z) und seine Größe (Länge, Höhe, Breite). In Abbildung 6.7 ist das Koordinatensystem schematisch für die Spielplatte und die Größenbeschreibung für die obere Außenwand dargestellt. Die Breite ist bei der dargestellten Draufsicht senkrecht zum Blatt definiert. Als weiterer Parameter der Box kann beispielsweise eine Farbe

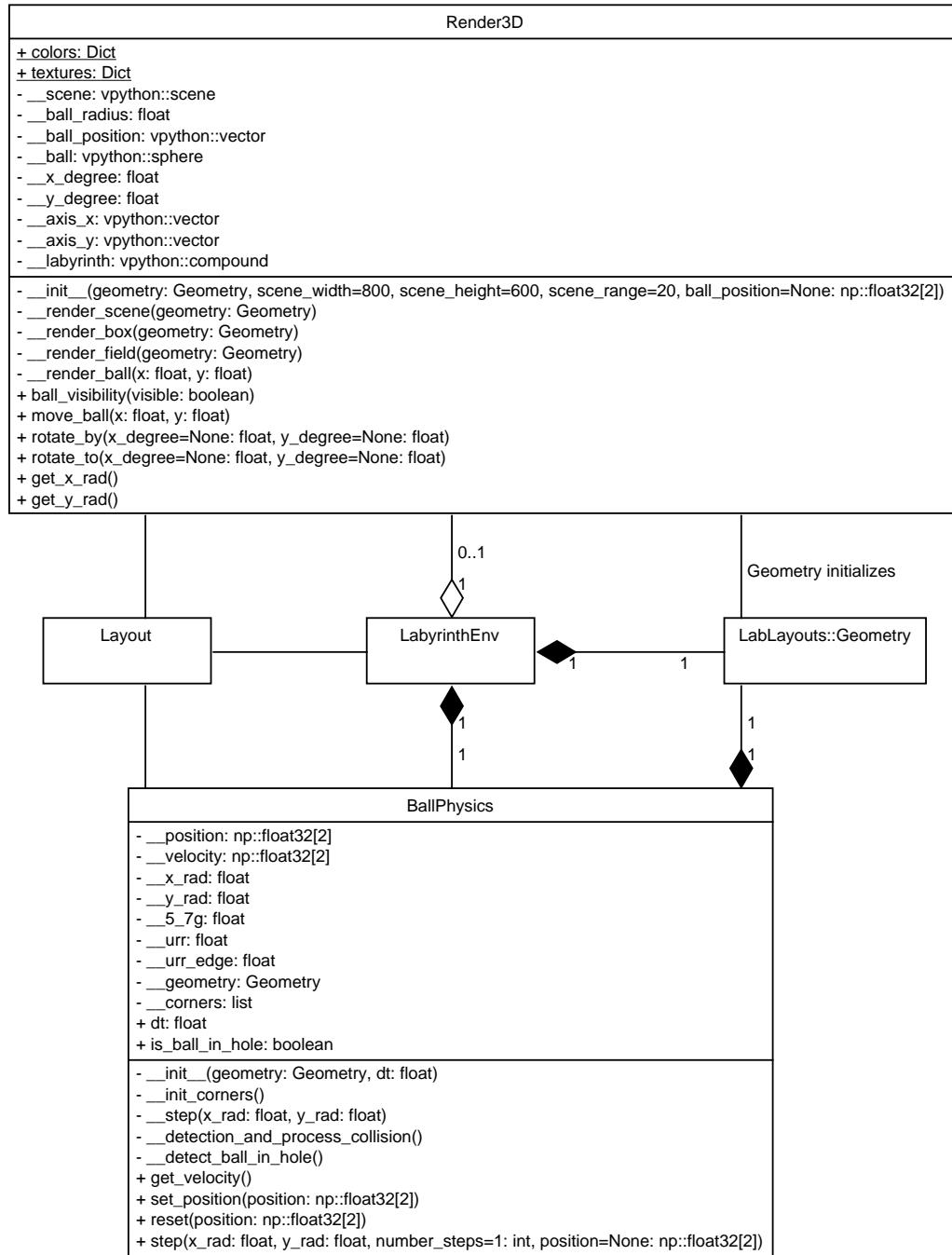


Abbildung 6.5: Klassendiagramm Render3D und Ballphysik

## 6 Entwicklung der virtuellen Umgebung

---

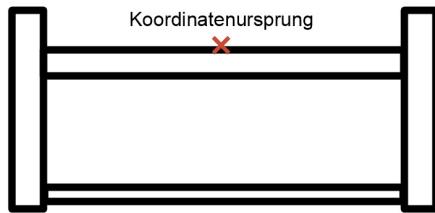


Abbildung 6.6: Koordinatenursprung

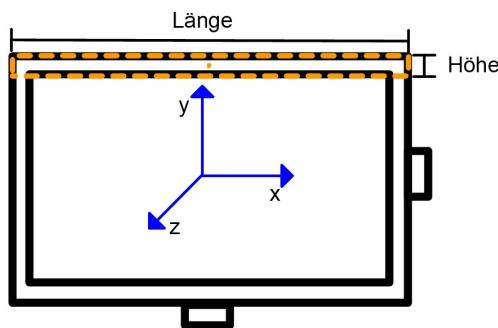


Abbildung 6.7: Box Objekt

hinterlegt werden. Die Drehräder des Spiels, sowie die Löcher sind als *cylinder*-Objekte definiert. Als letzte Komponente wird die Kugel durch ein *sphere*-Objekt repräsentiert. Die Spielplatte, welche rotiert werden soll, besteht aus vielen einzelnen Objekten. Diese können mit Hilfe der *compound*-Funktion zu einem gesamten Objekt zusammengefasst werden. Dadurch muss nur noch dieses zusammengefasste labyrinth-Objekt als Ganzes rotiert werden und nicht jedes einzelne Objekt. Um die Weglinie oder Ziel- und Startsymbole des Labyrinthes auf die Spielplatten zu bringen, können Texturen verwendet werden. Mit Hilfe des *texture*-Parameters können Bilder (eigene oder vordefinierte) als Hintergrund hinterlegt werden. Die Kugel erhält somit auch seine metallische Optik. Für die zuvor angesprochene Drehung der Spielplatte wird die *rotate*-Funktion verwendet. Dabei verwendete Einstellparameter sind der Mittelpunkt (origin), um welchen rotiert werden soll, die Achse (axis), die rotiert werden soll, sowie der konkrete Winkel (angle), auf den rotiert werden soll. Diese *rotate*-Funktion wird zudem für die 3D Raumdarstellung der Kugel verwendet. Dadurch kann die gesamte Kugelbewegung im zweidimensionalen Raum (x- und y-Komponente) berechnet werden (siehe Genauereres in Kapitel 6.3). Um das Hineinfallen der Kugel in Löcher zu simulieren wird in diesem Fall die Kugel unsichtbar gemacht. Dafür kann die *visible*-Funktion verwendet werden.

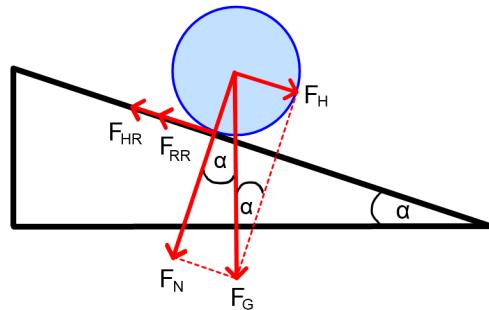


Abbildung 6.8: Kräfte einer rollenden Kugel an einer schießen Ebene

### 6.3 Physikalische Eigenschaften der Kugelbewegung

Um das BRIO Labyrinth präzise simulieren zu können müssen die physikalischen Eigenschaften und Gesetzmäßigkeiten simuliert werden. Nur so lässt sich die angestrebte Vergleichbarkeit zwischen dem virtuellen Modell und der realen Umgebung erreichen. Dies soll es ermöglichen die Parameter aus den simulativen Trainings des Agenten auf den physischen Demonstrator zu übernehmen. Die nachfolgend beschriebenen physikalischen Gesetzmäßigkeiten sind in der Pythondatei *LabBallPhysics* programmtechnisch umgesetzt. Das Klassendiagramm ist in Abbildung 6.5 visualisiert.

#### 6.3.1 Verhalten einer rollenden Kugel auf einer schießen Ebene

Die Bewegungsgleichungen einer rollenden Kugel auf einer schießen Ebene können über die Betrachtung der Kräfte beschrieben werden. Bei der Beschreibung des Verhaltens einer rollenden Kugel sind die in der Abbildung 6.8 dargestellten Kräfte zu berücksichtigen.

##### Gewichts- und Hangabtriebskraft

Da sich die Kugel im Schwerkraftfeld der Erde befindet, wirkt die Gewichtskraft  $F_G$  [33]. Die Gewichtskraft kann wie folgt berechnet werden:

$$F_G = m \cdot g \quad (6.4)$$

Hierbei entspricht  $g$  der Erdbeschleunigung ( $9,81 \text{ m/s}^2$ ) und  $m$  der Masse der Kugel. Die Hangabtriebskraft  $F_H$  [33] bewirkt die Beschleunigung der Kugel auf der Ebene.

Sie wird beschrieben durch

$$F_H = F_G \cdot \sin(\alpha) , \quad (6.5)$$

wobei  $\alpha$  den Neigungswinkel der Ebene beschreibt.

### **Rollreibung**

Die Rollreibung  $F_{RR}$  [33] beschreibt hingegen die Arbeit, die beim Abrollen geleistet wird. Der Kugel wird hierbei mechanische Energie entzogen, die in Wärmeenergie umgewandelt wird.

$$F_{RR} = F_N \cdot \mu_{RR} \quad (6.6)$$

Bei der Berechnung der Rollreibung beschreibt  $\mu_{RR}$  die Rollreibungszahl. Sie ist von den Materialien der Kugel (Stahl) und des Untergrundes (Holzgemisch mit einer Lackierung) abhängig. Die Rollreibungszahl wurde zuerst angenähert und später experimentell genauer bestimmt. Das genaue Vorgehen wird im nachfolgenden Unterkapitel weiter erläutert.  $F_N$  beschreibt dabei die Normalkraft [33], die dafür sorgt, dass sich die beiden Materialien (Kugel und Untergrund) beim Abrollen leicht verformen. Sie wird durch

$$F_N = F_G \cdot \cos(\alpha) \quad (6.7)$$

beschrieben.

### **Haftreibungskraft**

Außerdem ist noch die Haftreibungskraft  $F_{HR}$  [18] zu berücksichtigen. Sie sorgt dafür, dass die Kugel in eine Drehbewegung (Rotation) kommt. Die Haftreibung wirkt der Hangabtriebskraft entgegen. Sie bewirkt ein Drehmoment  $M$  auf die Kugel mit dem Radius  $r$ . Mit Hilfe des Drehmomentes

$$M = F_{HR} \cdot r = J \cdot \dot{\omega} \quad (6.8)$$

ergibt sich für die Haftreibung

$$F_{HR} = \frac{J \cdot \dot{\omega}}{r} . \quad (6.9)$$

Dabei beschreibt  $J$  das Trägheitsmoment und  $\dot{\omega}$  die Winkelbeschleunigung. Nachfolgend ist  $\ddot{v}$  die wirksame Beschleunigung auf die Kugel, welche auch mit  $a$  bezeichnet wird. Setzt man zusätzlich die Winkelbeschleunigung

$$\dot{\omega} = \frac{\ddot{v}}{r} \quad (6.10)$$

und das Trägheitsmoment einer Kugel

$$J = \frac{2}{5} \cdot m \cdot r^2 \quad (6.11)$$

ein, erhält man schlussendlich für die Haftreibung  $F_{HR}$ :

$$F_{HR} = \frac{2}{5} \cdot m \cdot \frac{r^2}{r^2} \cdot \dot{v} = \frac{2 \cdot m \cdot \dot{v}}{5} \quad (6.12)$$

### Bestimmung der Geschwindigkeit und Position der Kugel

Die resultierende Kraft in Vorwärtsrichtung  $F_{vor}$  für eine rollende Kugel setzt sich wie folgt zusammen [18, 49]:

$$F_{vor} = m \cdot a = m \cdot \dot{v} = F_H - F_{HR} - F_{RR} \quad (6.13)$$

Nach Einsetzen der einzelnen Kräfte

$$F_{vor} = m \cdot g \cdot \sin(\alpha) - \frac{2 \cdot m \cdot \dot{v}}{5} - \mu_{RR} \cdot m \cdot g \cdot \cos(\alpha) = m \cdot a \quad (6.14)$$

und Division durch  $m$

$$g \cdot (\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) - \frac{2}{5} \cdot a = a \quad (6.15)$$

bzw.

$$g \cdot (\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) = \frac{7}{5} \cdot a \quad (6.16)$$

ergibt sich die Beschleunigung der Kugel zu:

$$a = \frac{5}{7} \cdot g \cdot (\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) \quad (6.17)$$

Ausgehend von einer Geschwindigkeit  $v_1$  zum Zeitpunkt  $t_1$  kann die Geschwindigkeit  $v_2$  zum Zeitpunkt  $t_2 > t_1$  mit  $\Delta T = t_2 - t_1$  wie folgt bestimmt werden [52]:

$$v_2 = v_1 + \frac{5}{7} \cdot g \cdot (\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) \cdot (t_2 - t_1) = v_1 + a \cdot \Delta T \quad (6.18)$$

Die Geschwindigkeit wird für beide Richtungskomponenten  $x$  und  $y$  bestimmt, da das Spielfeld um diese zwei Achsen gedreht werden kann. Daraus ergibt sich dann der fol-

gende Geschwindigkeitsvektor der Kugel:

$$\mathbf{v} = \begin{pmatrix} v_x \\ v_y \end{pmatrix} \quad (6.19)$$

Die neue Position kann anhand des Zusammenhangs für gleichmäßig beschleunigte Bewegungen ( $\mathbf{a}$  = konstant) über

$$\mathbf{s}(t_2) = \mathbf{s}(t_1) + \mathbf{v}_1 \cdot \Delta T + \frac{\mathbf{a}}{2} \cdot \Delta T^2 \quad (6.20)$$

bestimmt werden [33]. Auch hierbei sind wieder die Komponenten  $x$  und  $y$  zu betrachten.

### 6.3.2 Bestimmung der Rollreibungszahl

Die Rollreibungszahl ist materialspezifisch. Zu Beginn wurde diese Konstante durch bekannte Rollreibungszahlen angenähert und anschließend in einem Experiment überprüft. Ein Fahrradreifen auf einer asphaltierten Straße hat eine Reibungszahl zwischen 0,004 bis 0,002 [73]. Da die Straße und der Fahrradreifen eine rauere Oberfläche besitzen, muss die hier verwendete Reibungszahl etwas niedriger sein. Bei Wälzlagern liegt die Rollreibungszahl zwischen 0,0013 (Pendelkugellager) und 0,0025 (Nadellager) [73]. Dieser Wertebereich erscheint als Näherung recht sinnvoll. In einem anschließend durchgeführten Experiment wurde eine Rollreibungszahl von 0,00118 ermittelt, was zu der vorherigen Annäherung sehr gut passt.

Der prinzipielle Aufbau zur Bestimmung des Koeffizienten ist in der Abbildung 6.9 dargestellt. Die Kugel wurde hierzu derart auf die Startposition, den Pfeil des Spielfeldes gelegt, dass diese keine Wand berührt. Die Ausgangsposition wurde als  $s_1 = 0$  cm festgelegt und es wurde mit einer Geschwindigkeit  $v_1 = 0$  m/s gestartet. Anschließend wurde mit dem automatisierten Spiel (die genaue Umsetzung der Automatisierung ist in dem Kapitel 7 erläutert) ein Winkel von  $\alpha = 2,1^\circ$  eingestellt, so dass die Kugel die Grade entlang der Linie herunter rollt. Das Verhältnis der erforderlichen Pulsweite für die Motoransteuerung zum Spielplattenwinkel wurde messtechnisch bestimmt (siehe Kapitel 7.2.2). Die Strecke bis die Kugel unten an die Wand trifft betrug  $s_2 = 11,45$  cm. Das Experiment wurde ein paarmal durchgeführt und mit einer Kamera aufgenommen. Zwei exemplarische Videoaufnahmen des Experimentes, die zur Auswertung herangezogen wurden, sind auf der CD einzusehen. Anschließend konnte die Zeit vom Beginn

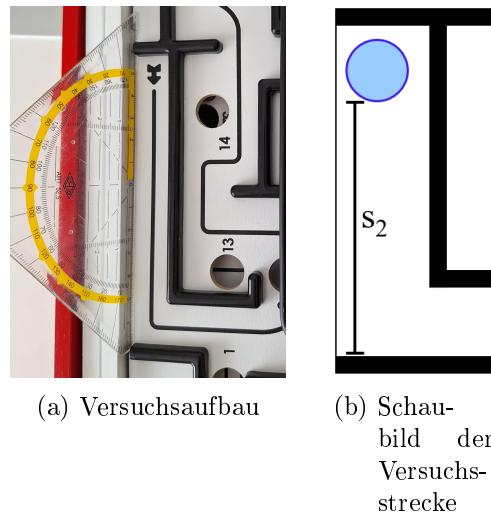


Abbildung 6.9: Aufbau zur Bestimmung der materialspezifischen Koeffizienten

des Losrollens bis zur Kollision mit der Wand genau bestimmt werden. Die Kugel hat hierbei eine Zeit von  $\Delta T = 0,96$  s für die Strecke  $s_2$  gebraucht. Das Spielfeld in der Abbildung 6.9 (a) ist im Vergleich zu der roten Box leicht schräg bei der Nulllage. Das Spiel hatte zu dem Zeitpunkt der Messung nur die zwei Halterungen für die Servos, die unter die Box geklemmt werden, und besaß noch keine weiteren Füße. Somit stand die Box außen herum leicht schief (genaueres ist dem Kapitel 7 zu entnehmen). Mit Hilfe der Formeln 6.20 und 6.17 und Einsetzen von  $s_1$  und  $v_1$  ergibt sich für die Strecke folgende Formel:

$$s_2 = \frac{5}{2} \cdot 7 \cdot g \cdot (\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) \cdot \Delta T^2 \quad (6.21)$$

Nach Umformung kann die Rollreibungszahl  $\mu_{RR}$  folgendermaßen berechnet werden:

$$\mu_{RR} = -\frac{s_2 \cdot 2 \cdot 7}{\Delta T^2 \cdot 5 \cdot g \cdot \cos(\alpha)} + \frac{\sin(\alpha)}{\cos(\alpha)} \quad (6.22)$$

Hierbei kann  $\frac{\sin(\alpha)}{\cos(\alpha)}$  auch durch  $\tan(\alpha)$  ersetzt werden. Durch Einsetzen der oben angegebenen Parameter ergibt sich somit ein  $\mu_{RR}$  von 0,00118.

### 6.3.3 Kollisionsdetektion zwischen Kugel und Wand

Die rechteckigen Wände des BRIO Labyrinthes sind lediglich senkrecht oder waagerecht platziert. Deshalb kann die Kollisionsdetektion zwischen der Kugel und der Wand auf

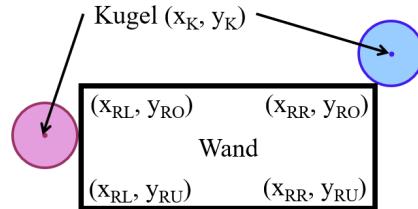


Abbildung 6.10: Kollision zwischen Kugel und Wand

eine einfache Kollisionserkennung zwischen einem Kreis (Kugel) und einem achsenparallelen Rechteck (Wand) zurückgeführt werden. Hierbei ist es relevant zwei Fälle zu unterscheiden, die Kollision mit einer Ecke und mit einer Kante. In Abbildung 6.10 wurden zur Unterscheidung der einzelnen Punkte Indizes eingeführt. Hierbei bezeichnet  $K$  den Kreis und  $R$  das Rechteck mit den Indizes für die Eckkoordinaten Rechts  $R$ , Links  $L$ , Oben  $O$ , Unten  $U$ .

Die Kollisionsbestimmung mit einer Ecke kann auf den Satz des Pythagoras  $a^2 + b^2 = c^2$  zurückgeführt werden. Am Beispiel der blauen Kugel liegt somit eine **Kollision mit einer Ecke** bei folgender Bedingung vor:

$$(x_K - x_{RR})^2 + (y_K - y_{RO})^2 \leq r^2 \quad (6.23)$$

Die **Kollision mit einer Kanten**, hier am Beispiel der lila Kugel liegt vor, wenn nachfolgende Bedingungen erfüllt sind:

$$(x_{RL} - x_K \leq r \wedge x_K \leq x_{RL}) \wedge (y_K \geq y_{RU} \wedge y_K \leq y_{RO}) \quad (6.24)$$

#### 6.3.4 Verhalten der Kugel nach der Kollision mit einer Kante

Es gibt zwei Arten von Kollisionen bzw. Stößen [32], den inelastischen und den elastischen Stoß. Die beteiligten Objekte bei einem inelastischen Stoß verlieren in der Summe an kinetischer Energie, beispielsweise durch Deformation, Wärme oder Verluste durch Reibung. Ein typisches Beispiel für einen inelastischen Stoß ist die Kollision von Autos. Ein Spezialfall des inelastischen Stoßes ist der unelastische Stoß. Hierbei verkoppeln sich die Stoßpartner miteinander und bewegen sich anschließend mit einer gleichen, gemeinsamen Endgeschwindigkeit. Bei einem elastischen Stoß bleibt die Summe der kinetischen Energie der beteiligten Objekte konstant, es gibt keine Energieverluste bei der Kollision. Als Beispiel für einen elastischen Stoß wird oft die Kollision von Billardkugeln

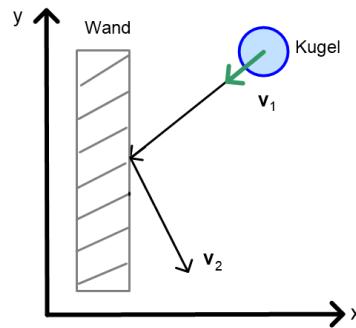


Abbildung 6.11: Kugelverhalten nach einer Kollision eines inelastischen Stoßes

angeführt. Die Kollision der Kugel mit der Wand wurde zu Beginn vereinfacht als elastische Kollision betrachtet. Diese anfängliche Betrachtung wurde im Laufe der ersten Simulationen zu einer inelastischen Kollision erweitert um ein realistischeres Verhalten der Kugel zu simulieren. Die Verhaltensunterschiede bei einer Kollision der Kugel mit der Wand waren zwischen dem realen Spiel und der rein elastischen Betrachtung sehr deutlich erkennbar.

### Geschwindigkeitsanpassung

Für die Behandlung der Kollision wird der Geschwindigkeitsvektor in die vektoriellen Richtungskomponenten x und y aufgeteilt.

$$\mathbf{v}_1 = \begin{pmatrix} v_{x1} \\ v_{y1} \end{pmatrix} \quad (6.25)$$

Bei einem rein elastischen Stoß gilt der Impulserhaltungssatz. Hierbei würde die Geschwindigkeit gleich bleiben. Die Geschwindigkeitsrichtungen stehen dann nach dem Stoß senkrecht aufeinander. Durch die Erweiterung zu einem inelastischen Stoß wird noch eine Stoßzahl  $k$  [32], auch Restitutionskoeffizient genannt, mit berücksichtigt. Eine Stoßzahl von 1 beschreibt dabei den ideal elastischen Stoß (maximaler Rückprall ohne Energieverlust) und eine Stoßzahl von 0 den ideal inelastischen Stoß (maximale Energieübertragung ohne Rückprall). Die genaue Bestimmung der Stoßzahl wird im nachfolgenden Unterkapitel beschrieben. Da die Wand starr ist, ergibt sich für die Kugel bei der Kollision mit einer vertikalen Wand (siehe Abbildung 6.11)

$$v_{x2} = -v_{x1} \cdot k \quad (6.26)$$

und bei einer horizontalen Wand dementsprechend

$$v_{y2} = -v_{y1} \cdot k . \quad (6.27)$$

Wenn die Kugel an einer Wand entlang rollt, wird auch die andere Geschwindigkeitskomponente angepasst. Beim Entlangrollen an der Wand entsteht weitere Reibung, die zur Rollreibung im Zusammenhang mit dem Spielfelduntergrund hinzu kommt. Am Beispiel der lila Kugel aus der Abbildung 6.10, die an einer vertikalen Wand hinunter rollt, wäre somit die y-Komponente zusätzlich zur x-Komponente anzupassen. Dabei wird das zuvor berechnete Verhalten der Kugel auf der schießen Ebene durch ein zusätzlichen Reibungsanteil erweitert. Die angepasste Geschwindigkeit wird dann wie folgt bestimmt

$$v_{y2} = v_{y2,ohneWand} + \Delta a_{Wand} \cdot \Delta T \quad (6.28)$$

mit

$$\Delta a_{Wand} = \frac{5}{7} \cdot g \cdot (-\Delta \mu_{RR} \cdot \cos(\alpha)) . \quad (6.29)$$

Dabei entspricht  $\Delta T$  dem Zeitintervall der Neuberechnung der Geschwindigkeiten und Positionen. Für die Bestimmung von  $\Delta \mu_{RR}$  wurde das Experiment zur Bestimmung der Rollreibungszahl (siehe Kapitel 6.3.2) einige Male wiederholt, mit dem Unterschied, dass die Kugel direkt an der Wand platziert wurde und entlang dieser runterrollt. Zwei exemplarische Videoaufnahmen des Experimentes, die zur Auswertung herangezogen wurden, sind auf der CD einzusehen. Es ergab sich eine benötigte Durchschnittszeit von 0,99 s für die selbe Strecke  $s_2 = 11,45$  cm, ohne Wand waren es 0,96 s. Mit der Wand ergab sich unter Anwendung der Formel 6.22 ein  $\mu_{RR}$  von 0,0033. Somit wird ein  $\Delta \mu_{RR} = \mu_{RR,mitWand} - \mu_{RR,ohneWand}$ , was einem Wert von 0,00212 entspricht, noch zusätzlich berücksichtigt.

### Positionsanpassung

Da zwischen den einzelnen Positionsneuberechnungen kleine Zeitintervalle liegen, kommt es dazu, dass der Ball einen Hauch in Wände eintauchen würde, wenn eine Kollision detektiert wird. Um das Eintauchen in die Wände zu verhindern, wird die neue Position angepasst und die Kugel an die Wandkante außerhalb der Wand geschoben. Am Beispiel der in Abbildung 6.10 dargestellten lila Kugel wird somit die neue Position  $x_{Kneu}$  wie folgt bei einer erkannten Kollision mit einer Kante angepasst:

$$x_{Kneu} = x_{RL} - r_K \quad (6.30)$$

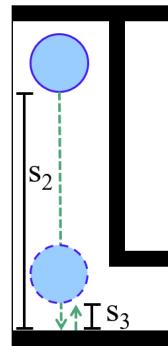


Abbildung 6.12: Schaubild zur Bestimmung der Stoßzahl

Dabei entspricht  $x_{RL}$  der x-Position der Kante und  $r_K$  dem Radius der Kugel.

### 6.3.5 Bestimmung der Stoßzahl

Für die genaue Bestimmung der Stoßzahl  $k$  wurde das gleiche Experiment durchgeführt wie schon zur Bestimmung der Rollreibungszahl. Der Spielfeldneigungswinkel wurde hierbei aber auf  $\alpha = 2,4^\circ$  erhöht, um einen höheren Abprall von der Wand zu erreichen. Somit ist der Aufbau der selbe wie im Unterkapitel 6.3.2 beschrieben und in Abbildung 6.9 (a) dargestellt. Das entsprechende Schaubild des Experiments ist in Abbildung 6.12 dargestellt. Die Kugel wird auf der Startposition ohne Wandkontakt platziert. Durch die Automatisierung des Spiels wird anschließend der entsprechende Winkel über die Servosteuerung eingestellt. Die Startposition ist mit  $s_1 = 0$  cm festgelegt und die Anfangsgeschwindigkeit beträgt  $v_1 = 0$  m/s. Die zurückgelegte Strecke der Kugel bis zur Wand entspricht wieder  $s_2 = 11,45$  cm, in der Zeit  $\Delta T_2 = 0,96$  s. Die Aufprallgeschwindigkeit entspricht  $v_2$ . Mit der Kamera wird ein Video von dem Experiment gemacht, so dass anschließend daraus die Entfernung des Rückpralls  $s_3 = 15$  mm anhand des platzierten Geodreiecks abgelesen werden kann. Die Geschwindigkeit an diesem höchsten Punkt des Rückpralls liegt bei  $v_{3,Ende} = 0$  m/s. Auch dieses Experiment wurde ein Paar mal durchgeführt. Zwei exemplarische Videoaufnahmen des Experiments, die zur Auswertung herangezogen wurden, sind auf der CD einzusehen. Die Strecke  $s_3$  errechnet sich dabei wie folgt

$$s_3 = v_{3,Anfang} \cdot \Delta T_3 + \frac{a_3}{2} \cdot \Delta T_3^2 \quad (6.31)$$

mit

$$v_{3,Anfang} = v_2 \cdot k = a_2 \cdot \Delta T_2 \cdot k \quad (6.32)$$

$$a_2 = \frac{5}{7} \cdot g \cdot (\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) \quad (6.33)$$

$$a_3 = \frac{5}{7} \cdot g \cdot (-\sin(\alpha) - \mu_{RR} \cdot \cos(\alpha)) \quad (6.34)$$

$$\Delta T_3 = \frac{v_{3,Ende} - v_{3,Anfang}}{a_3} = \frac{0 - v_2 \cdot k}{a_3} \quad (6.35)$$

Der Vorzeichenwechsel in den Gleichungen 6.33 und 6.34 ist durch den Richtungswechsel der Geschwindigkeiten begründet. Durch Einsetzen der Parameter

$$s_3 = v_2 \cdot k \cdot \frac{-k \cdot v_2}{a_3} + \frac{a_3}{2} \cdot \frac{-k \cdot v_2}{a_3} \cdot \frac{-k \cdot v_2}{a_3} \quad (6.36)$$

und Zusammenfassen der Formel ergibt sich

$$s_3 = \frac{(k \cdot v_2)^2}{a_3} \cdot \left(-1 + \frac{1}{2}\right). \quad (6.37)$$

Diese Formel kann nun nach k umgestellt werden:

$$k = \sqrt{\frac{-s_3 \cdot 2 \cdot a_3}{v_2^2}} = \sqrt{\frac{-s_3 \cdot 2 \cdot a_3}{(a_2 \cdot \Delta T_2)^2}} \quad (6.38)$$

Nach dem Einsetzen aller Zahlenwerte in die Gleichung erhält man eine Stoßzahl von  $k = 0,1099$ .

### 6.3.6 Verhalten der Kugel nach der Kollision mit einer Ecke

Bei der Kollision mit einer Ecke muss das Verhalten von dezentralen Stößen berücksichtigt werden. Bei dezentralen Stößen [10] liegt die Geschwindigkeitsrichtung nicht parallel zur Verbindungsgeraden der Stoßkörper. Ein Beispiel für den dezentralen Stoß ist in der Abbildung 6.13 dargestellt.

#### Geschwindigkeitsanpassung

Für die Berechnung der Geschwindigkeitsanpassung wird eine Hilfsebene (siehe in Abbildung 6.13 (a) die Kollisionsebene) eingeführt. Der Vektor  $\mathbf{e}_{Kollision}$  liegt auf der Verbindungsgeraden zwischen Mittelpunkt der Kugel und dem Stoßpunkt mit der Wandcke. Die Hilfsebene liegt senkrecht dazu. Der Vektor  $\mathbf{e}_{Kollision}$  ist ein Einheitsvektor der

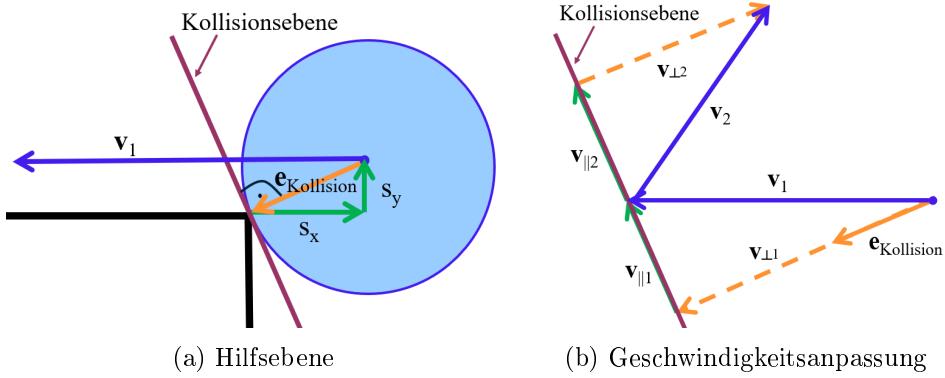


Abbildung 6.13: Geschwindigkeitsanpassung bei Ecken Kollision

Länge 1. Er kann wie folgt aus den einzelnen Komponenten berechnet werden:

$$\mathbf{e}_{Kollision} = \begin{pmatrix} \frac{-s_x}{\sqrt{s_x^2+s_y^2}} \\ \frac{s_y}{\sqrt{s_x^2+s_y^2}} \end{pmatrix} = \begin{pmatrix} e_x \\ e_y \end{pmatrix} \quad (6.39)$$

Die Strecken  $s_x$  und  $s_y$  berechnen sich dabei aus der Differenz der Ballposition und der Eckenposition.  $\mathbf{v}_1$  beschreibt den bekannten Geschwindigkeitsvektor der Kugel vor der Kollision. Für die Berechnung der Geschwindigkeit nach der Kollision wird der in der Abbildung 6.13 (b) dargestellte Zusammenhang herangezogen.  $\mathbf{v}_1$  ist somit durch die senkrechte und parallele Komponente definiert:

$$\mathbf{v}_1 = \mathbf{v}_{\perp 1} + \mathbf{v}_{\parallel 1}. \quad (6.40)$$

Die senkrechte Komponente kann mit Hilfe des Skalarproduktes wie folgt bestimmt werden:

$$\mathbf{v}_{\perp 1} = (\mathbf{v}_1 \cdot \mathbf{e}_{Kollision}) \cdot \begin{pmatrix} e_x \\ e_y \end{pmatrix} = (v_{1x} \cdot e_x + v_{1y} \cdot e_y) \cdot \begin{pmatrix} e_x \\ e_y \end{pmatrix} \quad (6.41)$$

Nach dem Stoß dreht sich die Geschwindigkeitsrichtung der senkrechten Komponente um, zudem wird hierbei auch die Stoßzahl  $k$  berücksichtigt:

$$\mathbf{v}_{\perp 2} = -\mathbf{v}_{\perp 1} \cdot k \quad (6.42)$$

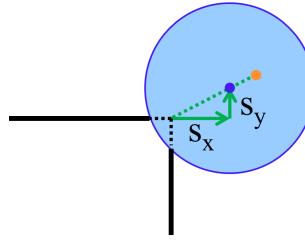


Abbildung 6.14: Positionsanpassung bei Eckenkollision

Da  $\mathbf{v}_1$  und  $\mathbf{v}_{\perp 1}$  bekannt sind kann die parallele Komponente  $\mathbf{v}_{\parallel 1}$  wie folgt bestimmt werden:

$$\mathbf{v}_{\parallel 1} = \mathbf{v}_{\parallel 2} = \mathbf{v}_1 - \mathbf{v}_{\perp 1} \quad (6.43)$$

Mit den nun bekannten senkrechten ( $\mathbf{v}_{\perp 2}$ ) und parallelen Komponenten ( $\mathbf{v}_{\parallel 2}$ ) nach dem Stoß kann der neue Geschwindigkeitsvektor der Kugel  $\mathbf{v}_2$  bestimmt werden:

$$\mathbf{v}_2 = \mathbf{v}_{\perp 2} + \mathbf{v}_{\parallel 2} = -\mathbf{v}_{\perp 1} \cdot k + \mathbf{v}_1 - \mathbf{v}_{\perp 1} = \mathbf{v}_1 - \mathbf{v}_{\perp 1} \cdot (1 + k) \quad (6.44)$$

### Positionsanpassung

Da zwischen den einzelnen Positionsneuberechnungen kleine Zeitintervalle liegen, kommt es dazu, dass der Ball einen Hauch in die Ecken eintauchen würde, wenn eine Kollision detektiert wird. Um das Eintauchen in die Wände zu verhindern, wird die neue Position angepasst und die Kugel direkt an die Wanddecke außerhalb der Wand geschoben. Am Beispiel der blauen Kugel aus der Abbildung 6.14, einer erkannten Kollision der Kugel ( $x_K, y_K$ ) mit der rechten oberen Ecke der Wand ( $x_{RR}, y_{RR}$ ) wird folgende Anpassung gemacht.

$$x_K = s_x \cdot s_f + x_{RR} \quad (6.45)$$

$$y_K = s_y \cdot s_f + y_{RR} \quad (6.46)$$

Der Abstandsfaktor  $s_f$  wird dabei durch

$$s_f = \frac{r}{\sqrt{s_x^2 + s_y^2}} \quad (6.47)$$

bestimmt, wobei  $r$  dem Radius der Kugel entspricht. In Abbildung 6.14 entspricht der blaue Mittelpunkt dem eigentlich berechnetem Wert der Kugelposition und der orangene Mittelpunkt der angepassten Position.

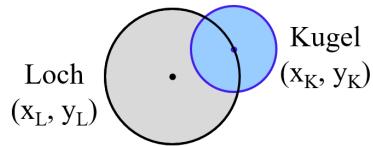


Abbildung 6.15: Kollision zwischen Kugel und Loch

### 6.3.7 Kollisionsdetektion zwischen Kugel und Loch

Der Fall der Kugel in ein Loch kann auf einen angepassten Fall einer Kollision von zwei Kreisen zurückgeführt werden. Die Kugel fällt in ein Loch, wenn der Mittelpunkt der Kugel den Radius des Loches überschreitet. In Abbildung 6.15 ist diese angepasste Kollision aufgezeigt. Die Kugel  $K$  fällt in das Loch  $L$  hinein, wenn folgende Bedingung erfüllt ist:

$$(x_K - x_L)^2 + (y_K - y_L)^2 < r_L^2 \quad (6.48)$$

## 6.4 OpenAI Gym-Environment

Die Pythondatei *LabyrinthEnv* dient als Schnittstelle zwischen der eigens entwickelten Umgebung und dem RL-Agenten. Hierbei wird nach den Konventionen der OpenAI Gym Bibliothek vorgegangen um die Interaktion des Agenten mit der Umgebung zu ermöglicht.

### 6.4.1 Environment Grundlagen

Für das Training eines Agenten in der eigenen Umgebung müssen verschiedene Funktionen implementiert werden<sup>1</sup>. Die vier Grundmethoden werden nachfolgend genauer betrachtet.

- `init()`: Zu Beginn müssen der Zustands- und Aktionsraum definiert werden.
- `reset()`: Diese Methode wird aufgerufen, wenn eine neue Episode gestartet werden soll. Dadurch wird die Umgebung in einen vordefinierten Anfangszustand zurückgesetzt. Die Rückgabe dieser Methode umfasst den Zustandsraum und weitere optionale Informationen.

---

<sup>1</sup>[https://www.gymlibrary.dev/content/environment\\_creation/](https://www.gymlibrary.dev/content/environment_creation/) - Zugriffssdatum: 07.05.2024

- `step()`: Die Aktion wird der Methode als Übergabeparameter mitgegeben. Die ausgewählte Aktion wird innerhalb dieser Methode weiter verarbeitet und ausgeführt. Dies beinhaltet die Berechnung des neuen Zustands nach der Aktion, die Bewertung der Aktion (Belohnung) und die Überprüfung, ob ein Endzustand erreicht wurde. Die Rückgabeparameter umfassen den neuen Zustand, die Belohnung, ob das Spiel beendet oder abgebrochen wurde und wieder eine optionale Information.
- `render()`: Mit dieser Methode wird die grafische Ausgabe der modellierten 3D-Umgebung ermöglicht. Die Visualisierung des aktuellen Zustands der Umgebung erlaubt es, das Verhalten und die Entwicklung des Agenten zu überprüfen.

### 6.4.2 Implementierung

In der Abbildung 6.16 ist das Klassendiagramm der Environment dargestellt. Nachfolgend wird die konkrete Implementierung genauer erläutert.

#### Init-Methode:

Als Übergabeparameter kann eingestellt werden, ob die 3D-Visualisierung angezeigt werden soll (`render_mode='3D'`) oder nicht. Des Weiteren sind zwei Zeitperioden als Standardwerte gesetzt. Mit dem Parameter `actions_dt` wird das Zeitintervall zwischen zwei Aktionen definiert. Er ist initial auf 100 ms gesetzt. Das Zeitintervall `physics_dt` legt die Aktualisierungszeit für die Ballphysikberechnung fest und ist mit 10 ms initialisiert. Letzteres muss klein genug sein, um zu verhindern, dass die Kugel beispielsweise durch Wände rollt. Zu kleine Aktualisierungszeiten würden jedoch das Training aufgrund der vielen Berechnungen verlängern. Das Zeitintervall für die Aktionswahl wurde so gewählt, dass sie auch beim physischen Demonstrator eingehalten werden kann. In der Init-Methode werden allgemeine Initialisierungen vorgenommen wie der Feldorientierung, der Kugel, des 3D-Rendering (falls ausgewählt) sowie des Zustandsraums und Aktionsraums. Der Aktionsraum wird durch diskrete Winkelstellungen beschrieben, wobei pro Achse (x und y) fünf verschiedene Winkelstellungen zur Verfügung stehen. Somit besteht der gesamte Aktionsraum aus zehn Komponenten. Mit größeren Aktionsräumen wurden auch experimentiert, siehe Kapitel 8.1.1.

#### get\_observation-Methode:

Diese Methode liefert den konkreten Zustandsraum bestehend aus sechs Parametern: der aktuellen x- und y-Position der Kugel, der vorangegangenen x- und y-Position der Kugel und der Spielfeldorientierung (Drehwinkel um x- und y-Achse).

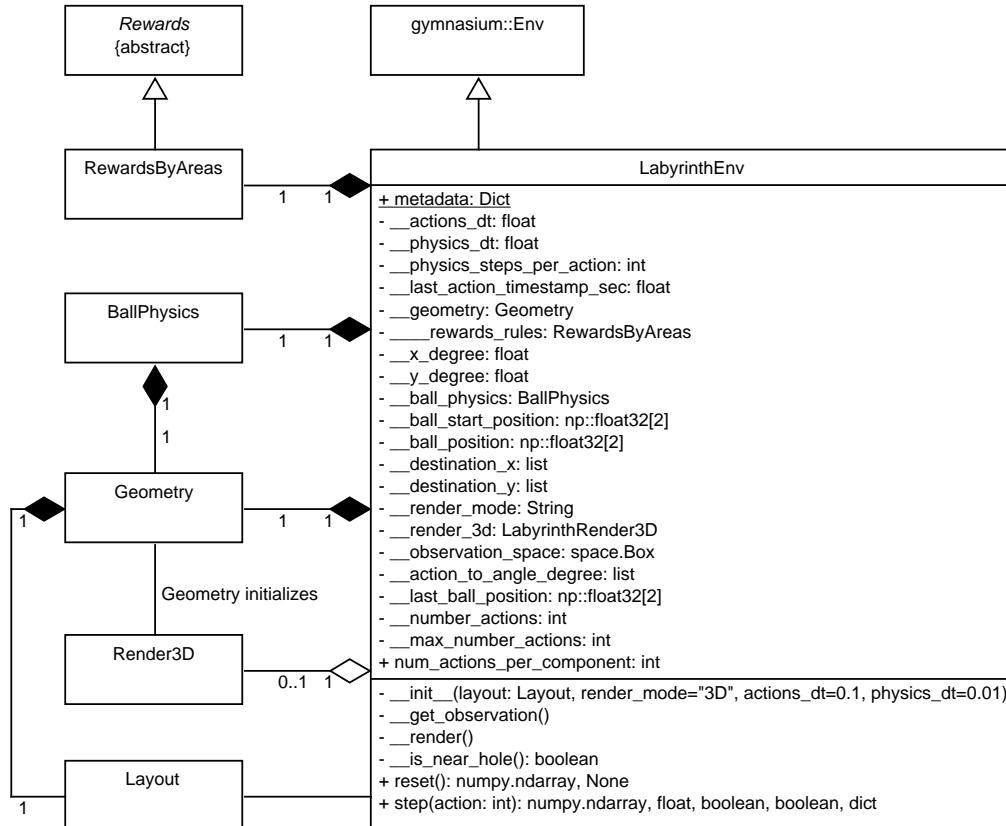


Abbildung 6.16: Klassendiagramm der Labyrinth Environment

### Reset-Methode:

Die Reset-Methode setzt die Umgebung auf einen Anfangszustand zurück. Zunächst wird die Startposition der Kugel in Abhängigkeit vom Layout des Labyrinths gesetzt. Bei der Spielplatte „HOLES\_0“ kann dies beispielsweise eine zufällige Position innerhalb eines größeren Feldes (beispielsweise des gesamten Spielfeldes) sein. Bei der Spielplatte „HOLES\_2“ wird die Kugel in einem kleineren, zufälligen Bereich um den Startpunkt positioniert. Ebenso werden beispielsweise für das Layout „HOLES\_8“ verschiedene vordefinierte Startpunkte zufällig ausgewählt. Dies soll dem Problem der ungleichmäßigen Verteilung der Eingabedaten (siehe Kapitel 2.3.10) beim Training entgegenwirken und eine gleichmäßige Erkundung ermöglichen. Anschließend wird der Zustandsraum, die Spielfeldorientierung und gegebenenfalls das 3D-Rendering zurückgesetzt. Die Rückgabe dieser Methode umfasst den Zustandsraum sowie einen Informationsparameter, der aber derzeit nicht genutzt wird.

### **Render-Methode:**

Die Render-Methode wird im Zeitzyklus der Aktionswahl (100 ms) aufgerufen. Dadurch kann die Kugelbewegung und die Spielfeldbewegung visualisiert werden. Dies ermöglicht es den Trainingsfortschritt des Agenten zu evaluieren. Dabei rotiert die Methode visuell das Spielfeld gemäß der aktuellen x- und y-Winkel und aktualisiert die Sichtbarkeit der Kugel, wenn diese in ein Loch fällt. Die Kugelposition wird ebenfalls aktualisiert, um die Bewegung im 3D-Raum darzustellen.

### **Step-Methode:**

Diese Methode führt die vom Agenten gewählte Aktion aus, indem die Rotationswinkel des Spielfeldes angepasst und die Position der Kugel entsprechend der neuen Spielfeldneigung berechnet werden. Die genaue Berechnung der Drehbewegung des Spielfeldes wird nachfolgend im Unterkapitel 6.4.3 detailliert erläutert. Falls keine Drehung erfolgt, wird die neue Kugelposition basierend auf der aktuellen Spielfeldneigung berechnet. Andernfalls wird die Drehbewegung des Spielfeldes über mehrere Schritte hinweg angepasst, um eine gleichmäßige Bewegung zu simulieren. Nach zehn Berechnungsschritten der Ballphysik ( $\text{actions\_dt}/\text{physics\_dt} = 10$ ) und dem Ablauf der 100 ms zur Synchronisation mit der 3D-Visualisierung (falls gewählt) wird überprüft, ob sich die Kugel im Ziel, in einem Loch oder nahe an einem Loch befindet. Um beispielsweise festzustellen ob sich eine Kugel nah an einem Loch befindet wird folgende Bedingung geprüft:

$$\sqrt{(x_k - x_l)^2 + (y_k - y_l)^2} < 1,4 \cdot r_l \quad (6.49)$$

Dabei beschreiben  $x$  und  $y$  die Positionskoordinaten der Kugel mit dem Index  $k$  und der Löcher mit dem Index  $l$  mit dem Lochradius  $r$ . Es wurde nur ein kleiner Rand um das Loch definiert, da die Kugel beispielsweise bei der „HOLES\_8“ Spielplatte recht nah an Löchern vorbeirollen muss, um das Labyrinth zu meistern. Nach diesen Überprüfungen wird der Zustandsraum aktualisiert und die konkrete Belohnung berechnet. Die genaue Belohnungsberechnung wird in Kapitel 6.5 erläutert. Zum Schluss wird überprüft ob eine definierte Anzahl an Aktionen überschritten wurde. Dies stellt sicher, dass die Episodenlänge endlich bleibt.

#### **6.4.3 Drehbewegung der Spielplatte**

Um die Übertragbarkeit der trainierten neuronalen Netze aus der virtuellen Umgebung auf den physischen Demonstrator zu verbessern, müssen die Winkelgeschwindigkeit der

Servos und Zeitverluste durch Datenübertragung sowie der Programmbearbeitung berücksichtigt werden. Für das Rotieren des Spielfeldes, das bei der Berechnung der Kugelbewegung benötigt wird, wurde deshalb eine Näherung implementiert. Diese Näherung wird nachfolgend genauer beschrieben. Für den physischen Demonstrator werden die Servos MG996R [3] und COM-Motor02 von Joy-IT [58] verwendet (Genaueres ist Kapitel 7.1.1 zu entnehmen). Der Servo MG996R hat laut Herstellerangaben eine Betriebsgeschwindigkeit (reziproke Winkelgeschwindigkeit) von 0,17 s/60° (4,8 V). Die reziproke Winkelgeschwindigkeit beim COM-Motor02 wird in gleicher Größenordnung wie beim MG996R angegeben. Mit dem messtechnisch bestimmten Übersetzungsverhältnis

$$i = \frac{d_{Drehknopf}}{d_{Riemenscheibe}} = \frac{34 \text{ mm}}{12 \text{ mm}} \quad (6.50)$$

sowie der reziproken Winkelgeschwindigkeit der Servos ergibt sich eine Änderungsrate  $a_\varphi$  für die Spielplatte von:

$$a_\varphi = \frac{0,17 \text{ s}}{60^\circ} \cdot i = 8 \text{ ms/}^\circ \quad (6.51)$$

Als Näherung kann der Berechnungszyklus der Kugelphysik von 10 ms pro 1° angesetzt werden. Das PWM Signal für die Servos hat eine Periodendauer von 20 ms (siehe Genaueres in Kapitel 7.1.1), sodass Werte nur alle 20 ms aktualisiert werden können. Da das Senden der Pulsbreiten an den Mikrocontroller und die Periodendauer nicht synchronisiert sind, wird dafür eine durchschnittliche Zeit von 10 ms angesetzt. Hinzukommen die Übertragungszeit zwischen dem Laptop und dem Mikrocontroller, die Übertragung wird dabei nach max. 2 ms beendet. Bei kleiner eingestellten Timeouts kam es ab und zu zu Datenverlusten. Außerdem ist noch die Pulsbreite des Signals zu berücksichtigen, die bei maximal 2,65 ms liegt (siehe Kapitel 7.1.1). Weiter hinzu kommen beispielsweise noch Programmbearbeitungszeiten. Als grobe Näherung wird eine Totzeit von 20 ms angesetzt, sodass zwei Berechnungszyklen der Kugelphysik mit der vorherigen Spielfeldneigung berechnet werden. Die konkret umgesetzte Näherung für die Drehbewegung der Spielplatte ist in Abbildung 6.17 am Beispiel einer Winkeländerung  $\Delta\varphi$  um 1,5° dargestellt (durchgezogene Linie). Die gestrichelte Linie entspricht dabei dem, wie zuvor beschriebenen, angenähertem realen Verhalten, bei konstanter Winkelgeschwindigkeit.

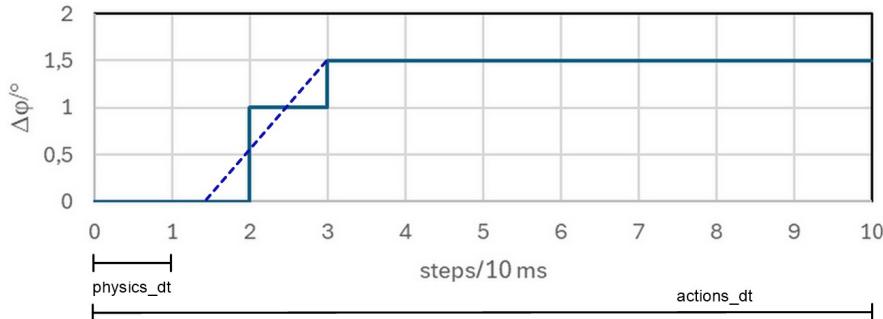


Abbildung 6.17: Drehbewegung der Spielplatte

## 6.5 Rewards

In der Abbildung 6.18 ist das Klassendiagramm zu den Belohnungssystemen bzw. Rewards dargestellt. Im Rahmen dieser Arbeit wurden zwei verschiedene Belohnungsstrategien getestet, die Belohnung mit Kacheln und Zwischenzielen, sowie die Schwellenbelohnung. Ein Vergleich verschiedener Belohnungen ist im weiteren Verlauf dieser Arbeit in Kapitel 8.1.1 zu finden. In der Abgabesoftware werden die Belohnungen mit den Kacheln und Zwischenzielen verwendet. Es wurde eine abstrakte Klasse Rewards implementiert. Die Klasse deklariert die Methoden, die aus der Labyrinth Environment aufrufbar sein müssen. Basierend auf diesen Methoden können einfach weitere Belohnungssysteme implementiert und integriert werden. Die wichtigste Methode ist dabei die *Step*-Methode, mit Hilfe derer die genaue Belohnung basierend auf der getätigten Aktion bestimmt wird.

### 6.5.1 Rewards by Areas

Die Klasse RewardByAreas dient zur Definition der geometrischen Dimensionen und Belohnungsstrukturen für das Trainieren des RL-gesteuertes Labyrinths nach dem Konzept der Kacheln und Zwischenzielen. Dabei geht es vor allem darum, ob sich die Kugel seit der letzten Aktion in die richtige Bewegungsrichtung bewegt hat oder nicht. Dafür werden die Spielplatten in zwei Kategorien eingeteilt. Sie werden in reine Spielplatten, sprich die „HOLES\_0(\_VIRTUAL)“ Platten, und in Labyrinthplatten, worunter die restlichen Platten mit Löchern und Innenwänden fallen, unterschieden. Bei der Belohnung der Labyrinthplatten wird Folgendes unterschieden:

- Hat die Kugel das Ziel erreicht?

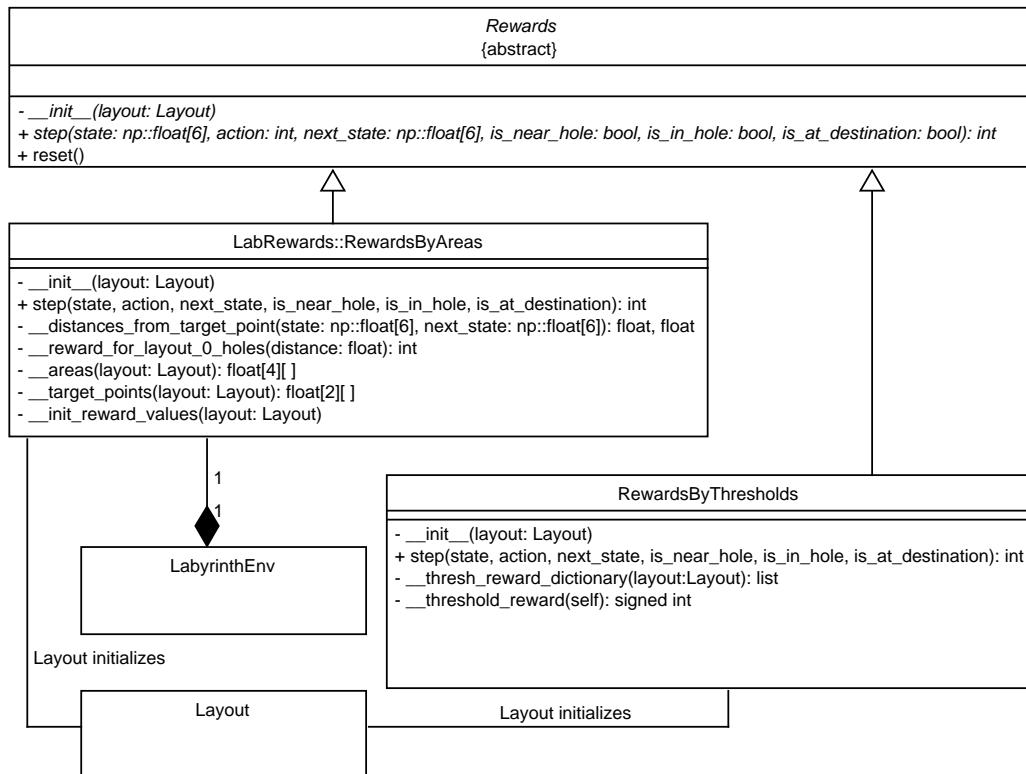


Abbildung 6.18: Klassendiagramm Rewards

- Ist die Kugel in ein Loch gefallen?
- Befindet sich die Kugel in der Nähe eines Loches?
- Gab es eine etwas größere Ballpositionsänderung (mehr als 0,25 cm) in die richtige Bewegungsrichtung seit der letzten Aktion?
- Gab es eine kleine Ballpositionsänderung in die richtige Bewegungsrichtung seit der letzten Aktion?
- Andernfalls

Für die Spielplatten „HOLES\_0(\_VIRTUAL)“ , wird die Belohnungen wie folgt unterschieden:

- Hat die Kugel das Ziel (den Mittelpunkt) erreicht?

- Gab es eine Ballpositionsänderung in die richtige Bewegungsrichtung? (Die Belohnungshöhe wird dabei anhand der gezeichneten Kreissegmente des Spielfeldes in Richtung der Mitte erhöht (*progress*))
- Andernfalls (Die negative Belohnungshöhe wird dabei anhand der gezeichneten Kreise des Spielfeldes in Richtung der Mitte angepasst (*progress*).)

Die genaue Berechnung der richtigen Bewegungsrichtung wurde bereits im Konzeptkapitel 5.1.5 in Formel 5.3 vorgestellt. Dafür kann in Python die Funktion *math.dist()* verwendet werden, die den Euklidischen Abstand der Kugel zum Zwischenzielpunkt bestimmt. Mit Hilfe des letzten und des aktuellen Abstandes kann die Bewegungsrichtung und -weite ermittelt werden.

Die dazu benötigten Kacheln werden in der Methode *\_\_areas()* definiert. Dabei beschreiben Kacheln bestimmte Bereiche im Labyrinth, in denen sich der Ball befinden kann. Die Kacheln werden durch ihre minimalen und maximalen x- und y-Koordinaten definiert. Die Zwischenziele werden in der Methode *\_\_target\_points()* definiert. Die Zwischenziele werden dabei durch ihre x- und y-Positionskoordinaten beschrieben. Die konkreten Belohnungen für verschiedene Ereignisse (siehe Kapitel 6.5) sind in der Methode *\_\_init\_reward\_values()* festgelegt. Die Höhe der Belohnung variiert dabei je nach Spielplatte.

Bei den Spielplatten „HOLES\_0(\_VIRTUAL)“ ist die Belohnungshöhe an den Kreisegmentfortschritt gekoppelt. Für diese Berechnung dient die Methode *\_\_reward\_for\_layout\_0\_holes()*, wobei die aktuell berechnete Distanz von der Kugel bis zum Ziel (Mittelpunkt des Spielfeldes) und die Radien der gezeichneten Kreissegmente verwendet werden. Als Rückgabewert wird der Kreisegmentfortschritt (*radius\_progress*) zurück geliefert. Je höher die Zahl des Rückgabewertes ist, je näher ist das Kreissegment in der Mitte bzw. desto kleiner ist der Kreis, in dem sich die Kugel befindet.

### 6.5.2 Rewards by Thresholds

Eine alternative Belohnungsstrategie stellt die Schwellenbelohnung dar, welche in der Klasse *RewardsByThresholds* implementiert ist. Diese wird in der Abgabeversion nicht aufgerufen, sie wurde aber in Tests zur RL-Parameterbestimmung untersucht und verwendet (siehe weiteres in Kapitel 8.1.1). Die Definition der Schwellen findet in der Methode *\_\_thresh\_reward\_dictionary* statt. Jede Schwelle wird dabei durch eine Achse



Abbildung 6.19: Schwellenüberquerungsberechnung

der Überquerung (x oder y), eine Bewegungsrichtung des Überquerens, die Schwellenkoordinate, die übertreten werden soll, und den Bereich (min und max), in dem die Schwelle übertreten werden soll, beschrieben. Für die Belohnungshöhen sind hierbei die Überschreitungen von Schwellen in positiver oder negativer Richtung von Bedeutung. Die Bedingung für eine Schwellenüberschreitung in die richtige Richtung wird nachfolgend am Beispiel einer Schwellenüberquerung der x-Koordinate  $S_x$  von größeren x-Werten zu kleineren x-Werten (siehe Abbildung 6.19, die blau gestrichelte Linie entspricht der Schwelle) beschrieben:

$$(x_{k-1} > S_x) \wedge (x_k < S_x) \wedge (y_k > y_{min}) \wedge (y_k < y_{max}) \quad (6.52)$$

Das bedeutet die letzte x-Positionskoordinate der Kugel ( $x_{k-1}$ ) muss größer als die der Schwelle sein und die neue x-Positionskoordinate der Kugel ( $x_k$ ) muss kleiner sein. Zudem muss sich die y-Positionskoordinate der Kugel in Bereich der y-Schwellenlänge ( $y_{min}$  bis  $y_{max}$ ) befinden.

## 6.6 Deep Q-Learning

Um durch Reinforcement Learning gesteuertes Spielen der Labyrinthe zu ermöglichen, wurde das im Konzept (Kapitel 5.1.6) gewählte Deep Q-Learning umgesetzt. Der grundlegende Deep Q-Learning Algorithmus, wie er von Google's DeepMind vorgestellt wurde, ist in Abbildung 6.20 dargestellt. Zu Beginn werden der Replay-Buffer  $D$  mit der Kapazität  $N$  sowie die Aktion-Wert-Funktion (Q-Funktion) durch ein neuronales Netz mit zufälligen Gewichten  $\Theta$  initialisiert. Danach durchläuft der Agent  $M$  Episoden. Zu Beginn jeder Episode wird der Anfangszustand  $s_1$  und die vorverarbeitete Sequenz  $\Phi_1$  initialisiert. Dabei beschreibt  $\Phi$  eine Funktion, die dazu dient den Zustand so zu transformieren, dass er als Eingabe für ein neuronales Netz geeignet ist. Anschließend werden die Zeitschritte  $t$  innerhalb der Episode durchlaufen. Zuerst wird eine Aktion  $a_t$  nach der Epsilon-Greedy-Strategie bzw. der Epsilon-Reduzierungs-Strategie ausgewählt. Danach wird diese Aktion ausgeführt und die resultierende Belohnung  $r_t$  sowie das neue Bild

---

**Algorithm 1** Deep Q-learning with Experience Replay

---

```

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

```

---

Abbildung 6.20: DQN Algorithmus [51]

(image)  $x_{t+1}$  beobachtet. Der neue Zustand  $s_{t+1}$  wird bestimmt und mit Hilfe der Vorverarbeitung  $\Phi_{t+1}$  so transformiert, dass er als Eingabe für ein neuronales Netz dienen kann. Die entstandene Transition  $(\Phi_t, a_t, r_t, \Phi_{t+1})$  wird im Replay-Memory gespeichert. Darauffolgend werden zufällige Minibatches aus dem Replay-Buffer gezogen. Für jede Transition im Minibatch wird der Zielwert  $y_j$  bestimmt. Bei einem Endzustand entspricht es der erhaltenen Belohnung selbst, andernfalls entspricht es der Differenz aus der beobachteten Belohnung sowie der diskontierten Schätzung der maximal künftigen Belohnung (siehe auch Kapitel 2.4.2 zum Q-Learning). Zum Schluss wird ein Gradientenabstieg durchgeführt, bei dem die quadratische Differenz zwischen dem tatsächlichen Wert  $y_i$  und dem von einem Modell vorhergesagten Wert  $(Q(\Phi_j, a_j; \Theta))$  minimiert werden soll und die Gewichte dementsprechend angepasst werden können.

In Anlehnung an diesen grundlegenden Deep Q-Learning Algorithmus wurde der eigene Agent programmiert. Das dazugehörige Klassendiagramm ist in Abbildung 6.21 dargestellt. In den nachfolgenden Unterkapiteln werden die einzelnen Klassen detaillierter betrachtet.

### 6.6.1 Replay-Buffer

Die Klasse *ReplayBuffer* repräsentiert das Replay-Memory, welches für Deep Q-Learning verwendet wird. Der Replay-Buffer speichert vergangene Erfahrungen, um das Korrelationsproblem aufeinanderfolgender Trainingsdaten zu verringern und die Trainingsstabilität zu erhöhen. Das Replay-Memory wird als FIFO (first in first out)-Speicher

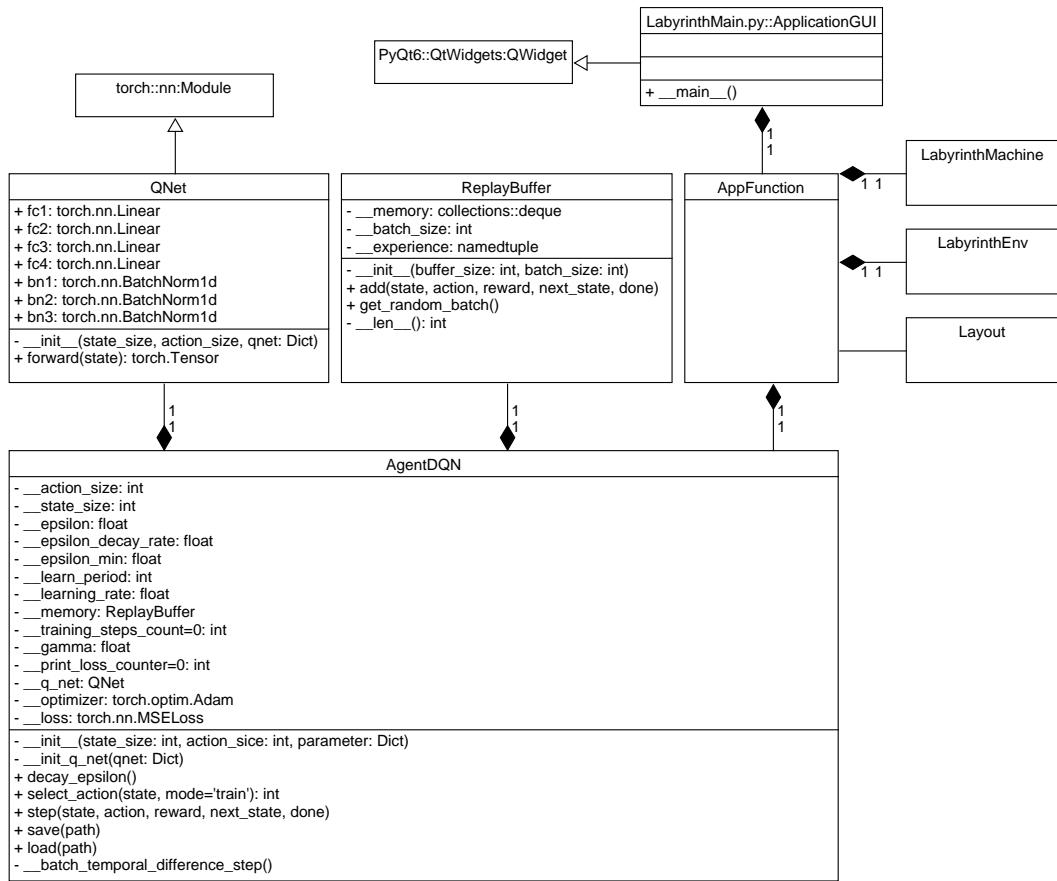


Abbildung 6.21: Klassendiagramm des DQN-Agenten

verwendet. Dabei wird effektives Hinzufügen und Entfernen von Elementen ermöglicht. Zudem kann die maximale Länge festgelegt werden. Wenn die Warteschlange bzw. der Puffer seine maximale Länge erreicht hat und ein neues Element hinzugefügt wird, wird automatisch immer das älteste Element entfernt. Dies verhindert das Überlaufen des Speichers und stellt sicher, dass nur die aktuellsten Erfahrungen gespeichert werden. Mit Hilfe der erstellten `add`-Methode werden die Erfahrungen strukturiert durch ein benanntes Tupel im Replay-Memory gespeichert. Die `get_random_batch`-Methode ermöglicht das zufällige Ziehen von Mini-Batches aus dem Replay-Buffer, die für das Training des neuronalen Netzes verwendet werden.

### 6.6.2 QNet

*QNet* stellt die Klasse für das neuronale Netz des Deep Q-Learnings dar. Es besteht aus mehreren vollständig verbundenen Schichten (linear layer), die in Pytorch durch *nn.Linear* repräsentiert werden. *nn.Linear* benötigt dabei zwei Parameter, die Anzahl der Eingänge und Ausgänge. Das neuronale Netz besteht aus vier linearen Schichten *fc1* bis *fc4*. *fc1* wandelt beispielsweise die Eingabe der Dimension *state\_size* (Dimension des Zustandraums) in *fc1\_unis* (Anzahl der Neuronen der ersten verdeckten Schicht) um. Zusätzlich wird zwischen den einzelnen Schichten Batch-Normalisierung angewendet um die Abhängigkeit der Eingabeverteilungen zu reduzieren und die Ausgaben der verdeckten Schichten zu normalisieren, wodurch das Training stabilisiert und beschleunigt wird. Dies wird durch *nn.BatchNorm1d* umgesetzt, welches für ein-dimensional strukturierte Daten geeignet ist, wie sie in vollständig verbundenen Schichten vorkommen. Eine 2D-Batch-Normalisierungsschicht könnte beispielsweise bei CNN Netzen eingesetzt werden. Als Aktivierungsfunktion wird leaky ReLU verwendet, die in PyTorch durch *nn.functional.leaky\_relu* bereitgestellt wird. Die zufällige Gewichtsinitialisierung erfolgt dementsprechend mit der Kaiming-Initialisierung, auch He-Initialisierung genannt (siehe Kapitel 2.3.8). Die Initialisierung kann in Pytorch durch *nn.init.kaiming\_uniform\_* erfolgen. Diese Initialisierungsmethode hilft, die Trainingsdynamik zu stabilisieren, indem sie die Gewichtswerte auf eine Weise verteilt, die auf die Arten der ReLU-Aktivierungsfunktion abgestimmt ist. Die erstellte *forward*-Methode ermöglicht den Vorwärtsdurchlauf durch das neuronale Netz. Der Zustand (*state*) wird durch die einzelnen linearen Schichten und die Batch-Normalisierungen, sowie durch die Aktivierungsfunktion geleitet. Die finale Ausgabe erfolgt durch die letzte lineare Schicht *fc4*, die die Q-Werte für jede mögliche Aktion zu dem gegebenen Zustand zurückgibt.

### 6.6.3 Agent

Die Klasse *AgentDQN* implementiert die wesentlichen Funktionen für das Trainieren und die Aktionsauswahl des Agenten. Sie integriert somit auch das neuronale Netz sowie den Replay-Buffer. Nachfolgend werden die einzelnen Methoden der Agentenklasse detaillierter vorgestellt.

#### **Agenteninitialisierung:**

Bei der Agenteninitialisierung werden verschiedenste Parameter wie Epsilon, Zustands- und Aktionsgröße, Gamma, Batchgröße und die Lernrate initialisiert. Der Replay-Buffer

wird ebenfalls initialisiert und die Methode `__init_q_net` aufgerufen, welche das neuronale Netz initialisiert sowie den Optimierer und die Verlustfunktion definiert. Als Optimierer wird Adam verwendet, welcher in Pytorch durch `optim.Adam` verwendet werden kann. Als Verlustfunktion wird MSE (Mean Square Error) eingesetzt, die durch `nn.MSELoss()` nutzbar ist.

#### **Epsilon-Aktualisierung:**

Zu Beginn jeder Episode erfolgt die Anpassung von Epsilon (`decay_epsilon()`), da eine Epsilon-Reduzierungs-Strategie verwendet wird. Dabei wird Epsilon  $\epsilon$  gemäß folgender Formel reduziert:

$$\epsilon \leftarrow \max(\epsilon_{decay} \cdot \epsilon, \epsilon_{min}) \quad (6.53)$$

$\epsilon_{decay}$  beschreibt dabei die Reduzierungsrate und  $\epsilon_{min}$  das Minimum, auf welches  $\epsilon$  sinken kann.

#### **Aktionsauswahl:**

Eine Trainings- oder Evaluierungssequenz beginnt mit der Auswahl einer Aktion nach der Epsilon-Greedy Strategie (`select_action`-Methode). Für die Aktionswahl nach der Greedy Strategie muss der state in den richtigen Datentyp von einem Numpy Array in ein Pytorch Tensor konvertiert werden, damit das neuronale Netz diesen verarbeiten kann. Neuronale Netzwerke in PyTorch erwarten typischerweise Eingaben in der Form: Batchgröße (hier 1), Featuregröße (hier 6 Zustandsmerkmale). Anschließend wird das neuronale Netz in den Evaluierungsmodus (`self.__q_net.eval()`) geschaltet und die Gradientenberechnung deaktiviert (`torch.no_grad()`). Dadurch kann Speicherplatz und Rechenzeit eingespart werden, da kein Backpropagation durchgeführt werden muss, sondern nur eine Vorhersage getätigt wird. Mit Hilfe der Vorhersage wird die Aktion gewählt, die den höchste Q-Wert liefert hat. Nach der Vorhersage wird das Netz wieder zurück in den Trainingsmodus versetzt (`self.__q_net.train()`).

#### **Trainieren:**

Die Beobachtungen aus der Environment (state, action, reward, next\_state, done) werden an die `step`-Methode des Agents übergeben. Dort werden diese Beobachtungen (Erfahrungen) im Replay-Buffer gespeichert. Sobald genügend Erfahrungen fürs Training im Replay-Buffer enthalten sind, wird die `__batch_temporal_difference_step`-Methode aufgerufen. Diese Methode ermöglicht das Training des neuronalen Netzes. Zu Beginn wird ein Minibatch aus dem Replay-Buffer gezogen und in Pytorch Tensoren konvertiert, damit die Daten vom neuronalen Netz verarbeitet werden können. Danach wird die temporale Differenzmethode (*td*), unter Berücksichtigung, ob ein Endzustand erreicht

wurde ( $\text{dones} = 1$ ) oder nicht ( $\text{dones} = 0$ ), gebildet:

$$td = r + (\gamma * \max(Q(s', a')) \cdot (1 - \text{dones})) - Q(s, a) \quad (6.54)$$

Die temporale Differenzmethode und deren Parameter wurde bereits in Kapitel 2.4.2 erläutert. Dieses Vorgehen ist auch in dem vorgestellten grundlegenden Deep Q-Learning Algorithmus von DeepMind in Abbildung 6.20 wieder zu finden. Anschließend wird mit Hilfe der gewählten Verlustfunktion MSE der Fehler zwischen  $td$  und Null bestimmt ( $\text{error} = self.__loss(td, torch.zeros(td.shape))$ ). Bevor die Backpropagation bzw. die Anpassung der Gewichte erfolgen kann, müssen die Gradienten aller optimierbaren Parameter auf Null zurückgesetzt werden ( $self.__optimizer.zero_grad()$ ). Dieses Zurücksetzen ist nötig, da in PyTorch Gradienten standardmäßig bei jeder Backpropagation-durchführung akkumuliert (addiert) werden. Ohne Zurücksetzen würde es deshalb zu falschen Netzanpassungen kommen. Anschließend werden die Gradienten des Fehlers berechnet werden ( $\text{error.backward()}$ ). Zum Schluss werden die Parameter des neuronalen Netzes durch den Optimizer Adam mit Hilfe der berechneten Gradienten aktualisiert werden ( $self.__optimizer.step()$ ).

### Speichern und Laden:

Um auf trainierte Agenten zurückgreifen zu können müssen zuerst die Parameter (Gewichte und Biases) des neuronalen Netzes gespeichert werden. Dies erfolgt mit der `save()`-Methode. Dadurch kann beispielsweise der Lernfortschritt dokumentieren und später unterschiedliche Stände evaluiert sowie die besten Lernzustände ermittelt und genutzt werden. In der Agenten `save()`-Methode werden die erlernbaren Parameter in einem Wörterbuch (dict) gespeichert und unter einem definierten Dateipfad und -namen gespeichert (`torch.save(self.__net.state_dict(), path)`). Um trainierte Netze evaluieren oder weiter trainieren zu können wurde die `load()`-Methode implementiert. Dabei kann ein gespeichertes Wörterbuch mit seinen erlernten Parametern wieder geladen werden (`self.__net.load_state_dict(torch.load(path))`).

### 6.6.4 Main

Im Rahmen dieser Abschlussarbeit wurde eine GUI (Graphical User Interface) entwickelt (siehe Abbildung 6.22). Die GUI ermöglicht es dem Benutzer, alle notwendigen Parameter zur Steuerung des Trainings oder der Evaluation eines RL-Algorithmus auf

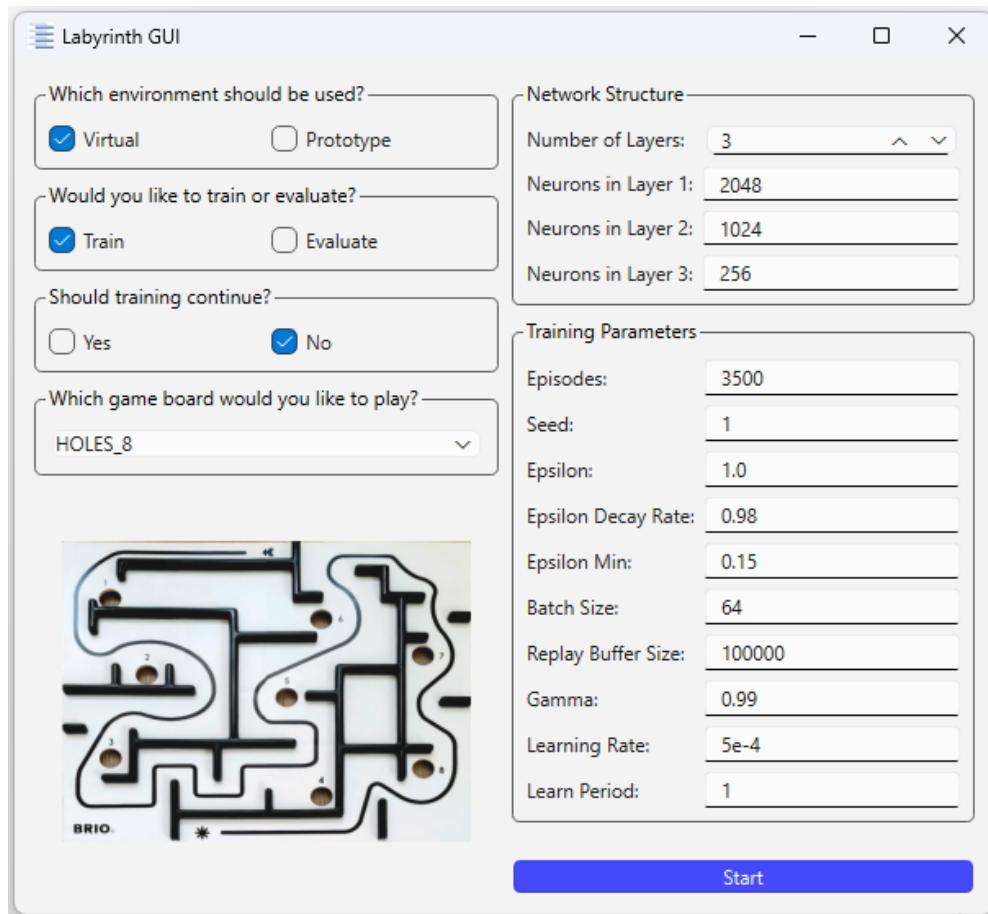


Abbildung 6.22: Benutzeroberfläche für das Training

einfache und intuitive Weise anzupassen. Die Benutzeroberfläche ermöglicht dabei verschiedenste Eingabemöglichkeiten. Dabei kann die Umgebung, in der gespielt werden soll, gewählt werden: virtuell oder am physischen Demonstrator (Prototyp). Es kann zwischen dem Trainingsmodus (zur Verbesserung des Agentenverhaltens) und dem Evaluationsmodus (zur Bewertung der Performance) unterschieden werden. Im Trainingsmodus hat der Nutzer zudem die Möglichkeit, das Training mit bereits gelernten Agenten fortzusetzen. Sechs verschiedene Spielplatten stehen dem Benutzer zur Verfügung, welche unterschiedliche Layouts und Schwierigkeitsgrade besitzen. Je nach Auswahl wird das jeweilige Layout der Spielplatte angezeigt, um den Benutzer visuell zu unterstützen. Die Struktur des neuronalen Netzes kann auch konfiguriert werden. Dabei kann der Benutzer die Anzahl der verdeckten Schichten (hidden layers) und deren Neuronenzahl individuell anpassen. Es können bis zu drei verdeckte Schichten konfiguriert und

verwendet werden. Als letzte Einstellmöglichkeit stehen beim Trainieren verschiedenste Parameter zur Justierung zur Verfügung, wie zum Beispiel die Anzahl der Episoden, die Lernrate oder der Epsilon-Wert. Diese Parameter beeinflussen maßgeblich das Verhalten des RL-Algorithmus. Als Standardparameter sind bei der Netzstruktur und den Trainingsparametern jeweils die Werte hinterlegt, mit denen die einzelnen Spielplatten trainiert wurden. Zum Abschluss der Konfiguration kann der Nutzer durch Drücken des Start-Buttons das Training oder die Evaluation, basierend auf den getroffenen Einstellungen, beginnen.

Die GUI wurde unter Verwendung von Python und der PyQt6-Bibliothek implementiert, welche eine leistungsstarke Plattform zur Erstellung plattformübergreifender Desktop-Anwendungen darstellt. Das dazugehörige Klassendiagramm für diese GUI und die hinterlegten Funktionen für das Training und die Evaluation sind in Abbildung 6.23 dargestellt. Nachfolgend wird konkreter auf die Trainings- und Evaluationsfunktionen in der Klasse *AppFunction* eingegangen.

### Training

Der auf die Grundfunktionalität zusammengestellte Trainingsalgorithmus ist in Listing 6.1 dargestellt. Das vollständige Programm kann der CD entnommen werden. Dieses Listing verdeutlicht die Ähnlichkeit zum grundlegenden Deep Q-Learning Algorithmus, der zuvor in Abbildung 6.20 dargestellt und erläutert wurde. Zu Beginn werden die Trainingsumgebung *env* und der Agent initialisiert. Der *render\_mode* der Trainingsumgebung ist standardmäßig *None*, um beim Trainieren Rechenzeit einzusparen. Zu Beginn jeder Episode wird die Trainingsumgebung auf einen definierten Anfangszustand gesetzt (*env.reset*) und Epsilon angepasst. Am Anfang der Trainingssequenz wählt der Agent eine Aktion aus. Die ausgewählte Aktion wird anschließend ausgeführt und die Belohnung sowie der nächster Zustand erfasst (*env.step*).

Nach ausgeführter *step*-Methode wird der Zustand aktualisiert und es wird geprüft, ob eine Episode zu Ende ist. Eine Episode wird entweder durch das Erreichen des Ziels beendet (*done*) oder wenn eine gewisse Anzahl an durchgeführten Aktionen überschritten wurde (*truncated*), damit die Episode endlich bleibt. Am Ende eines Trainings, wird ein Diagramm erzeugt, welches die Belohnung über die Episoden hinweg anzeigt. Dieses Diagramm liefert oft schon einen Anhaltspunkt dafür, wie erfolgreich und stabil das durchgeführte Training war und welche gespeicherten Parameterdateien einen erfolgreich trainierten Agenten zeigen.

### Evaluation

Für die Evaluierung werden nicht so viele Einstellparameter wie für das Training be-

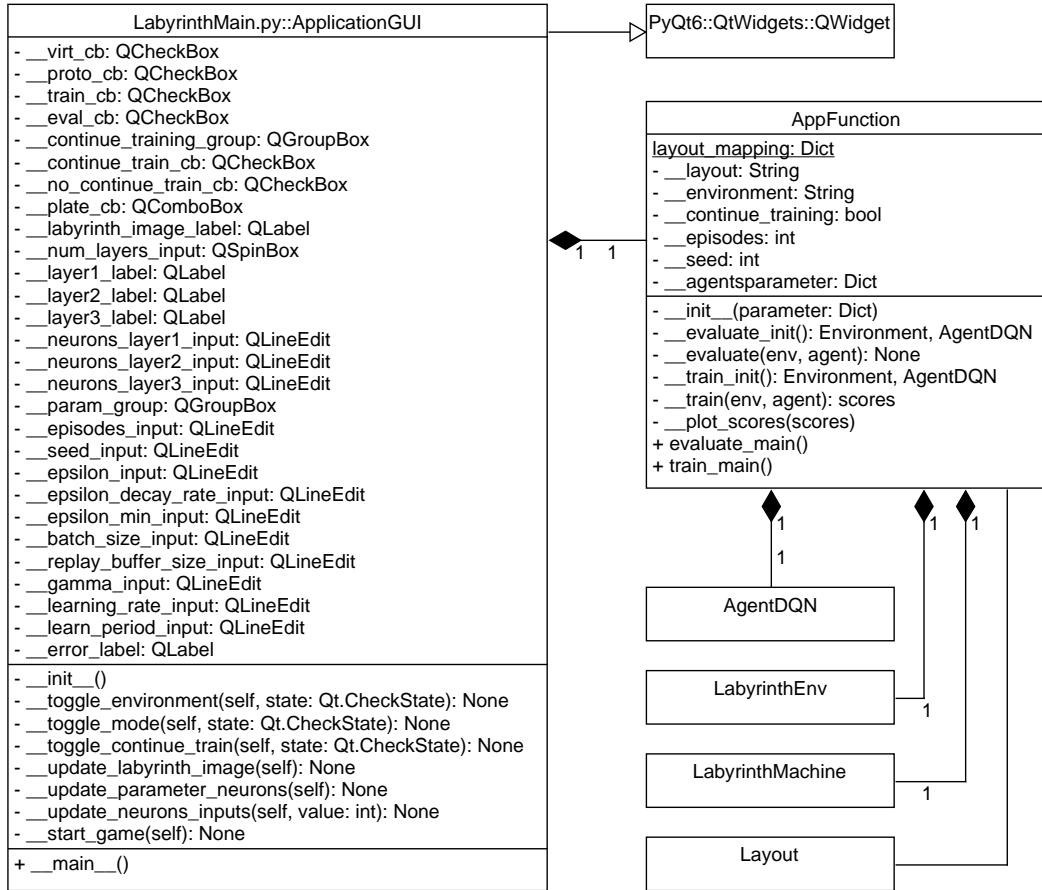


Abbildung 6.23: Klassendiagramm der Main

nötigt. Aus dem Grund werden dem Benutzer hierfür nur eine kleinere Anzahl an Einstellmöglichkeiten in der GUI bereitgestellt (siehe Abbildung 6.24). Für das Evaluieren werden gespeicherte Dateien von trainierten neuronalen Netzen wieder geladen. Das erlernte Verhalten kann in der 3D-Simulation (`render_mode='3D'`) visualisiert werden. Es kann aber auch eine Evaluierung am physischen Demonstrator durchgeführt werden. Die Evaluierungs-Funktionen sind recht ähnlich aufgebaut wie die zum Trainieren. Hierbei wird beispielsweise auch die `select_action`-Methode des Agenten verwendet, wobei die Aktion ausschließlich auf Basis des maximalen Q-Wertes gewählt wird. Auf Grund der langen Rechenzeit beim Training von großen Netzen bei der Verwendung eines normalen Laptops wurde das Training der komplexeren Labyrinthplatten („HOLES\_8“ und „HOLES\_21“) auf wenige kleinere Netze aufgeteilt. Die Aufteilung erfolgte gleichmäßig

```
1 # Init environment and agent
2 env = LabyrinthEnvironment(layout=self.__layout, render_mode=None)
3 agent = LabyrinthAgentDQN(state_size = 6, action_size = env.
4     num_actions_per_component * 2)
5 # Train agent
6 for e in range(1, self.__episodes + 1):
7     state, _ = env.reset()
8     agent.decay_epsilon()
9     while True:
10         action = agent.select_action(state)
11         next_state, reward, done, truncated, _, _ = env.step(action)
12         agent.step(state, action, reward, next_state, done)
13         state = next_state
14         if done or truncated:
15             break
16     if e % 25 == 0:
17         save_path = model_path + str(e) + "_" + f'{self.__layout.name}.pth'
18         agent.save(save_path)
```

Listing 6.1: Main

anhand des Fortschritts im Labyrinth. Bei der Evaluierung wird zwischen diesen Netzen automatisch umgeschaltet.

### Fehlerbehandlung bei Eingaben in der GUI

Die einzustellenden Parameter für das Training bzw. das Evaluieren werden auf ihre Gültigkeit geprüft. Hierfür wurden in der grafischen Benutzeroberfläche verschiedene Mechanismen zur Fehlerüberprüfung implementiert. Dabei wird zum einen überprüft, ob die Eingaben den richtigen Datentyp beinhalten, sodass beispielsweise keine Buchstaben oder wie beispielsweise bei der Größe des Replay-Buffers nur ganze Zahlenwerte als Eingaben erlaubt sind. Zum anderen wird der Zahlenwertebereich überprüft, da beispielsweise die Epsilon-Werte nur in einem Bereich von null bis eins möglich bzw. definiert sind. Bei fehlerhaften Eingaben wird unterhalb des Start-Buttons eine entsprechende Fehlermeldung eingeblendet (Beispieltexte der Fehlermeldungen sind in Abbildung 6.25 zu sehen). Das Training oder die Evaluation kann im Fehlerfall nicht gestartet werden. Für die CheckBoxen wurden Togglemechanismen integriert, sodass pro Fragengruppe immer nur eine der beiden Auswahlmöglichkeiten aktiviert werden kann. Dies verhindert Mehrfachauswählen und erleichtert dem Benutzer die Bedienung.

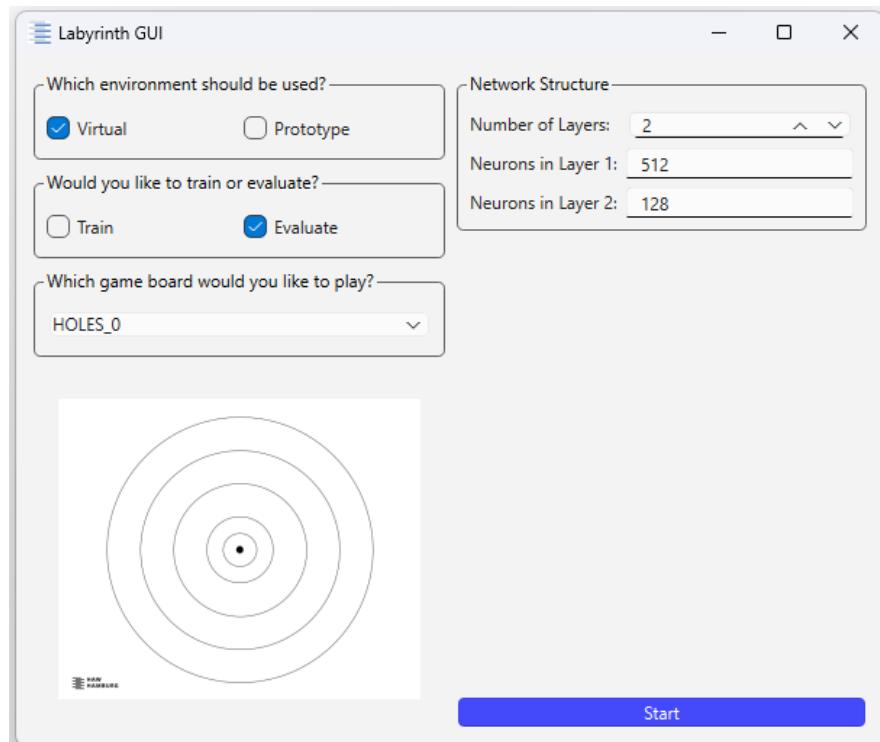


Abbildung 6.24: Benutzeroberfläche für das Evaluieren

Check the data types of the inputs, not all are correct.  
Some input parameters are outside the permitted definition range

Abbildung 6.25: Fehlermeldungen der Benutzeroberfläche



# 7 Entwicklung des physischen Demonstrators

In diesem Kapitel wird die Umsetzung des physischen Demonstrators hard- und softwareseitig vorgestellt und erläutert.

## 7.1 Hardware

Zu Beginn werden die eingesetzten Elektronik-Komponenten und zusätzlich benötigte Bauteile zur Automatisierung des realen BRIO Labyrinthspiels vorgestellt. Anschließend werden die eingesetzten 3D-gedruckten Elemente, wie die benötigten Servohalterungen vorgestellt. Den Abschluss dieses Unterkapitels bildet die Vorstellung der designten und realisierten weiteren Spielplatten.

### 7.1.1 Elektronik

In diesem Unterkapitel werden die verschiedenen genutzten Elektronik-Komponenten zur Automatisierung des BRIO-Labyrinthes vorgestellt. Die Hauptkomponenten, der Mikrocontroller, die Servos und der Servotreiber, sowie deren Verbindungen sind in einem entsprechenden Verdrahtungsplan in Abbildung 7.1 dargestellt.

Der Mikrocontroller wird über ein USB-Kabel mit dem Laptop verbunden. Darüber wird er zum einen mit Spannung versorgt und zum anderen erfolgt hierüber die Datenübertragung der gewünschten Winkelstellungen des Spielfeldes. Für den Servotreiber wird eine externe 5 V Spannungsversorgung benötigt [6]. Dafür wurde ein 9 V Steckernetzteil in Verbindung mit einem Tiefsetzsteller, der die 5 V Ausgangsspannung ermöglicht, sowie eine kleines Steckboard verwendet. Die konkret verwendeten Bauteilkomponenten sind der Tabelle 7.1 aufgelistet. Die Servos werden mittels PWM (Pulsweitenmodulation [25]) gesteuert. Ein PWM-Signal ist ein Rechteckimpuls, der typischerweise zwischen

## 7 Entwicklung des physischen Demonstrators

---

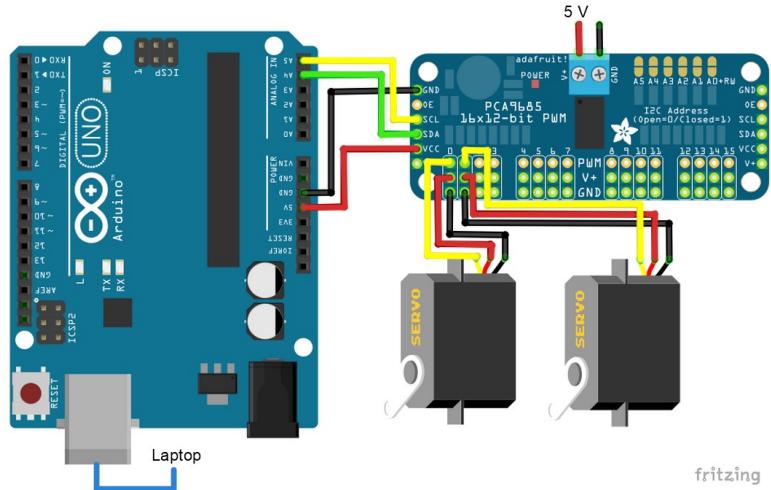


Abbildung 7.1: Verdrahtungsplan

Tabelle 7.1: Elektronik-Bauteilkomponenten

Bauteil	Firma	Eigenschaften	Preis
Microcontroller Board	AZ-Delivery	Arduino UNO R3 [4]	9,99 € [5]
Servotreiber PCA9685	AZ-Delivery	16-Kanal 12-Bit PWM [7]	12,99 € [6]
Servomotor MG996R	AZ-Delivery	Arbeitswinkel = 180° (500 µs - 2500 µs) [8]	9,99 € [3]
Servomotor COM-Motor02	Joy-IT	Arbeitswinkel = 180° (500 µs - 2500 µs) [40]	16,59 € [58]
Tiefsetzsteller Bread Power SBC-POW-BB	Joy-IT	von 6 V - 12 V oder 5 V zu 5 V oder 3,3 V [39]	3,15 € [65]
W06 1080P HD Webcam	Jelly Comb	1920x1080 pixel, 30 fps [38]	23,44 € [37]

$U_{max}$  und GND oszilliert (siehe Abbildung 7.2). Die Periodendauer  $T$  ist hierbei konstant und lässt sich aus der Frequenz  $f$  berechnen. Die Frequenz  $f$  liegt typischerweise bei 50 Hz und wird somit in der Konfiguration verwendet.

$$T = \frac{1}{f} = \frac{1}{50 \text{ Hz}} = 20 \text{ ms} \quad (7.1)$$

Die vom Hersteller angegebenen Pulsbreiten, zwischen denen variiert werden kann, sind in Tabelle 7.1 aufgeführt. Die tatsächlich möglichen Pulslängen der Endlagen wurden an den realen Servos ermittelt. Hierzu wurden die Pulswerte soweit verändert, bis die Servos ihre tatsächliche Endlage erreichten und somit keine weitere Drehung der Servos mehr erfolgte. Für den Servo MG996R ergab sich ein minimaler Pulswert  $t_{min}$  von

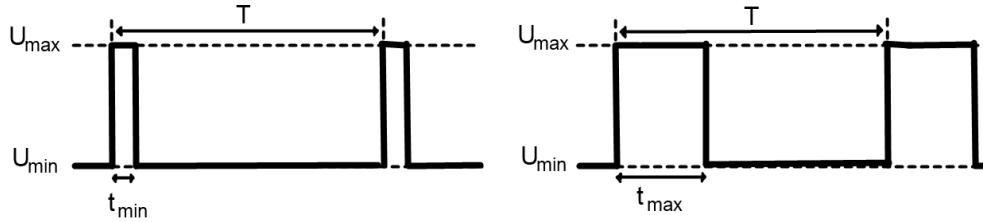


Abbildung 7.2: PWM-Signale

Tabelle 7.2: Weitere Bauteilkomponenten

Bauteil	Firma	Eigenschaften	Preis
Zahnriemen GT2	DOLD Mechatronik	Breite 6 mm, Länge 1 m	2,50 € [19]
Zahnriemenscheibe 20 und 60 Zähne	Zeberoxyz 3D	6,35 mm Bohrungsdurchmesser, 6 mm Riemen	14,99 € [79]

350  $\mu$ s und ein maximaler Pulswert  $t_{max}$  von 2650  $\mu$ s. Es konnte ein minimaler Pulswert von 420  $\mu$ s und ein maximaler Pulswert von 2540  $\mu$ s für den Servo COM-Motor02 von Joy-IT bestimmt werden.

Die Kommunikation zwischen dem Mikrocontroller Board und dem PWM-Treiber erfolgt über einen I<sup>2</sup>C-Bus (Inter-Integrated Circuit [35]). Dies ist ein serieller Datenbus (Kommunikation erfolgt über einen Datenkanal), welcher zwei Signalleitungen benötigt: die Takteleitung SCL (Serial Clock) und die Datenleitung SDA (Serial Data).

### Weitere Bauteile:

Für die mechanische Umsetzung der Drehbewegung von den Servos auf die Drehräder des Labyrinth Spiels werden Zahnriemen und Zahnriemenscheiben eingesetzt. Als Zahnriemenscheiben stehen zwei verschiedene Durchmesser zur Verfügung. Diese Bauteilkomponenten sind in der Tabelle 7.2 zusammengefasst.

Zudem wird ein Kameragestell benötigt, welches die Kamera oberhalb des Spielfeldes platziert. Es wurde aus vorhandenen Aluminium-Profilen (20x20 mm, B-Typ, 6Nut) sowie verschiedenem Zubehör gebaut. Dieses Kameragestell ist in Abbildung 7.3 dargestellt. Bis auf die Zahnriemenscheibe hatte Herr Prof. Dr. Hensel, der Betreuer dieser Arbeit, all die vorgestellten (Elektronik-)Bauteilkomponenten vorrätig, somit entstanden dafür keine Kosten.



Abbildung 7.3: Kameragestell

### 7.1.2 3D-gedruckte Elemente

Für die Befestigung der Servos an dem Spiel wird ein Klemmprinzip verwendet, damit kein Eingriff in das ursprüngliche Spiel vorgenommen werden muss. Dafür wurden Servohalterungen entworfen und gedruckt. Da die Servohalterung unter das Spiel geklemmt wird, erhöhte sich das Spielfeld und stand schief. Abhilfe konnte durch zusätzliche Standfüße für die anderen Ecken geschaffen werden. Als letzte Komponenten wurden noch Drehradklemmen gedruckt, damit die Zahnräder nicht von den Drehräder rutschen. Die dazugehörigen CAD-Zeichnungen sind in der Abbildung 7.4 dargestellt. Bei den 3D-gedruckten Elementen hat Herr Prof. Dr. Hensel Unterstützung geleistet, da der Zugang zu einem 3D-Drucker schwierig war. Alle STL-Dateien der 3D Elemente sind auf der CD einsehbar und werden über ein GitHub-Repository<sup>1</sup> zugänglich gemacht.

### 7.1.3 Eigene Spielplatten

Als Erweiterung zu den verschiedenen BRIO Labyrinthen sind zwei selbst designte Spielplatten „HOLES\_0“ und „HOLES\_2“ realisiert worden (siehe Abbildung 7.5). Dazu sind Hartschaumplatten bedruckt worden. Für die Spielplatte „HOLES\_2“ wurden zudem die beiden Wandelemente mit dem 3D-Drucker erstellt.

---

<sup>1</sup>[https://github.com/MarcOnTheMoon/ai\\_labyrinth](https://github.com/MarcOnTheMoon/ai_labyrinth)

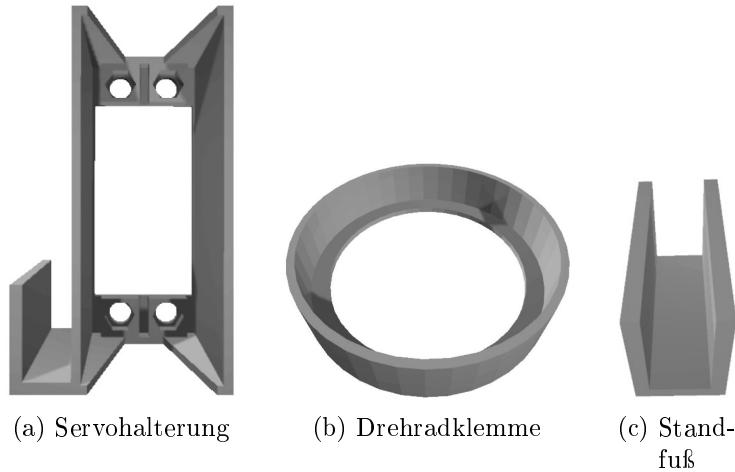


Abbildung 7.4: CAD-Zeichnungen

## 7.2 Software

In Abbildung 7.6 sind die realisierten Klassen für den physischen Demonstrator in Form eines Klassendiagramms dargestellt. In den nachfolgenden Unterkapiteln werden die einzelnen Klassen detaillierter betrachtet und erläutert. Dabei wird zuerst auf die Klasse `LabyrinthMachine` eingegangen, die als Schnittstelle zwischen der Bildverarbeitung und der Servoansteuerung dient. Anschließend wird die Umsetzung der Servokommunikation und -ansteuerung genauer erläutert. Zum Schluss wird kurz auf die Bildverarbeitung eingegangen. Das Training am physischen Demonstrator kann, analog zur virtuellen Umgebung, durch das Ausführen der Hauptfunktion der Pythondatei `LabyrinthMain` gestartet werden. Der einzige Unterschied darin, dass jeweils eine andere Umgebung verwendet wird:

- physischer Demonstrator:

```
env = LabyrinthMachine(layout=Layout.HOLES_0, cameraID=0)
```

- virtuelle Umgebung:

```
env = LabyrinthEnv(layout=Layout.HOLES_0, render_mode='3D')
```



Abbildung 7.5: Realisierte Spielplatten

### 7.2.1 physische Umgebung

Die Klasse LabyrinthMaschine ist vergleichbar mit der Klasse LabyrinthEnv der virtuellen Umgebung (siehe Kapitel 6.4). Beide Klassen verwenden eine reset und step-Methode und greifen auf Methoden der Klasse RewardsByAreas zu. Sie stellt somit die Schnittstelle zwischen der Aktionsausführung (Ansteuerung der Servomotoren) und Bestimmung des neuen Zustandes (Positionsbestimmung der Kugel auf Kamerabildern unter Verwendung von Bildverarbeitung) dar.

#### **Init-Methode:**

Zu Beginn wird das Zeitintervall zwischen zwei Aktionen standardmäßig mit 100 ms definiert. Außerdem wird die Kommunikation zu den Servomotoren initialisiert und diese werden in die Mittelstellung ( $0^\circ$ ) gefahren. Anschließend wird eine Messagebox mit einer Erläuterung, wie die einmalige Kalibration für die Bildverarbeitung durchzuführen ist (siehe Abbildung 7.7), erzeugt. Diese Messagebox wurde mit Hilfe der Python-Bibliothek TKinter erstellt (*messagebox.showinfo*). Nachdem der Hinweis gelesen und bestätigt wurde, wird eine Kalibration ermöglicht. In Abbildung 7.8 ist der kalibrierte Zustand zu sehen. Diese Kalibrationen werden für die Bildverarbeitung und die Bestimmung der Kugelposition benötigt. Die Kalibrationsmöglichkeiten werden durch die Bildverarbeitungsklassen bereitgestellt (siehe Kapitel 7.2.3). Zudem werden in der Init-Methode die Bildpunkte auf die Dimensionen bzw. Längen des realen BRIO Spielfeldes angepasst und der Aktionsraum definiert.

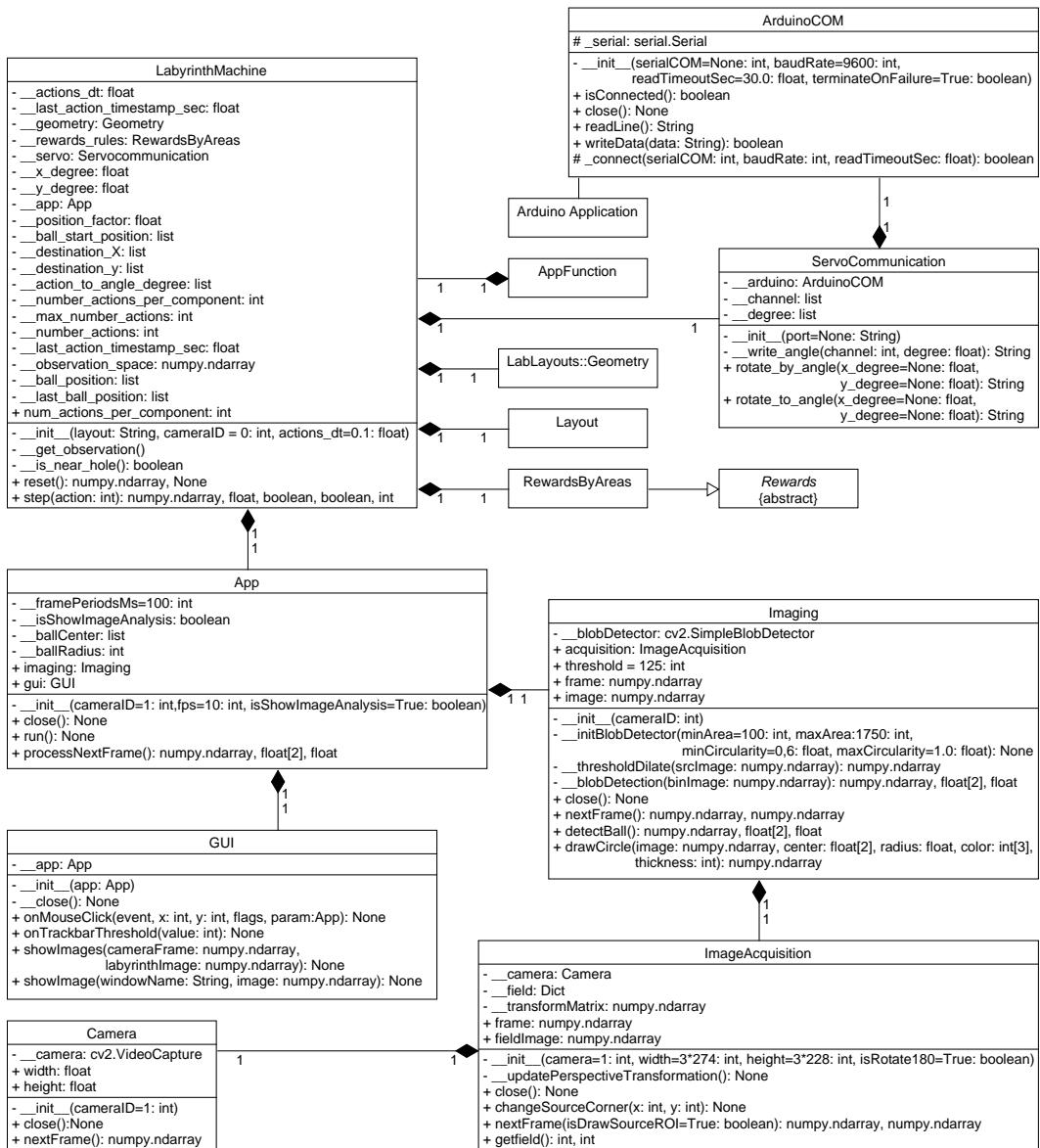


Abbildung 7.6: Klassendiagramm zum physischen Demonstrator

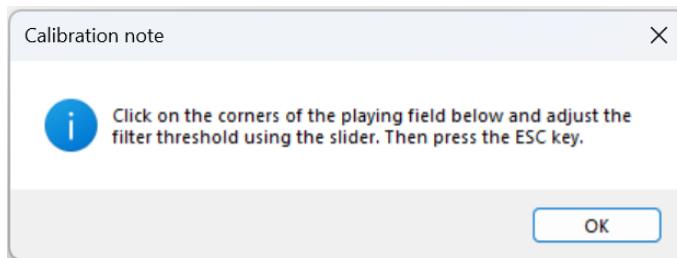


Abbildung 7.7: Kalibrationshinweis zur Bildverarbeitungskalibration

#### **Reset-Methode:**

Die Reset-Methode ermöglicht das Zurücksetzen der Umgebung auf einen definierten Anfangszustand. Dabei werden die Servos in die Mittelstellung ( $0^\circ$ ) gefahren. Anschließend erscheint eine Messagebox (siehe Abbildung 7.9), die es dem Anwender ermöglicht die Kugel vor jedem Episodenstart auf eine beliebige Startposition zu legen. Nach Bestätigung der Messagebox erfolgt das Einlesen des Kugelpositions. Anschließend wird eine Skalierung der Pixelpositions Werte auf das in der Simulation verwendete Koordinatensystem (siehe Kapitel 6.2.1) durchgeführt. Zum Schluss wird der Zustandsraum auf den Anfangszustand gesetzt.

#### **Step-Methode:**

In dieser Methode wird die jeweilige gewählte Aktion des Agenten ausgeführt und der entsprechende Servo angesteuert. Außerdem wird sichergestellt, dass das Zeitintervall zwischen zwei Aktionen eingehalten wird. Zuletzt wird der Zustandsraum aktualisiert, die Belohnung berechnet und überprüft, ob die Episode beendet ist oder abgebrochen werden soll. Da es ab und an vorkommt, dass beispielsweise in Ecken oder zum Teil an Wänden keine Kugelposition mit Hilfe der derzeitigen Bildverarbeitung ermittelt werden kann, wird in dem Fall die alte Position erneut verwendet.

### **7.2.2 Servokommunikation**

Die Servokommunikation besteht aus zwei Python-Klassen (ServoCommunication und ArduinoCOM) und einem Arduino-Programm. Die Installationsanleitung für die genutzten Bibliotheken bzw. Pakete befinden sich in Anhang A. Es ist wichtig, dass der Servomotor COM-Motor02 von Joy-IT an Kanal 0 des PWM-Treibers angeschlossen wird und der Servomotor MG996R an Kanal 1. Die minimalen und maximalen Pulslängen der Servos und auch die Mittelstellungen sind auf diese Hardwarekonfiguration

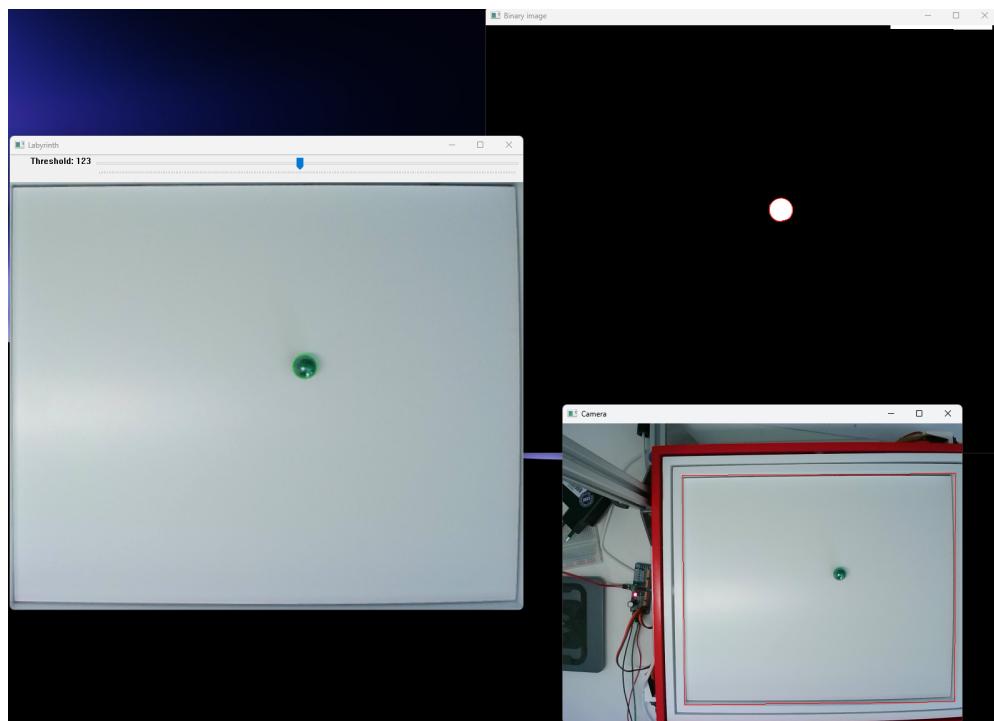


Abbildung 7.8: Kalibrierung für die Bildverarbeitung

abgestimmt. Die ermittelte Pulslänge des COM-Motor02 geht von 420  $\mu\text{s}$  bis 2540  $\mu\text{s}$  und beim Servo MG996R von 350  $\mu\text{s}$  bis 2650  $\mu\text{s}$ .

### ServoCommunication:

In der Python-Klasse ServoCommunication erfolgt die Zusammenstellung der Übertragungsdaten an den Microcontroller. Für die Kommunikation mit dem Arduino wurde die Baudrate auf 115200 gesetzt, dass bedeutet es können 115200 Bits pro Sekunde übertragen werden. Der Nachrichtenstring für die Übertragung wird wie folgt zusam-

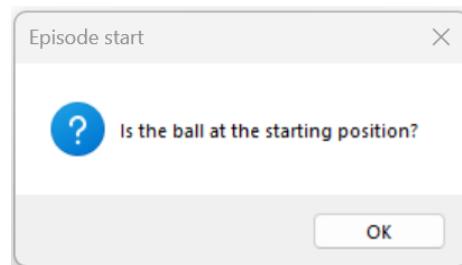


Abbildung 7.9: Bestätigung des Episodenstarts

mengesetzt: Kanal des Servos (0 oder 1), dann folgt ein Trennzeichen „;“ und am Ende die gewünschte Winkelstellung des Spielfeldes in Grad (mit einem Dezimalpunkt zur Trennung).

### **ArduinoCOM:**

Diese Klasse ermöglicht zum einen das automatische Ermitteln des seriellen COM-Ports (Communication-Port), an dem der Arduino an den Laptop angeschlossen wurde. Zum anderen können Daten mit Hilfe der Python-Bibliothek *serial* an den Arduino gesendet (*serial.Serial.write()*) und von ihm empfangen (*serial.Serial.readLine()*) werden.

### **Ardunio Applikation:**

Der Servotreiber wird mit der Standardadresse 0x40 initialisiert und die Servofrequenz wird auf 50 Hz gesetzt, da Servos typischerweise mit dieser Frequenz arbeiten. Die Applikation ermöglicht es, über den Servotreiber das gewünschte PWM-Signal zu erzeugen. Dazu wird zuerst die Nachricht eingelesen und anhand des Trennzeichens „;“ in den Kanal und die Winkelstellung unterteilt (*strtok()*). Anschließend wird der gewünschte Winkel in die entsprechende Pulslänge des PWM-Signals umgerechnet. Zum Schluss kann das PWM-Signal erzeugt werden (*pwm.writeMicroseconds(channel, pulse\_width)*). Dabei wird sichergestellt, dass die Servos innerhalb ihrer sicheren Betriebsgrenzen betrieben werden.

### **Umrechnung von Winkeln in Pulslängen für das PWM-Signal:**

Die Berechnung der Pulslänge  $t_{puls}$  für das PWM Signal aus dem gewünschten Winkel  $\Theta$  ist nachfolgend dargestellt:

$$t_{puls} = \Theta \cdot u + t_{mitte} \quad (7.2)$$

Die mittlere Pulslänge  $t_{mitte}$  ist dabei durch

$$t_{mitte} = \frac{t_{max} + t_{min}}{2} \quad (7.3)$$

gegeben.  $t_{max}$  und  $t_{min}$  sind hierbei die maximale und minimale Pulslänge des PWM-Signals für die Servos. Der Umrechnungsfaktor  $u$  gibt das Verhältnis Pulse pro Grad an und kann wie folgt ermittelt werden:

$$u = \frac{t_{max} - t_{mitte}}{\Theta_{Max}} = \frac{t_{max} - t_{mitte}}{\arcsin\left(\frac{2 \cdot h_{Max}}{l_{Spielfeld}}\right)} \quad (7.4)$$

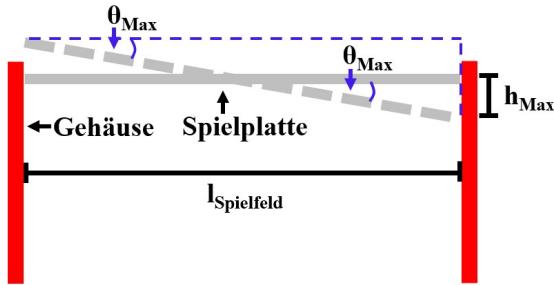


Abbildung 7.10: Bestimmung des Winkel-Puls-Verhältnisses

Nachfolgend wird nun beschrieben, wie genau die einzelnen Parameter bestimmt wurden. Ausgehend von der waagerechten Spielplatte (bei mittlerer Pulslänge) wurde die maximale Pulslänge angefahren. Die sich hierbei ergebende maximale Höhe  $h_{\text{Max}}$  wurde mit Hilfe eines Messschreibers gemessen. Bei der Drehung der x-Achse wurden eine Höhe von 8,6 mm und bei der Drehung der y-Achse eine Höhe von 5,2 mm bestimmt. Zusätzlich wurde die Spiellänge  $l_{\text{Spielfeld}}$  in x-Richtung mit 31,1 cm und in y-Richtung mit 24,5 cm bestimmt. Die geometrischen Zusammenhänge sind zur Verdeutlichung in Abbildung 7.10 dargestellt.

### 7.2.3 Bildverarbeitung

Die Bildverarbeitung besteht aus fünf Klassen: App, GUI, Camera, ImageAcquisition, Imaging. Diese Klassen wurden von Herrn Prof. Dr. Hensel, dem Betreuer dieser Arbeit, bereit gestellt. Die Klasse *Camera* dient der Erfassung von Echtzeit-Bildern einer Videokamera und nutzt hierfür die OpenCV-Bibliothek. Sie dient dazu die Verbindung zu einer Kamera herzustellen und Bilder zu erfassen. Die Extraktion eines spezifischen Bildbereiches, hier des Labyrinth-Spielfeldes, aus diesen Bildern wird mit der Klasse *ImageAcquisition* ermöglicht. Durch die Möglichkeit, den Bildbereich anzupassen und eine perspektivische Transformation durchzuführen, wird eine präzisere Extraktion und Verarbeitung von relevanten Bilddaten ermöglicht. Die Klasse *Imaging* wurde entwickelt, um die Bildverarbeitung für eine KI-Labyrinthanwendung zu ermöglichen. Dazu wird ein Blob-Detector (Blob = Binary Large Object) verwendet, um die Kugel im Labyrinth zu erkennen. Mit einem Blob-Detector können bestimmte Regionen in einem Bild, die zusammenhängende Pixelbereiche darstellen und bestimmte Kriterien wie Helligkeit, Farbe oder Form erfüllen, ermittelt werden. Im Fall der Kugelerkennung muss diese zusammenhängende Region einen gewissen Größenbereich (Minimalgröße

bis Maximalgröße) und eine bestimmte Rundheit aufweisen. Dafür wird das Bild zuerst in ein binäres (schwarz-weiß) Bild umgewandelt und die Kugel mit Hilfe des Blob-Detectors herausgefiltert. Anschließend werden die erkannten Blobs identifiziert und gekennzeichnet. Mit Hilfe der *GUI*-Klasse wird eine graphische Benutzeroberfläche und die Interaktion mit der Bildverarbeitungsanwendung ermöglicht. Hier wird die Mauser-eignisbehandlung durchgeführt. In der Ereignisbehandlung wird ein Schieberegler für die Einstellung des binären Schwellwertes und die Bildbereichseinstellung zur Anpassung des Bildausschnittes ermöglicht. Die Klasse *App* bildet das Herzstück und koordiniert die Interaktion zwischen der Bildverarbeitung und der Benutzeroberfläche.

Im Rahmen dieser Masterarbeit wurde der Bildverarbeitungsalgorithmus der Klasse *Imaging* durch einen alternativen Algorithmus ersetzt, der in Eigenleistung entwickelt wurde. Anstelle der Umwandlung in ein binäres Bild, wird das Bild nun in den HSV-Farbraum (HSV = Hue, Saturation, Value) konvertiert. Anschließend wird ein Farbfilter angewendet, der die grüne Kugel im Bild isoliert. Die gefilterten grünen Bildbereiche werden in ein Graustufenbild konvertiert, um die Blob-Erkennung zu erleichtern. Dabei muss, wie zuvor, eine zusammenhängende Region identifiziert werden, die einen bestimmten Größenbereich und eine bestimmte Rundheit aufweist. Der Einsatz des Farbfilters weist gegenüber der früheren binären Bildverarbeitung einige Vorteile auf. Zum einen ist der Algorithmus viel weniger lichtabhängig, was die Zuverlässigkeit der Erkennung unter verschiedenen Beleuchtungsbedingungen verbessert. Zum anderen ist der Algorithmus robuster gegenüber dunklen Löchern, den Linien und der Beschriftung auf der Spielfläche, die zuvor immer wieder zu Erkennungsproblemen der Kugel führten. Die Herausforderung bei der Erkennung der Kugel in Ecken oder zum Teil an den Wänden bleibt jedoch weiterhin bestehen.

# 8 Evaluierung

Dieses Kapitel bietet einen kleinen Einblick in die durchgeführten Tests zur Bestimmung der spezifischen RL-Parameter und Evaluierungen des Erlernten für die einzelnen Spielplatten. Den Abschluss bildet die Anforderungsprüfung, in der evaluiert wird, inwieweit die in Kapitel 4 aufgestellten Anforderungen umgesetzt und erfüllt wurden.

## 8.1 Tests zur RL-Parameterbestimmung

In diesem Abschnitt werden verschiedene Tests zum Vergleich und der Feinabstimmung der RL-Parameter vorgestellt. Die Tests wurden für unterschiedliche Spielplatten durchgeführt und die jeweiligen Parameterkonfigurationen hinsichtlich ihrer Stabilität und des Lernerfolges bewertet. Den nachfolgend dokumentierten Tests, sind sehr viele Trainingsdurchläufe vorangegangen, um ein Verständnis dafür zu entwickeln, wie die verschiedenen Parameter und Implementierungskonzepte das Training beeinflussen. Erst das Zusammenspiel vieler Einstellparameter ermöglicht ein erfolgreiches Training und somit das Erlernen der Fähigkeit des Agenten, die Kugel ins Ziel zu steuern. Nach der Implementierung des DQN mit der Epsilon-Greedy Policy und verschiedensten Belohnungssystemen konnte der Agent erst durch den Einsatz des Konzepts der Batch-Normalisierung (siehe Kapitel 2.3.9) erfolgreich trainiert werden. Vor Anwendung der Batch-Normalisierung ist das Problem der explodierenden Gradienten (siehe Kapitel 2.3.10) aufgetreten. Dies führte dazu, dass die Verlustfunktion nicht konvergierte und der Fehler zwischen den vorhergesagten und den tatsächlichen Werten teilweise bis auf NaN (Not a Number), über alle berechenbaren Grenzen anwuchs.

### 8.1.1 Spielplatte HOLES\_2

Nach der Implementierung des essentiellen Konzeptes Batch-Normalisierung konnte durch iteratives Testen ein besseres Verständnis für die weiteren Parameter und de-

ren Zusammenspiel entwickelt werden. Des Weiteren hat sich der Optimierer Adam in den durchgeführten Tests als sehr gut erwiesen (weitere Optimierer wurden in Kapitel 2.3.7 vorgestellt). Anhand des Labyrinthes „HOLES\_2“ werden nachfolgend einige Tests und deren Ergebnisse genauer vorgestellt. Dabei wurden die Gewichte alle 25 Episoden gespeichert, wobei insgesamt immer 1000 Episoden trainiert wurden. Die spätere Evaluierung des Erlernten wurde mit deterministischen Werten durchgeführt. Als Startbereich für jede Episode dient die richtige Startposition (als Pfeilsymbol gekennzeichnet). Da die Kugel im realen Spiel nie exakt auf eine Position gelegt werden kann, wurde der Startbereich wie folgt definiert: Startposition  $[-0,79; 9,86] \pm \text{random } 0,4$ . Für die Trainingsphase wurde der Seed-Wert der Zufallsgeneratoren für Random, NumPy und PyTorch auf 1 gesetzt um die Reproduzierbarkeit zu ermöglichen. Es werden zufällige Aktionen gewählt um die Umgebung zu erkunden sowie zufällige Datensätze auf dem Replay-Buffer gezogen. Die Initialisierung der Gewichte des neuronalen Netzes erfolgt auch zufällig. Seed setzt dabei die Startzahl für den Zufallszahlengenerator und führt dabei immer zur gleichen Sequenz von Zufallszahlen. Wobei für PyTorch keine vollständig reproduzierbaren Ergebnisse über PyTorch-Versionen oder verschiedene Hardwareplattformen hinweg garantiert werden [60]. Zudem kann die Verwendung von Seed bei PyTorch dazu führen, dass die Leistung der Modelle abnimmt und das Training länger dauert [60]. Für die Vergleichbarkeit von Parametereinstellungen bezüglich der Ergebnisse ist es allerdings trotzdem sinnvoll mit den gleichen Zufallszahlenfolgen zu arbeiten. Es wurden Testreihen in den folgenden Bereichen durchgeführt: Schichten und Neuronen des neuronalen Netzes, Verlustfunktionen, Aktivierungsfunktionen, Belohnungen, Epsilon, Lernperiode und Batchsize, Gamma, Zustandsraum und Aktionsraum, sowie Seed. Dafür wurde jeweils eine grundlegende Parameterkombination verwendet, die sich in vorherigen Tests als sehr gut erwiesen hat. Bei jedem Bereichstest wurde auf den gesammelten Erfahrungen und besten Parametern der vorherigen Tests aufgebaut. Dabei erwiesen sich die grundlegenden Parameter im Nachhinein weiterhin als die beste Einstellung. Ausgehend von den grundlegenden Parametereinstellungen

- Zustandsraum: aktuelle Position, alte Position, Feldorientierung
- Aktionsraum:  $[-1,5; -1; -0,5; 0; 0,5; 1; 1,5]$
- Verlustfunktion: MSE
- Aktivierungsfunktion: leaky\_relu
- Schichten: 128, 128

```

1 self.__destination = 600
2 self.__in_hole = -200
3 self.__near_hole = -10
4 self.__correct_direction = -1
5 self.__interim = lambda progress, areas: 3 / len(areas) * (len(areas) -
    progress)
6 self.__default = -2

```

Listing 8.1: Belohnungen HOLES\_2

- Epsilon = 1; Epsilon\_decay = 0,98; Epsilon\_min. = 0,15
- Lernperiod = 1, Batchsize = 64
- Gamma = 0,99
- Belohnung = (siehe Listing 8.1)

wurde jeweils ein Parameterbereich verändert und verglichen. Die Tests werden nachfolgend in Tabellen zusammengefasst und verglichen. Dabei wird in der Spalte „Resultat“ eine qualitative Einschätzung gegeben, wie gut die jeweilige Parametereinstellung ist. Das Symbol „++“ steht für sehr schnelles und stabiles Training, das heißt dass bei mindestens 60% der abgespeicherten Gewichtsdateien die Kugel in der Evaluation immer erfolgreich das Ziel erreicht. In der Evaluation wurden hierbei jeweils mindestens 15 Episoden gespielt. Das „+“ steht dafür, dass viele Dateien verwendet werden können, aber gegebenenfalls das Training länger gedauert hat oder sich nicht ganz so stabilisiert hat. „o“ beschreibt, dass eine bis sehr vereinzelte Dateien erzeugt wurden, bei denen die Kugel immer ins Ziel gesteuert wird. Allerdings ist keine Stabilität eingetreten bzw. die Qualität der Evaluierungsergebnisse ist sehr wechselnd. „-“ symbolisiert, dass bei der Evaluierung der Dateien keine dabei war, bei der der Agent die Kugel immer ins Ziel gesteuert hat. Für jeden dieser Tests befinden sich ein Diagramm mit der aufsummierten Belohnung über alle trainierten Epochen sowie die beste Datei zu den trainierten Gewichten auf der CD. Die Diagramme liefern dabei oft einen guten Anhaltspunkt wie stabil und erfolgreich das Training war.

### 1. Schichtentest

Als Neuronenanzahl je Schicht wird oft eine Zweierpotenz gewählt [54], deswegen wurde dies auch hierbei berücksichtigt. In der Tabelle 8.1 sind die Ergebnisse des Testes zusammengestellt. In der Spalte „Schichten“ stehen die Neuronenanzahl der verdeckten

Schichten. Die erste Zahl stellt die Neuronenanzahl der ersten verdeckten Schicht dar, die zweite Zahl die der zweiten verdeckten Schicht und so weiter.

Tabelle 8.1: Schichtentest

Schichten	beste Episode	Resultat	Kommentar
32, 16	1000	o	Sehr instabiles Lernen, einziges Evaluierungsresultat beinhaltet viele Oszillationen bevor das Ziel erreicht wird
32, 32	850	o	
128, 64	900	o	
128, 128	650	++	Alle Dateien von Episode 300-1000 führen ins Ziel (Ausnahme: 375 & 400), stabiles Lernen
256, 128	625	+	Alle Dateien von Episode 400-900 führen ins Ziel
256, 256	keine	-	Hohe Fehler zwischen vorhergesagtem und tatsächlichem Wert
512, 256	675	o	
32, 32, 16	1000	o	
32, 32, 32	1000	o	
128, 64, 8	900	o	

**Fazit:** Sind sehr wenig Neuronen vorhanden, wie bei dem Schichtentest 32, 16, dann werden die Evaluierungsergebnisse schlechter, da sehr wenige Informationen gespeichert werden können. Würde die Neuronenanzahl noch weiter verringert werden, dann kann das Labyrinth irgendwann nicht mehr bewältigt werden. Ist hingegen die Neuronenanzahl recht hoch für die Aufgabe, wie bei den Schichtentests 256, 256 oder 512, 256, dann verschlechtert sich das Ergebnis auch. Da gegebenfalls zu spezifische Zustände und daraus folgende Aktionen gelernt werden. Zu groß ausgelegte neuronale Netze neigen auch zu explodierenden Gradienten. Außerdem dauert das Training bei der erhöhten Neuronenanzahl deutlich länger, da eine größere Anzahl an Gewichten aktualisiert werden muss. Bei den Schichtentests konnten die Ansätze der Probleme mit Underfitting und Overfitting beobachtet werden (wurden in Kapitel 2.3.10 erläutert).

## 2. Verlustfunktionstest

Da es sich bei der Implementierung um ein Regressionsproblem und nicht um ein Klassifizierungsproblem handelt, wurden drei der in Kapitel 2.3.6 vorgestellten Verlustfunktionen verglichen (siehe Tabelle 8.2).

Tabelle 8.2: Verlustfunktionstest

Verlustfunktion	beste Episode	Resultat	Kommentar
HuberLoss	keine	-	
MAE	keine	-	in PyTorch L1Loss
MSE	650	++	Siehe Schichtentest 128, 128

**Fazit:** Der Agent lässt sich mit der Verlustfunktion MSE sehr gut trainieren, Huber Loss und MAE funktionieren hingegen nicht gut.

### 3. Aktivierungsfunktionstest

Im Kapitel 2.3.3 wurden verschiedenste Aktivierungsfunktionen vorgestellt. Der Vergleich von drei dieser vorgestellten Aktivierungsfunktionen ist in Tabelle 8.3 dargestellt.

Tabelle 8.3: Aktivierungsfunktionstest

Aktivierungsfunktion	beste Episode	Resultat	Kommentar
LeakyRELU	650	++	Siehe Schichtentest 128, 128
ELU	1000	o	
RELU	925	o	

**Fazit:** Leaky RELU ermöglicht sehr stabiles Lernen. In der Evaluierung ist RELU stabiler als ELU, aber instabiler als Leaky RELU.

### 4. Belohnungstest

Der Belohnungstest ist in drei Unterbereiche geteilt. Im ersten Abschnitt wird das Ziel und gegebenenfalls das Loch oder die Lochnähe belohnt. Im zweiten Abschnitt wird zusätzlich die richtige Bewegungsrichtung nach dem Konzept der Kacheln und Zwischenziele (siehe Kapitel 5.1.5) belohnt. Im dritten Abschnitt wird das Übertreten von Schwellen (siehe das Konzept in Kapitel 5.1.5) belohnt. Dabei sind die einzelnen Tests in den folgenden Tabellenspalten „Nr.“ durchnummieriert.

#### Ziel (und Loch):

Die jeweils getesteten Belohnungen sind in der Tabelle 8.4 dargestellt. Dabei steht die Spalte „destination“ für die Belohnung, die es beim Erreichen des Ziels gibt, „hole“ für das Hineinfallen in ein Loch, „near\_hole“ für Positionen die nahe am Loch sind und „else“ welche Belohnung es ansonsten gibt.

Die Auswertung und der Vergleich sind in Tabelle 8.5 zusammengefasst.

Tabelle 8.4: Belohnungen mit Ziel (und Loch)

Nr.	destination	hole	near_hole	else
1	1			0
2	500			-1
3	10000	-40000	-25	0
4	100	-50	-11	0
5	600	-200	-10	-1
6	1000	-500	-10	-1

Tabelle 8.5: Auswertung zu Ziel (und Loch)

Nr.	beste Episode	Resultat	Kommentar
1	keine	-	
2	950	o	
3	600	o	Vergleichbar mit der Belohnung der 2. Masterarbeit der Linköping University (siehe Kapitel 3.4)
4	keine	-	
5	825	o	
6	875	o	

**Fazit:** Das Ziel sollte nicht zu gering bewertet werden (vgl. 1 oder 4). Zu große Werte für die Belohnung sind aber auch hinderlich und führen gegebenenfalls zu explodierenden Gradienten. Beispielsweise wird in Konfiguration 3 die Löchernähe extrem gemieden. Dies führt zu Problemen wenn die Wege an Löchern vorbei enger werden. Aber auch zu geringe negative Belohnung der Löcher ist nicht hilfreich, da sich der Agent sonst nicht merkt, dass die Kugel nicht reinfallen soll. Die Belohnungen 5 und 6 führte zu ähnlichen guten Evaluationsergebnissen und waren besser als die vorherigen Belohnungen 1 bis 4.

### Kacheln und Zwischenziele:

Die genaue Einteilung der Kacheln und Zwischenzielpunkten ist in der Abbildung 8.1 dargestellt. Aufbauend auf den gesammelten Belohnungserfahrungen sind in Tabelle 8.6 die jeweils getesteten Belohnungen zusammengefasst. Dabei steht die Spalte „interim“ für die Belohnung, die es gibt, wenn die Kugelposition nach Ausführung der Aktion um 0,25 cm näher am Zwischenziel liegt. Die Belohnung für „correct\_direction“ gibt es, wenn die Distanz zwischen der Kugelposition und dem Zwischenziel verkürzt wurde. Das „\*“ -Symbol beschreibt die Berechnung, dass ein Kachelfortschritt anteilig mehr positive Belohnung bringt (siehe auch Listing 8.1).

In Tabelle 8.7 ist eine qualitative Auswertung dargestellt.

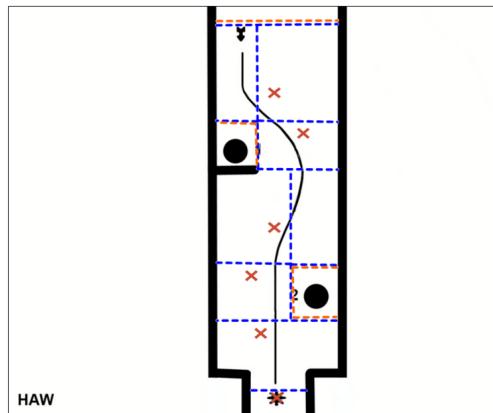


Abbildung 8.1: Kacheln und Zwischenziele der Spielplatte HOLES\_2

Tabelle 8.6: Belohnungen mit Labyrinthfortschritt

Nr.	destination	hole	near_hole	interim	correct_direction	else
7	600	-200	-10	1	1	-1
8	600	-200	-10	1		-1
9	600	-200	-10	2	1	-2
10	600	-200	-10	2	0,5	-2
11	600	-200	-10	3/*	0,5	-2
12	600	-200	-10	3/*	-1	-2
13	600	-200	-10	3/*	0	-3/*
14	600	-600	-10	3/*	-1	-2
15	800	-300	-10	5/*	-1	-4
16	1000	-500	-15	5/*	-1	-4

**Fazit:** Sehr kleine Änderungen in der Belohnungsstrategie haben schon große Auswirkungen darauf, wie stabil das Training wird und ob die Verlustfunktion konvergiert. Einen größeren Positionsfortschritt in die richtige Richtung positiver zu belohnen als eine sehr kleine Positionsänderung ist sinnvoll (Vergleich 7 und 10). Wenn geringe Bewegungen zu hoch positiv bewertet werden, kommt es öfters dazu, dass die Kugel an einer Stelle liegen bleibt (verschiedenste Zwischenevaluierungsergebnisse von 9 oder 7). Eine geringe Bewegung in die richtige Richtung sollte am besten klein negativ belohnt werden (Vergleich 11 und 12). Die Berücksichtigung des Kachelfortschritts bei der positiven Belohnungshöhe hat zu einem sehr guten und stabilen Lernergebnis geführt (siehe 12). Zu hohe oder zu geringe Belohnungen des Labyrinthfortschrittes führen zu einem verschlechterten Lernverhalten. Es lässt sich festhalten, dass das Training durch Zwischenbelohnungen stabilisiert und beschleunigt werden kann.

Tabelle 8.7: Auswertung zu Kacheln und Zwischenzielen

Nr.	beste Episode	Resultat	Kommentar
7	800	o	
8	675	o	
9	600	o	
10	925	+	Alle Dateien von Episode 450-1000 führen ins Ziel (Ausnahme: 625, 650, 950)
11	800	o	
12	650	++	Siehe Schichtentest 128, 128
13	275	o	
14	825	o	
15	850	o	
16	850	o	

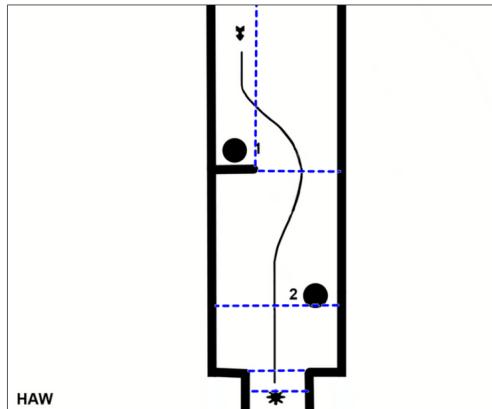


Abbildung 8.2: Schwellen der Spielplatte HOLE\_2

### Schwellen:

Die genaue Einteilung der Schwellen ist in Abbildung 8.2 dargestellt. Aufbauend auf den gesammelten Belohnungserfahrungen sind in Tabelle 8.8 die jeweils getesteten Schwellenbelohnungen zusammengefasst. Dabei steht die Spalte „r\_threshold“ für die Belohnung, die es gibt, wenn die Kugel eine Schwelle in richtiger Bewegungsrichtung überquert, und „f\_threshold“ für die Überquerung der Schwelle in falscher Richtung.

In Tabelle 8.9 ist die qualitative Auswertung der Schwellenbelohnung dargestellt.

**Fazit:** Mit Hilfe von Schwellen kann ein ähnlich gutes Ergebnis erzielt werden wie mit der Fortschrittsbelohnung basierend auf den Kacheln und Zwischenzielen. Auch hier lässt sich erkennen, dass kleine Änderungen die Lernqualität erheblich beeinflussen und instabilisieren können.

Tabelle 8.8: Belohnungen mit Schwellenbelohnung

Nr.	destination	hole	near_hole	r_threshold	f_threshold	else
17	600	-200	-10	5	-5	-1
18	600	-200	-10	5		-2
19	600	-200	-10	7	-7	-2
20	800	-300	-10	7	-7	-1

Tabelle 8.9: Auswertung zu Schwellen

Nr.	beste Episode	Resultat	Kommentar
17	600	++	Alle Dateien von Episode 350-1000 führen ins Ziel (Ausnahme: 475, 400)
18	600	o	
19	550	o	
20	825	o	

## 5. Epsilontest

Der Vergleich verschiedener Epsilon Strategien ist der Tabelle 8.10 zu entnehmen.  $\epsilon$  beschreibt die Erkundungsrate (siehe Kapitel 2.4.1). Nach der Epsilon-Reduzierungs-Strategie (siehe Kapitel 5.1.6) wird bei fortschreitendem Training  $\epsilon$  schrittweise reduziert. Dieser Reduzierungsfaktor ist der Spalte „decay“ zu entnehmen. Die Spalte „ $\epsilon$ “ beschreibt den Startwert für die Erkundungsrate und in der Spalte „min.“ ist die untere Grenze definiert, auf die das Epsilon sinken kann.

Tabelle 8.10: Epsilontest

Nr.	$\epsilon$	decay	min.	beste Episode	Resultat	Kommentar
1	1	0,98	0,15	650	++	Siehe Schichtentest 128,128
2	1	0,99	0,01	475	+	Alle Dateien von Episode 425-1000 führen ins Ziel (Ausnahme: 650)
3	0,1		0,1	600	o	
4	0,7	0,95	0,3	600	o	
5	1	0,95	0,05	500	o	
6	0,7	0,95	0,1	450	++	Alle Dateien von Episode 200-1000 führen ins Ziel (Ausnahme: 225,375,575,600)

**Fazit:** Die Epsilon-Reduzierungs-Strategie funktioniert besser als das  $\epsilon$  konstant zu lassen (Vergleich 3 und 6). Auch kleinere Änderungen in der Reduzierungsrate haben

## 8 Evaluierung

---

einen Einfluss auf die Stabilität des Lernens. Außerdem darf  $\epsilon$  nicht zu klein werden (siehe 5). Aber auch ein zu großes  $\epsilon$  (siehe 4) destabilisiert das Lernen. Bei einem zu groß bleibendem  $\epsilon$  entstehen recht große Fehlerwerte (großer Unterschied zwischen vorhergesagtem und tatsächlichen Q-Wert).

### 6. Lernperiod-/Batchsizetest

Bei diesem Test wird das Konzept des Replay-Buffers verwendet (siehe Kapitel 5.1.6). Der Vergleich verschiedener Einstellmöglichkeiten ist in Tabelle 8.11 zusammengefasst. Wenn über 200 Erfahrungen im Replay-Buffer gespeichert sind (es müssten mindestens Batchsize Erfahrungen vorhanden sein), wird nach „Lernperiod“ Aktionen ein Minibatch (siehe Spalte „Batchsize“) aus dem Replay-Buffer gezogen und anhand dessen das neuronale Netz trainiert.

Tabelle 8.11: Lernperiod-/Batchsizetest

Nr.	Lernperiod	Batchsize	beste Episode	Resultat	Kommentar
1	1	64	650	++	Siehe Schichtentest 128,128
2	2	128	1000	o	
3	1	32	600	+	
4	1	128	300	o	
5	2	64	575	o	

**Fazit:** Nur nach jeder zweiten durchgeföhrten Aktion das neuronale Netz zu trainieren funktioniert nicht so gut, wie nach jeder Aktion (Vergleich 1 mit 2 und 5). Für diese Aufgabe funktioniert die Batchsize von 64 sehr gut. Eine größere Batchsize hat nur zu einer sehr kurzen Stabilität geföhrt und die Evaluierungsergebnisse wurden schnell wieder schlechter, das Netz wurde übertrainiert.

### 7. Gammatest

Der Diskontierungsfaktor  $\gamma$  beschreibt, wie weit zukünftige Belohnungen abgewertet werden (siehe Kapitel 2.4.2). In Tabelle 8.12 ist ein kleiner Vergleich verschiedener Gamma-Werte zusammengefasst.

Tabelle 8.12: Gammatest

Nr.	Gamma	beste Episode	Resultat	Kommentar
1	0,99	650	++	Siehe Schichtentest 128,128
2	0,8	475	o	
3	0,995	keine	-	
4	0,98	350	o	

**Fazit:** Ein Wert von  $\gamma = 0,99$  hat sich als sehr gut bewährt (siehe 1). Eine geringfügige Erhöhung oder Verringerung von Gamma führte zu erheblich schlechteren oder gar keinen gelernten Evaluierungsergebnissen.

## 8. Zustandsraumtest

Der Vergleich zweier Zustandsräume ist in der Tabelle 8.13 dargestellt.

Tabelle 8.13: Zustandsraumtest

Nr.	Zustandsraum	beste Episode	Resultat
1	aktuelle Position, alte Position, Feldorientierung	650	++
2	aktuelle Position, Geschwindigkeit, Feldorientierung	450	++

**Fazit:** Die beiden getesteten Zustandsräume führen schnell zu sehr stabilen Evaluierungsergebnissen. Der erste Test entspricht dem Schichtentest 128,128 und beim zweiten Test führen alle Dateien von Episode 250-1000 immer ins Ziel (Ausnahme 325, 500, 600).

## 9. Aktionsraumtest

In der Tabelle 8.14 wurden verschiedene große Aktionsräume verglichen. Dabei beschreibt eine Aktion eine konkrete Winkelstellung. Für die Drehung in x-Richtung und in die y-Richtung wurden immer die gleichen Winkelstellungen verwendet. In der Tabellen Spalte „Aktionsraum“ sind diese konkret möglichen Winkelstellungen beider Achsen beschrieben.

Tabelle 8.14: Aktionsraumtest

Nr.	Aktionsraum	beste Episode	Resultat
1	[-1,5;-1;-0,5;0;0,5;1;1,5]	650	++
2	[-1,25;-1;-0,5;-0,25;0;0,25;0,5;1;1,25]	800	o
3	[-1,5;-1,25;-1;-0,5;-0,25;0;0,25;0,5;1;1,25;1,5]	825	o
4	[-1,75;-1,5;-1,25;-1;-0,5;-0,25;0;0,25;0,5;1;1,25;1,5;1,75]	900	o
5	[-2,5;2;-1,5;-1;-0,5;0;0,5;1;1,5;2;2,5]	1150	o

**Fazit:** Die Aktionsräume 3 bis 5 wurden 1200 Episoden lang trainiert. Je mehr mögliche Aktionen zur Auswahl stehen, desto instabiler wird das Training. Je größer die möglichen Feldorientierungswinkel werden (Vergleich 3 und 5), desto länger dauert es, bis sich die Evaluierungsergebnisse stabilisieren und die Kugel immer ins Ziel gesteuert wird. Als

Maßnahme um größere Aktionsräume besser zu trainieren, könnte die Episodenanzahl oder die Neuronenanzahl erhöht werden.

## 10. Seedtest

In der Tabelle 8.15 werden einige Startwerte für Seed verglichen. Des Weiteren wurden

Tabelle 8.15: Seedtest

Seed	beste Episode	Resultat	Kommentar
1	650	++	Siehe Schichtentest 128,128
0	900	o	
2	175	+	
5	500	+	Alle Dateien von Episode 325-925 führen ins Ziel (Ausnahme: 475, 800)
42	600	o	

Tests ohne gesetztem Seed durchgeführt, dabei wurde die Neuronen- und Schichtenanzahl verändert (ähnlich wie bei 1. Schichtentest). Eine kleine Auswahl dieser Tests ist in der Tabelle 8.16 dargestellt, weitere durchgeführte Tests sind auf der CD einzusehen. Dabei wurde mit einer bis drei verdeckte Schichten getestet.

Tabelle 8.16: Test ohne Seed

Schichten	beste Episode	Resultat	Kommentar
32,32	700	++	Alle Dateien ab Episode 400 führten ins Ziel
256,128	550	++	Alle Dateien ab Episode 275 führten ins Ziel
128,128	400	o	
256,256	500	++	Alle Dateien ab Episode 275 führten ins Ziel

**Fazit:** Die Lerndauer und Ergebnisse werden erheblich durch die Zufallszahlenreihenfolge beeinflusst. Bei einem Seed von 1 funktionieren zwei verdeckte Schichten mit jeweils 128 Neuronen sehr gut, wohin gegen beim Testen ohne Seed viel bessere Ergebnisse mit anderen Schichtengrößen erzielt werden konnten. Zu Beginn dieses Kapitels wurde darauf hingewiesen, dass PyTorch keine Reproduzierbarkeit über verschiedene CPU-Ausführungen hinweg garantiert [60]. Dieses Verhalten konnte beobachtet werden. Trotz des selben Programms und der gleichen PyTorchVersion „2.3.1+cpu“ kann das Ergebnis auf verschiedenen Laptops abweichen. Auf dem selben Laptop ist die Reproduzierbarkeit hingegen immer gegeben.

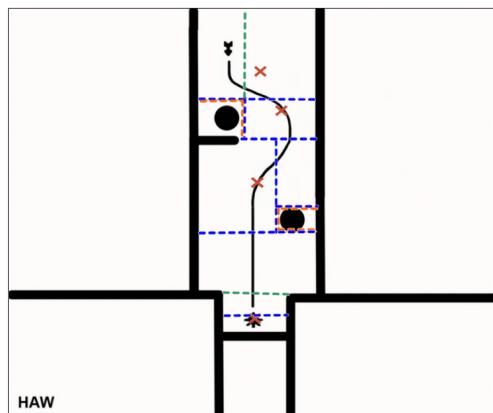


Abbildung 8.3: Kacheln und Zwischenziele der HOLES\_2\_VIRTUAL Spielplatte

### 8.1.2 Spielplatte HOLES\_2\_VIRTUAL

Aufbauend auf den gesammelten Erfahrungen sowie den Erkenntnissen aus den dokumentierten Tests mit der Spielplatte „HOLES\_2“, wurde die sehr ähnlich aufgebaute Spielplatte „HOLES\_2\_VIRTUAL“ trainiert.

#### Platzierungstest der Kacheln und Zwischenpunkte:

Es wurde mit den grundlegenden Parametereinstellungen aus dem vorherigen Kapitel 8.1.1 trainiert. Dabei wurden zwei verschiedene Kacheleinteilungen getestet, die in der Abbildung 8.3 dargestellt sind. Als erstes wurde nur mit den blauen Kacheln trainiert und bei einem zweiten Test wurden diese um die dunkel grünen Kacheln und deren Zwischenzielpunkte erweitert. Die Auswertung dieser Tests ist in der Tabelle 8.17 zusammengefasst.

Tabelle 8.17: Platzierungstest Kacheln und Zwischenziele

Kacheln	beste Episode	Resultat	Kommentar
blau	775	o	
grün, blau	625	++	alle Dateien ab Episode 350 führten ins Ziel

**Fazit:** Die Platzierung und Anzahl der Kacheln hat Auswirkungen darauf, wie stabil und wie schnell das Training des Agenten verläuft. Somit müssen diese mit Bedacht platziert werden. Des Weiteren wurden weitere Belohnungen getestet, es war aber keine besser als die, die für die zuvor vorgestellt Spielplatte „HOLES\_2“ verwendet wurde (siehe Listing 8.1).

### 8.1.3 Spielplatte HOLES\_0

Beim Testen mit der Spielplatte „HOLES\_0“ wurde auf den bisher gesammelten Erfahrungen aufgebaut, für ein stabiles und erfolgreiches Training mussten allerdings einige Parametereinstellungen angepasst werden. Beim Training wurden die Gewichte wieder alle 25 Episoden gespeichert und seed = 1 gesetzt. Bei dieser Spielplatte wird eine Episode abgebrochen, wenn die maximale Anzahl an getätigten Aktionen erreicht wurde oder der kumulierte Reward über 2000 liegt. Anders als bei den vorherigen Labyrinthplatten führt die Zielerreichung (innerster schwarzer Kreis) zu keinem Ende der Episoden. Als gut funktionierende Parametereinstellungen haben sich nach einer Vielzahl von Tests folgende Konfigurationen herausgestellt:

- Schichten: 512, 128
- Epsilon = 1; Decay = 0,9; Min = 0,1
- Aktionsraum: [-1; -0,5; 0; 0,5; 1]
- Zustandraum: aktuelle Position, alte Position, Feldorientierung
- Innen Startposition random:  $x = \pm 6,06$  und  $y = \pm 5,76$
- Außen Startposition random:  $x = \pm 13,06$  und  $y = \pm 10,76$
- Evaluierung mit 0% Zufall
- Verlustfunktion: MSE
- Aktivierungsfunktion: leaky\_relu

Um das Training erheblich zu beschleunigen und um bessere Lernerfolge bzw. Evaluierungsergebnisse zu erhalten, wurde das Training in zwei aufbauende Trainings unterteilt. Zuerst wird nur von Startpositionen näher in der Mitte trainiert (Innen), bis sich dort gute Evaluierungsergebnisse einstellen. Anschließend wird diese Gewichtsdatei geladen und es wird mit Startpositionen auf der gesamten Spielplatte weitertrainiert. Um Stabilität in das Training reinzubringen musste die Neuronenanzahl im Vergleich zur „HOLES\_2(\_VIRTUAL)“ Spielplatte erhöht werden und der Epsilon Reduzierungsfaktor sowie der minimale Epsilonwert angepasst werden. Bei trainierten Agenten hat dieser nur die Aktionen mit kleinen Winkelstellungen gewählt, da sonst die Gefahr zu hoch erschien, dass die Kugel eine zu hohe Geschwindigkeit aufbaut und immer wieder

```

1 self.__destination = 600
2 self.__default = -2
3 self.__interim_dict = {
4     6: 100,
5     5: 25,
6     4: 2,
7     3: -0.2,
8     2: -0.4 }
9 self.__default_dict = {
10    6: -0.2,
11    5: -0.4 }
```

Listing 8.2: Belohnungen HOLES\_0

über den Mittelpunkt hinweg rollt. Deshalb wurden im Laufe der Zeit der Aktionsraum generell auf kleine Werte beschränkt, um das Training zu beschleunigen. Auch bei dieser Spielplatte wurden viele Tests zum Belohnungssystem durchgeführt. Vereinzelt dokumentierte Tests sind auf der CD enthalten. Die Verwendung der Kachelbelohnung bzw. einer Belohnung, die beim Übertreten der einzelnen gezeichneten Kreissegment gegeben wird, und gegebenfalls in Kombination mit einer positiven Richtungsbelohnung im innersten Kreis, führte zu keinen guten Ergebnissen. Die Kugel hat hohe Bewegungsgeschwindigkeiten angenommen und schießt in den Evaluierungen immer wieder deutlich über die Mitte hinweg. Die beste Belohnungsstrategie für die zwei aufbauenden Trainings ist in Listing 8.2 dargestellt. Die erste Ziffer beschreibt dabei den progress bzw. in welchem Kreisegment sich die Kugel aufhält. Dabei entspricht 6 dem innersten Kreis und 1 dem äußersten Bereich außerhalb der gezeichneten Kreise. Die zweite Ziffer beschreibt die entsprechende Belohnungshöhe. Beim Testen hat sich außerdem herausgestellt, dass die Kugel öfters nicht den letzten Schritt zum schwarzen kleinen Zielpunkt schafft, sondern irgendwo im innersten Kreis bleibt. Um diesen letzten Schritt zu ermöglichen, wurde die Erfahrung des Zielerreichens beim Trainieren doppelt in den Replay-Buffer gespeichert. Somit erhöht sich die Wahrscheinlichkeit, dass aus dieser selten auftretenden Erfahrung mehr gelernt wird. Da das Spielfeld im Vergleich zu dem kleinen Punkt in der Mitte sehr groß ist, kommt die Kugel nicht so oft an diese Position. Wenn das Ziel aber zu oft in den Replay-Buffer gespeichert wird oder direkt zum Training verwendet wird, dann lernt der Agent den Weg bis in die Mitte zu wenig, was auch nicht zum Erfolg führt.

```
1 self.__destination = 600
2 self.__default = -1
3 self.__interim_dict = {
4     4: 100,
5     3: 20,
6     2: 2 }
7 self.__default_dict = {
8     4: -0.2,
9     3: -0.4 }
```

Listing 8.3: Belohnungen HOLES\_0\_VIRTUAL

### 8.1.4 Spielplatte HOLES\_0\_VIRTUAL

Die trainierte Gewichtsdatei der Spielplatte „HOLES\_0“ kann auch bei der Spielplatte „HOLES\_0\_VIRTUAL“ verwendet werden (siehe Kapitel 8.1.3). Bei beiden Spielplatten besteht das Ziel darin, die Kugel in die Mitte zu manövrieren und dort zu balancieren. Zusätzlich wurde noch mit einem modifizierten Zustandsraum bestehend aus der aktuellen Position, der Geschwindigkeit und der Feldorientierung trainiert. Dabei wurde ein Netz mit zwei verdeckten Schichten mit jeweils 128 Neuronen verwendet. Bei diesem Training wurde kein Seed verwendet. Die beste Belohnungsstrategie für dieses wieder zweigeteilte Training ist in Listing 8.3 dargestellt. Zuerst wird von Startpositionen, die näher an der Mitte liegen, trainiert (Innen), bis sich dort gute Evaluierungsergebnisse einstellen. In einem zweiten Schritt wird ausgehend von diesen Gewichtswerten das Training auf Startpositionen der gesamte Spielplatte ausgeweitet. Um die finale Annäherung an den Mittelpunkt zu optimieren, wurde die Erfahrung des Zielerreichens beim Trainieren wieder doppelt in den Replay-Buffer gespeichert. Das Evaluierungsergebnis für den Zustandsraum, der die Geschwindigkeit an Stelle der letzten Position nutzt, ist geringfügig besser. Die Kugel erreicht dabei etwas öfter den Mittelpunkt und rollt nicht im innersten Kreissegment hin und her. Beide Trainingsdateien befinden sich auf der CD.

### 8.1.5 Spielplatte HOLES\_8

Aufbauend auf den Erkenntnissen aus dem Training der Spielplatten „HOLES\_2(\_-VIRTUAL)“ wurde die Labyrinthplatte mit 8 Löchern trainiert. Das Training wurde vom Ziel in Richtung des Starts aufgebaut, dabei wurden sukzessiv Startpositionen näher in Richtung des Starts verschoben. Aufgrund des höheren Schwierigkeitsgrades im

Tabelle 8.18: Parameter des neuronalen Netzes der HOLES\_8 Spielplatte

Schicht	Neuronen	Parameter
Eingangsschicht	6	
1.	2048	14.336
2.	1024	2.098.176
3.	256	262.400
Ausgangsschicht	10	2.570
Gesamte Parameteranzahl:		2.377.482

Vergleich zur „HOLES\_2(\_VIRTUAL)“ -Labyrinthplatte musste die Architektur des neuronalen Netzes vergrößert werden. Die Netzgröße wurde schrittweise erhöht, sobald ein erfolgreiches Training nicht mehr möglich war. Dies diente dazu, das Netzwerk so klein wie möglich, aber dennoch ausreichend leistungsfähig zu gestalten, um die Trainingszeit zu minimieren. Je größer das Netz, desto länger dauern die Berechnungen zur Aktualisierung der Gewichte und Biases. Derzeit ist das Wissen des Agenten auf zwei neuronale Netze der gleichen Größe (siehe Tabelle 8.18) aufgeteilt. In der Spalte Parameter ist die Anzahl der trainierbaren Gewichte und Biases des Netzes aufgeführt. Beim Training dieser Labyrinthplatte konnte festgestellt werden, dass das Netz im Schnitt ohne Verwendung von Seed etwas kleiner (weniger Neuronen) gestaltet werden konnte als mit Seed. Allerdings wurde trotzdem ein Seed gesetzt (Seed=1) um eine Reproduzierbarkeit zu ermöglichen und die Parameterkonfigurationen besser auf einander abstimmen zu können. Die erste und die zweite Labyrinthhälfte wurden separat trainiert, dabei war das jeweilige Training mit einem herkömmlichen Laptop sehr zeitintensiv. In dieser Arbeit wurde ein Laptop mit einer CPU des Typs Intel Core i7 150U 1.80 GHz und 16GB RAM, ohne GPU eingesetzt. Das Training einer Labyrinthhälfte, die 4000 Episoden umfasste, benötigte hierbei mehr als 20 Stunden Rechenzeit. Um das Training mit einem gesamten Netz zu ermöglichen müssten die erlernbaren Parameter verdoppelt werden. Dies würde beispielsweise einem Netz mit drei verdeckten Schichten und den Größen 2048, 2048 und 256 Neuronen entsprechen (Parameteranzahl = 4.737.802). Nach ausgiebigem iterativen Testen wurde eine gemeinsame Belohnungsstrategie entwickelt, die für beide Labyrinthhälften geeignet ist (siehe Listing 8.4). Dabei wurde der erste Kachel- und Zwischenzielentwurf (siehe Konzeptkapitel 5.1.5 und Abbildung 5.1) erweitert. Diese Erweiterung ist in Abbildung 8.4 dargestellt. Dabei wurden kleinere Kacheln beispielsweise in der Ziel- und Startgraden festgelegt. Dadurch erhält der Agent schneller Erhöhungen in der Belohnung, wodurch bessere Trainingsergebnisse erzielt werden konnten. Während des Trainings trat häufiger das Problem auf, dass die Kugel in Ecken

## 8 Evaluierung

---

```
1 self.__destination = 1000
2 self.__in_hole = -300
3 self.__near_hole = -15
4 self.__correct_direction = -1
5 self.__interim = lambda progress, areas: 6 / len(areas) * (len(areas) -
    progress)
6 self.__default = -2
```

Listing 8.4: Belohnungen HOLES\_8

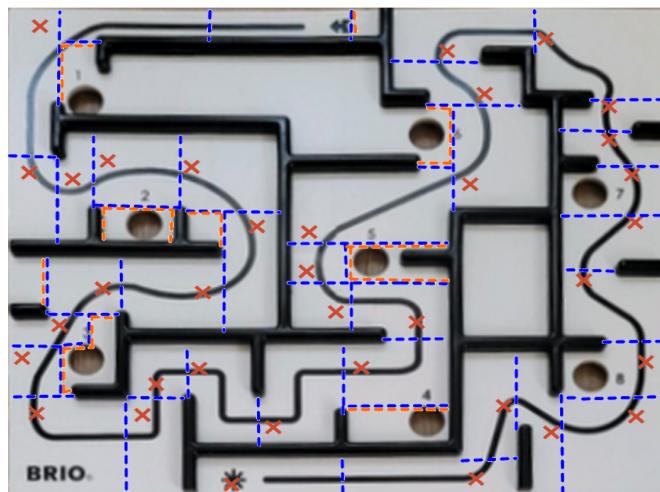


Abbildung 8.4: Kacheln und Zwischenziele der Spielplatte HOLES\_8

hängen blieb. Hieraus entstand die Idee die Ecken (Kollision mit zwei Wänden) negativ zu belohnen, wenn sich die Kugel dabei nicht oder nicht in die richtige Richtung bewegt hat. Tests mit dieser Belohnungsstrategie zeigten jedoch schnell, dass diese negative Belohnung das Training verschlechterte, weshalb dieser Ansatz verworfen wurde.

Nach zahlreichen weiteren Tests erwiesen sich zudem folgende Parametereinstellungen als besonders geeignet:

- Epsilon = 1; Decay = 0,99; Min = 0,15
- Aktionsraum: [-1; -0,5; 0; 0,5; 1]
- Zustandsraum: aktuelle Position, alte Position, Feldorientierung
- Startposition jeweils mit random  $\pm 0,4$  in [x, y]-Richtung:
  - erste Labyrinthhälfte: [[0,13; 10,53]; [-12,85; 3,92]; [-3,8; 1,29]; [-9,8; -1,49]; [-12,58; -7,03]; [-5,82; -5,23]]

- zweite Labyrinthhälfte: [[4,54; 9,6]; [-1,2; 1,14]; [3,35; -2,99]; [0,94; -5,23]; [-5,82; -5,23]; [-12,6; -7,03]; [-11,63; -3,18]]
- Evaluierung mit 0% bis 5% Zufall
- Verlustfunktion: MSE
- Aktivierungsfunktion: leaky\_relu

Es ist wichtig zu beachten, dass die Wahl der Startpositionen erheblichen Einfluss auf den Trainingserfolg hat. Besonders die Platzierung der Startpositionen im Bereich des dritten Lochs waren entscheidend dafür, ob der Agent lernte, die Kugel erfolgreich daran vorbei zu steuern oder nicht. Es wurde mit verschiedensten Startpositionen trainiert, um eine gleichmäßige Erkundung der gesamten Labyrinthhälfte sicherzustellen, so dass dem Problem mit der ungleichmäßigen Verteilung der Eingabedaten (siehe Kapitel 2.3.10) entgegengewirkt wird. Für das Training der ersten Labyrinthhälfte wurde zudem der Zielbereich vorverlegt um dem Agenten eine Zielbelohnung geben zu können. Der Versuch das ganze Labyrinth in einem einzigen Training und einem neuronalen Netz zu trainieren führte selbst bei 5500 Episoden zu keinem befriedigenden Ergebnis. Auch die „HOLES\_0(\_VIRTUAL)“ -Spielplatte mussten in zwei Trainingseinheiten unterteilt werden, um innerhalb eines begrenzten Zeitrahmens und begrenzter Episodenanzahl erfolgreiche Ergebnisse zu erzielen. Um erfolgreiches Training der „HOLES\_8“ Spielplatte mit einem gesamten Netz zu ermöglichen müsste somit das Training auch in aufbauende Trainings unterteilt werden. Aufgrund der zeitlichen Beschränkung der Masterarbeit und des sehr zeitaufwendigen Trainings bei halb so vielen Parametern (Gewichte, Bias) wurde dies jedoch nicht mehr weiterverfolgt und trainiert. Derzeit wird in der Evaluierung automatisch zwischen den beiden neuronalen Netzen umgeschaltet, was problemlos funktioniert, da beide Netze mit einigen überlappenden Kachelabschnitten des Labyrinths trainiert wurden. Bei der Evaluierung der trainierten RL-Agenten wurde zwischenzeitlich eine kleine Zufälligkeit (bis zu 5%) in die Aktionswahl eingebracht. Dadurch konnten bessere Evaluationsergebnisse erzeugt werden in dem verhindert werden konnte, dass die Kugel in suboptimalen Zuständen hängen bleibt. Bei den zuletzt trainierten neuronalen Netzen ist in der Evaluierung mit 0% Zufall kein nennenswerter Unterschied zu einer Aktionsauswahl mit 5% Zufall festzustellen. Die trainierten neuronalen Netze sowie deren kumulierte Belohnungsdiagramme vom Training sind auf der CD einzusehen.

### 8.1.6 Spielplatte HOLES\_21

Des Weiteren wurde noch die Spielplatte „HOLES\_21“ trainiert. Dieses Training wurde auch wie bei der vorherigen Labyrinthplatte auf mehrere Netze aufgeteilt, zwischen denen umgeschaltet wird. Nach einigen Tests erwiesen sich folgende Parametereinstellungen als besonders geeignet:

- Epsilon = 1; Decay = 0,99; Min = 0,15
- Aktionsraum: [-1; -0,5; 0; 0,5; 1]
- Zustandsraum: aktuelle Position, alte Position, Feldorientierung
- Startposition jeweils mit random  $\pm 0,4$  in [x, y]-Richtung:
  - erstes Labyrinthdrittel: [[3,2; 10,47]; [-8,43; 5,14]; [-6,53; -0,38]; [-10,6; -7,25]; [-3,59; -2,53]]
  - zweites Labyrinthdrittel: [[-1,57; -6,2]; [-0,31; -1,35]; [-2,83; 7,42]]
  - drittes Labyrinthdrittel: [[8,77; 3,87]; [5,94; -2,34]; [8,03; -9,03]; [12,68; 5,96]; [5,74; 6]]
- Zielbereiche  $[x_{min}; x_{max}]; [y_{min}; y_{max}]$ :
  - erstes Labyrinthdrittel: [[-0,94; 2,49]; [-8,32; -4,29]]
  - zweites Labyrinthdrittel: [[4,47; 9,27]; [7,07; 9,49]]
  - drittes Labyrinthdrittel: [[-4,2; -3,3]; [-11,4; -8,91]]
- Evaluierung mit 0% Zufall
- Verlustfunktion: MSE
- Aktivierungsfunktion: leaky\_relu
- Belohnungen: siehe Listing 8.5

Die Trainingsdateien sowie die einzelnen Diagramme der kumulierten Belohnung über die Episoden hinweg sind auf der CD hinterlegt.

```

1 self.__destination = 1500
2 self.__in_hole = -300
3 self.__near_hole = -15
4 self.__correct_direction = -1
5 self.__interim = lambda progress, areas: 9 / len(areas) * (len(areas) -
    progress)
6 self.__default = -2

```

Listing 8.5: Belohnungen HOLES\_21

### 8.1.7 Weitere Testerfahrungen

In den zuvor beschriebenen Tests wurde immer mit einer Lernrate von 0,0005 trainiert. Eine erhöhte Lernrate begünstigt das Problem der explodierenden Gradienten und verschlechtert das Lernverhalten. Außerdem ist eine gute Gewichtsinitialisierung des neuronalen Netzes wichtig. Eine falsche oder ungünstige Initialisierung begünstigt auch das Problem der explodierenden Gradienten. Im Rahmen dieser Arbeit wird die He-Initialisierung verwendet, welche speziell für die ReLU-Aktivierungsfunktion und seine Varianten entwickelt wurde. Des Weiteren konnten explodierende Fehler zwischen tatsächlichem und vorhergesagtem Q-Wert im Rahmen der RL-Parameterbestimmungen beobachtet werden. Dabei lässt sich festhalten, dass die Verlustfunktion eher nicht mehr konvergiert, wenn diese Fehlerwerte über eine Million anwachsen. Als Folge sollten Parameteranpassungen getätigt und von vorne trainiert werden.

## 8.2 Anforderungsprüfung

In den beiden nachfolgenden Unterkapiteln werden die Anforderungen, die in Kapitel 4 für die virtuelle Umgebung und den physischen Demonstrator aufgestellt wurden, überprüft.

### 8.2.1 Virtuelle Umgebung

In diesem Unterkapitel wird für die Anforderungen an die virtuelle Umgebung aus Kapitel 4.3.2) überprüft, inwiefern diese *erfüllt* oder *teilweise erfüllt* sind.

**VU-F1:** *Erfüllt*: Für ein realistisches Verhalten der Kugelbewegung wurde die Rollreibungszahl und die Stoßzahl mit Hilfe des physischen Demonstrators bestimmt. Anhand

der Labyrinthplatte „HOLES\_21“ wurde für die Gerade vom Startpunkt bis zur Wand bei einem Winkel von  $2,1^\circ$  in x-Richtung in der Realität eine Zeit von  $\Delta T = 0,96$  s und in der Simulation eine Zeit von  $\Delta T = 0,93$  s ermittelt (die entsprechenden Videos befinden sich auf der CD). Der Versuchsaufbau ist in Abbildung 6.9 in Kapitel 6.3.2 einsehbar. Das restliche Kugelverhalten wurde mit Hilfe von detaillierten physikalischen Zusammenhängen beispielsweise im Bereich der Kollisionen mit Wänden oder Ecken berechnet. Abweichungen zur Realität sind beispielsweise durch leicht gebogene oder unebene Spielplatten möglich. Zudem wurde die Rollreibung anhand der BRIO Labyrinthplatte bestimmt, die selbst designten Spielplatten bestehen aus einem anderen Grundmaterial, auf dem die Kugel ein etwas anderes Rollverhalten aufweist. Zudem stellt der schwarze Druck auf den selbst designten Spielplatten jeweils eine Erhöhung da, was das Rollverhalten verändert.

**VU-F2:** *Erfüllt:* Nachdem eine Kugel in ein Loch gefallen ist und die gewünschte Episodenanzahl noch nicht erreicht wurde wird das Spiel erneut gestartet, um das Training des Agenten fortzuführen.

**VU-F3:** *Erfüllt:* Es wurden sechs verschiedene Spielumgebungen bzw. Spielfelder realisiert. Darunter zwei reale BRIO Labyrinthplatten (HOLES\_8 und HOLES\_21) sowie zwei einfache selbstdesignete Labyrinthplatten mit horizontal und vertikalen Wänden, Löchern sowie den Start- und Zielpunkten und deren Pfad dazwischen. Zusätzlich wurden zwei Spielplatten realisiert, bei denen das Ziel darin besteht die Kugel in der Mitte zu balancieren. Somit weicht das Design von den anderen Labyrinthplatten ab.

**VU-F4:** *Erfüllt:* In der Simulation prallt die Kugel an einer Wand ab, zudem verschwindet die Kugel, wenn sie in ein Loch fällt.

**VU-F5:** *Erfüllt:* Die Drehgeschwindigkeit der Servos bzw. die reziproke Winkelgeschwindigkeit von  $0,17$  s/ $60^\circ$  wurde in der Simulation mit berücksichtigt. Ergänzend wurden verschiedenste Abschätzungen gemacht (siehe Kapitel 6.4.3), sodass die simulative Drehbewegung für die Berechnung der Kugelphysik dem realen System sehr nah kommt.

**VU-F6:** *Erfüllt:* Die sechs realisierten Spielfelder wurden trainiert und konnten mit Hilfe von KI beherrscht werden. Die Spielfelder umfassen die „HOLES\_0(\_VIRTUAL)“, die „HOLES\_2(\_VIRTUAL)“, „HOLES\_8“ sowie die „HOLES\_21“ Spielplatte. Zu jeder beherrschten Spielplatte befindet sich ein Video auf der CD.

**VU-F7:** *Erfüllt:* Die Parameter des RL-Algorithmus können zum Weitertrainieren in der realen Umgebung verwendet werden. Die gespeicherten Gewichtsdateien der trainierten Agenten aus der Simulation können für das Training am physischen Demonstrator verwendet werden. Um das Weitertrainieren zu ermöglichen wurde deshalb auch die vorherige Position, anstelle der Geschwindigkeit in den Zustandsraum aufgenommen. Die vorherige Position ist sehr viel einfacher zu ermitteln bzw. zu speichern, als die Geschwindigkeit zu berechnen, wenn beispielsweise Wandkollisionen zu berücksichtigen sind.

**VU-F8:** *Erfüllt:* Es gibt einen Trainingsmodus und einen Evaluationsmodus.

**VU-NF1:** *Erfüllt:* Das BRIO Labyrinthspiel mit zwei BRIO Übungsplatten (sowie eigens designten Spielplatten) wurde mit Hilfe von VPython in 3D simuliert.

**VU-NF2:** *Erfüllt:* Die virtuelle Umgebung bildet die realen BRIO Labyrinth Spielfeldmaße proportional ab. Die realen Abmessungen wurden messtechnisch bestimmt und sind in Anforderung VU-NF2 niedergeschrieben und für die virtuellen 3D-Objektlängen entsprechend gesetzt, wobei die Längen der einzelnen 3D-Objekte in der Einheit cm interpretiert werden.

**VU-NF3:** *Erfüllt:* Auch die Kugel wurde proportional zu den realen BRIO Labyrinth Spielfeldmaßen modelliert.

**VU-NF4:** *Erfüllt:* Die maximale Spielfeldneigung, die durch das reale BRIO Labyrinth vorgegeben ist, wird durch den begrenzten diskreten Aktionsraum des Agenten gewährleistet. Hierbei kann das Spielfeld um maximal  $\pm 1,5^\circ$  geneigt werden.

**VU-NF5:** *Erfüllt:* Die nächste auszuführende Aktion wird durch den Agenten nach der Epsilon-Greedy Strategie bzw. der Epsilon-Reduzierungs-Strategie gewählt.

**VU-NF6:** *Erfüllt:* Im Konzept sind verschiedene mögliche Darstellungen des Zustands- und Aktionsraumes beschrieben (siehe Kapitel 5.1.4). Schlussendlich dienen als Aktionsraum diskrete Winkelstellungen. Der Zustandsraum wurde durch die aktuelle Position, die vorherige Position (oder Geschwindigkeit) und die Spielfeldorientierung beschrieben.

**VU-NF7:** *Erfüllt:* Die Implementierung der virtuellen Umgebung erfolgte in Python mit Hilfe der Python-Bibliotheken VPython und OpenAI Gym. Dadurch wurde eine nahtlose Integration mit gängigen RL-Frameworks ermöglicht.

**VU-NF8:** *Erfüllt:* Die Programmimplementierung erfolgte strukturiert und gut kommentiert. Dafür wurde mit verschiedensten Klassen, aussagekräftigen Namen bei Variablen bzw. Methoden, strukturierten Dictionaries sowie Programmkommentaren gearbeitet. Außerdem befinden sich in dieser Arbeit weitere Erläuterungen und UML-Diagramme (UML = Unified Modeling Language) zu den erstellten Programmen, die die Wartbarkeit erleichtern.

### 8.2.2 Physischer Demonstrator

Nachdem die Anforderungen für die virtuelle Umgebung überprüft wurden, wird für die Anforderungen an den physischen Demonstrator aus Kapitel 4.4.2 überprüft, in wie weit diese *erfüllt* oder *teilweise erfüllt* wurden.

**PD-F1:** *Erfüllt:* Es gibt eine Steuerungssoftware, die es ermöglicht die beiden Servomotoren gezielt anzusteuern. Dafür dienen die Pythondateien „ServoCommunication.py“ und „ArduinoCOM.py“, sowie das Arduinoprogramm „LabyrinthMachine.ino“.

**PD-F2:** *Erfüllt:* Das Stoppen der Motoren ist jeder Zeit möglich. Zum einen kann die Stromversorgung jederzeit durch Steckerziehen oder durch Drücken des Schalters des Tiefsetzstellers unterbrochen werden. Zum anderen kann jederzeit das Pythonprogramm beendet werden, wodurch keine neuen Winkelstellungen mehr übertragen werden.

**PD-F3:** *Erfüllt:* Vor jedem Spielstart wird die Spielplatte wieder zurück in die Ausgangslage gebracht, was bei den Servos der Mittelstellung von 0° entspricht.

**PD-F4:** *Erfüllt:* Ein manuelles Starten des Spiels wird durch die Bestätigung einer Messagebox ermöglicht (siehe Kapitel 7.2.1, Abbildung 7.9). Dadurch hat der Anwender die Möglichkeit die Kugel vor dem Beginn eines Spiels auf die Startposition zu legen.

**PD-F5:** *Erfüllt:* Für die Ermittlung des aktuellen Zustandes wird die W06 1080P HD Webcam Pro eingesetzt sowie OpenCV für die Bildverarbeitung. Eine weitergehende Optimierung der Bildverarbeitung ist Gegenstand einer zukünftigen, aufbauenden Arbeit. Dies wurde von Beginn an mit Herrn Prof. Dr. Hensel, dem Betreuer dieser Arbeit, in Abstimmungen festgelegt, um einen klaren inhaltlichen Fokus der Arbeit auf den KI-Bereich zu gewährleisten.

Tabelle 8.19: Gesamtausgaben

Produkt	Kosten
BRIO Labyrinth	56,78 €
Zahnriemenscheiben	14,99 €
Bedruckte Hartschaumplatten	5,90 €

**PD-F6:** *Erfüllt:* Die Automatisierung des realen Labyrinthspiels funktioniert sehr gut, dazu kann ein Video auf der CD eingesehen werden. Außerdem sind zwei Videoausschnitte zur Evaluierung eines in der Simulation vortrainierten Agenten am Demonstrator auf der CD hinterlegt. Vortrainierte Agenten können am Demonstrator weitertrainiert werden, was anhand von ca. 20 aufeinanderfolgenden Episoden getestet wurde. Da die Bildverarbeitung für die Ermittlung des nächsten Zustandes an den Ecken und vereinzelt an Wänden nicht stabil funktioniert, ist aktuell noch kein sinnvolles Weitertrainieren am Demonstrator möglich.

**PD-F7:** *Erfüllt:* Wie für die virtuelle Umgebung, gibt es für den physischen Demonstrator einen Trainings- und Evaluationsmodus.

**PD-NF1:** *Erfüllt:* Die Gesamtausgaben beliefen sich auf 77,67 €, da viele Elektronik- und Bauteilkomponenten bereits vorrätig waren. Die Ausgaben sind in der Tabelle 8.19 zusammengefasst. Die Obergrenze der Projektkosten von 250 € wurde somit eingehalten.

**PD-NF2:** *Erfüllt:* Die Spielfeldgröße beträgt 27,4 cm x 22,8 cm. Dies entspricht den Abmessungen der BRIO Labyrinthplatten sowie den zwei selbst designten und gedruckten Spielplatten.

**PD-NF3:** *Erfüllt:* Es kann eine grüne Kugel des GraviTrax-Spiels verwendet werden um den Kontrast zum Hintergrund zu erhöhen. Diese Stahlkugel entspricht von den Abmessungen und dem Gewicht der Kugel des eingesetzten BRIO Spiels.

**PD-NF4:** *Erfüllt:* Es wird eine Zahnriemenscheibe mit 20 Zähnen eingesetzt, wodurch die begrenzende Spielplattenauslenkung durch die Servos gegeben ist und nicht durch die mechanischen Gegebenheiten des realen Spiels. In x-Richtung ist eine maximale Auslenkung von  $\pm 3,17^\circ$  und in y-Richtung von  $\pm 2,43^\circ$  möglich. Diese maximal möglichen Winkel auslenkungen bzw. deren entsprechenden maximalen Pulsbreiten für die Servoansteuerung werden vor dem Setzen des PWM-Signals überprüft und gegebenenfalls

begrenzt. Des Weiteren ist der Aktionsraum des Agenten durch diskrete Winkelstellungen unterhalb der maximalen Servogrenzen definiert.

**PD-NF5:** *Erfüllt:* Auf den invasiven Eingriff bei der Motoranbringung für die Automatisierung des Spiels konnte durch das Klemmprinzip und die Verwendung von Riemen komplett verzichtet werden.

**PD-NF6:** *Erfüllt:* Die nächste auszuführende Aktion wird durch den Agenten nach der Epsilon-Greedy Strategie bzw. der Epsilon-Reduzierungs-Strategie gewählt.

**PD-NF7:** *Erfüllt:* Im Konzept sind verschiedene mögliche Darstellungen des Zustands- und Aktionsraumes beschrieben (siehe Kapitel 5.1.4). Schlussendlich dienen als Aktionsraum diskrete Winkelstellungen und der Zustandsraum wird durch die aktuelle Position, die vorherige Position und die Spielfeldorientierung beschrieben. In Kapitel 5.1.5 wurden verschiedenste Belohnungsstrategien beschrieben. Im Rahmen dieser Arbeit wurden die Belohnungsstrategie der Kacheln und Zwischenziele sowie die Schwellenbelohnung implementiert und getestet.

**PD-NF8:** *Erfüllt:* Bei dem Design der realisierten Labyrinthspielplatte „HOLES\_2“ wurde darauf geachtet, dass die Löcherpositionen so platziert sind, dass diese exakt mit Löchern der fest eingebauten Grundlabyrinthplatte übereinstimmen. Diese Übereinstimmung ist auch in Abbildung 7.5 in Kapitel 7.1.3 erkennbar.

**PD-NF9:** *Erfüllt / Teils Erfüllt:* Das automatisierte BRIO Spiel ist sehr kompakt transportierbar. Durch das Klemmprinzip kann es sehr einfach in seine Einzelteile zerlegt (und wieder zusammengesetzt) werden, die anschließend kompakt zusammengepackt werden können. Das Kameragestell ist hingegen nicht ganz so schnell zerlegbar und aufbaubar, es ist aber auch in einem Stück noch recht gut transportierbar.

**PD-NF10:** *Erfüllt:* Als Programmiersprache für das Reinforcement Learning wird Python eingesetzt.

**PD-NF11:** *Erfüllt:* Die Programmimplementierung erfolgte strukturiert und gut kommentiert. Dafür wurde mit verschiedene Klassen, aussagekräftigen Namen bei Variablen bzw. Methoden, strukturierten Dictionaries sowie einigen Programmkommentaren gearbeitet. Außerdem befinden sich in dieser Arbeit weitere Erläuterungen und UML-Diagramme zu den erstellten Programmen. Zur Erhöhung der Code-Qualität hat Herr Prof. Dr. Hensel, der Betreuer dieser Arbeit umfangreiche Code-Reviews durchgeführt, aufgrund deren viel Code-Refactoring betrieben wurde.

## 9 Fazit und Ausblick

Im Rahmen dieser Arbeit wurde ein Reinforcement-Learning-System entwickelt, welches in der Lage ist eine Kugel autonom durch verschiedene komplexe Labyrinthe zu navigieren. Hierzu wurde eine detaillierte 3D-Simulation des BRIO-Labyrinths erstellt, die diverse physikalische Eigenschaften der Kugelbewegung berücksichtigt. In der virtuellen Umgebung stehen sechs Spielplatten zur Verfügung, darunter sowohl original BRIO-Labyrinthplatten als auch eigens designte Spielplatten. Diese Platten wurden mithilfe eines Deep-Q-Learning-Algorithmus trainiert und konnten erfolgreich beherrscht werden. Zur Bestimmung der optimalen RL-Parameter wurden umfangreiche Tests durchgeführt und die Einstellungen miteinander verglichen. Dabei zeigte sich, wie entscheidend eine sorgfältige Parameterauswahl für den Lernerfolg ist. Zu den wesentlichen Parametern gehören unter anderem die Anzahl der Neuronen und Schichten im Netz, die Wahl der Aktivierungs- und Verlustfunktionen, die Werte für die Epsilon-Greedy-Strategie bzw. Epsilon-Reduzierungs-Strategie sowie die Belohnungssystematik. Parallel zur Simulation wurde das physische BRIO-Labyrinth automatisiert, wobei die Position der Kugel mithilfe von Farbfilterungen in der Bildverarbeitung ermittelt wird. Für die Automatisierung kommen Servomotoren zum Einsatz, die über ein Klemmprinzip am Spiel befestigt werden und unter Verwendung von Riemen die Rotationsbewegung an die Drehknöpfe übertragen. Ein invasiver Eingriff in das Spiel war dabei nicht notwendig. In der physischen Umgebung kann ein RL-Algorithmus trainiert oder vortrainierte Agenten aus der Simulation evaluiert werden. Eine benutzerfreundliche Oberfläche erlaubt zudem die Parametrierung des Trainings oder der Evaluierung, indem beispielsweise die Spielplatte, die Umgebung (physisch oder virtuell) sowie Parameter des neuronalen Netzes und des Agenten individuell angepasst werden können.

Die Kombination aus der 3D-Simulation und dem physischem Demonstrator ermöglichte nicht nur ein tieferes theoretisches Verständnis der KI-Thematik, sondern brachte auch wertvolle Erfahrungen in der praktischen Anwendung und Implementierung. Dabei wurde interdisziplinär gearbeitet, indem Informatik, Mathematik bzw. Physik, Neurowissenschaften, Elektronik und Mechanik vereint wurden. Alle erarbeiteten Ergebnisse

## *9 Fazit und Ausblick*

---

sowie der Quellcode werden in einem öffentlich zugängliches GitHub-Repository<sup>1</sup> zur Verfügung gestellt, um künftige Arbeiten und Weiterentwicklungen zu unterstützen.

Für vorgesehene weiterführende Arbeiten sind mehrere Erweiterungen und Optimierungen des bestehenden Systems angedacht, um die Funktionalität und Effizienz weiter zu verbessern. Ein zentraler Punkt ist die Integration einer hochwertigen Industriekamera um eine präzisere Erfassung des Zustands im physischen Spiel zu ermöglichen. Hierfür soll die derzeit eingesetzte Kamera zeitnah durch eine von der Firma Allied Vision Technologies GmbH zur Verfügung gestellte Kamera der Modellreihe Alvium 1800<sup>2</sup> ersetzt werden. Diese Kamera wird eine höhere Bildqualität sowie eine schnellere und genauere Erkennung der Kugelposition ermöglichen, was die Leistung der Bildverarbeitung und des Reinforcement-Learning-Systems optimieren dürfte. Ein weiterer zentraler Punkt ist die Weiterentwicklung der Bildverarbeitung um ein sinnvolles Weitertrainieren und Spielen am physischen Demonstrator zu ermöglichen. Durch die Implementierung erweiterter Algorithmen der Bildanalyse könnten etwa die Erkennung der Kugel beispielsweise in den Ecken des Spielfeldes verbessert werden. Darüber hinaus könnten weitere Reinforcement-Learning-Algorithmen getestet und mit dem bisher verwendeten DQN-Algorithmus verglichen werden. Durch andere RL-Algorithmen kann möglicherweise die Lernkurve beschleunigen oder die Stabilität des Agenten weiter optimiert werden. Ein weiteres spannendes Themengebiet stellt die Generalisierung des Systems auf gänzlich unbekannte Labyrinth-Layouts dar. Dies könnte über erweiterte Bildverarbeitungstechniken realisiert werden, die in der Lage sind Hindernisse und Wege im Labyrinth automatisch zu erkennen und zu interpretieren. Alternativ könnte ein flexibleres Belohnungssystem entwickelt werden, dass beispielsweise das Erlernen beliebiger Pfade innerhalb eines Labyrinths ermöglicht. Zusätzliche technische Erweiterungen können ebenfalls in Betracht gezogen werden. So ist die Integration einer Ballrückführungsmaschine denkbar, die den physischen Demonstrator vollständig automatisiert und für durchgängig kontinuierliche Trainingsszenarien nutzbar macht.

---

<sup>1</sup>[https://github.com/MarcOnTheMoon/ai\\_labyrinth](https://github.com/MarcOnTheMoon/ai_labyrinth)

<sup>2</sup><https://www.alliedvision.com/de/produktportfolio/alvium-kameraserien/> - Zugriffsdatum: 16.09.2024

## 10 Danksagung

Ganz herzlich möchte ich mich an dieser Stelle bei all denen bedanken, die mich im Hinblick auf meine Masterarbeit unterstützt haben.

Mein ganz besonderer Dank gilt meinem betreuenden Professor, Herrn Dr. Hensel. Zum einen dass er mir dieses interessante, vielseitige und zukunftsweisende Thema ermöglicht hat. Zum anderen für die sehr gute Betreuung und Unterstützung während der Anfertigung meiner Masterarbeit. Besonders schätze ich seine wertvollen Beiträge beispielsweise im Bereich der 3D-Drucke, sowie die wertvollen Impulse hinsichtlich einer eleganten Programmstrukturierung und der Auswahl der effizientesten Datentypen. Von ihm konnte ich unglaublich viel lernen, was mich auch über die Zeit an der Hochschule hinaus begleiten und weiterbringen wird.

Ein herzlicher Dank geht auch an Frau Prof. Dr. Herster, die recht kurzfristig die Rolle der Zweitprüferin übernimmt.

Weiterhin möchte ich meiner Familie und meinen Freunden danken, die mir während der gesamten Zeit Rückhalt gegeben und mich motiviert haben.



# Literaturverzeichnis

- [1] ALBUS, JAMES S.: *A New Approach to Manipulator Control: The Cerebellar Model Articulation Controller (CMAC)*, Research Paper in Journal of Dynamic Systems, Measurement and Control, 1975
- [2] AMHERD, FABIAN; RODRIGUEZ, ELIAS: *Heatmap-based Object Detection and Tracking with a Fully Convolutional Neural Network*, Kantonsschule Stadelhofen, Matura Paper, 2021
- [3] AZ-DELIVERY: *MG996R Micro Digital Servo Motor mit Metall Getriebe für RC Roboter Hubschrauber Flugzeug*. – URL <https://www.az-delivery.de/products/az-delivery-servo-mg996r>. – Zugriffsdatum: 19.04.2024
- [4] AZ-DELIVERY: *Mikrocontroller Board ATMega328 Datenblatt*. – URL [https://cdn.shopify.com/s/files/1/1509/1638/files/Mikrocontroller\\_Board\\_ATMega328P\\_Datenblatt\\_AZ-Delivery\\_Vertriebs\\_GmbH.pdf?](https://cdn.shopify.com/s/files/1/1509/1638/files/Mikrocontroller_Board_ATMega328P_Datenblatt_AZ-Delivery_Vertriebs_GmbH.pdf?). – Zugriffsdatum: 14.09.2024
- [5] AZ-DELIVERY: *Mikrocontroller Board AZ-ATmega328-Board mit USB-Kabel*. – URL <https://www.az-delivery.de/products/mikrocontroller-board>. – Zugriffsdatum: 19.04.2024
- [6] AZ-DELIVERY: *PCA9685 16 Kanal 12 Bit PWM Servotreiber für Raspberry Pi*. – URL <https://www.az-delivery.de/products/pca9685-servotreiber>. – Zugriffsdatum: 19.04.2024
- [7] AZ-DELIVERY: *PCA9685\_Servotreiber\_Datenblatt*. – URL [https://cdn.shopify.com/s/files/1/1509/1638/files/PCA9685\\_Servotreiber\\_Datenblatt.pdf?](https://cdn.shopify.com/s/files/1/1509/1638/files/PCA9685_Servotreiber_Datenblatt.pdf?). – Zugriffsdatum: 14.09.2024
- [8] AZ-DELIVERY: *Servo MG996R Datenblatt*. – URL [https://cdn.shopify.com/s/files/1/1509/1638/files/Servo\\_MG996R\\_Datenblatt.pdf?](https://cdn.shopify.com/s/files/1/1509/1638/files/Servo_MG996R_Datenblatt.pdf?). – Zugriffsdatum: 14.09.2024

## *Literaturverzeichnis*

---

- [9] BADIA, ADRIÀ PUIGDOMÈNECH; PIOT, BILAL; KAPTUROWSKI, STEVEN; SPRECHMANN, PABLO; VITVITSKYI, ALEX; GUO, DANIEL; BLUNDELL, CHARLES: *Agent57: Outperforming the Atari Human Benchmark*, Paper, 30.03.2020
- [10] BAUM, Hermann: *Dezentraler elastischer Stoß*. 30.03.2023. – URL [https://hermann-baum.de/elastischer\\_stoss/](https://hermann-baum.de/elastischer_stoss/). – Zugriffsdatum: 24.04.2024
- [11] BI, Thomas: History. (20.12.2023). – URL <https://www.cyberrunner.ai/history/>. – Zugriffsdatum: 28.03.2024
- [12] BI, THOMAS; D'ANDREA, RAFFAELLO: *Sample-Efficient Learning to Solve a Real-World Labyrinth Game Using Data-Augmented Model-Based Reinforcement Learning*, ETH Zürich, Paper, 2023
- [13] BRIO ONLINE SHOP: *Labyrinth mit Übungsplatten, rot.* – URL <https://www.brio.de/de-DE/produkte/spiele/labyrinth-mit-uebungsplatte-n-rot-63402000>. – Zugriffsdatum: 14.09.2024
- [14] BRIO ONLINE SHOP: BRIO Labyrinth. (2023). – URL <https://www.brio-shop.de/brio-labyrinth/>. – Zugriffsdatum: 28.03.2024
- [15] BÜNTE, Oliver: Rekord: KI-Roboter knackt Labyrinth-Geschicklichkeitsspiel. In: *heise online* (22.12.2023). – URL <https://www.heise.de/news/Rekord-KI-Roboter-knackt-Labyrinth-Geschicklichkeitsspiel-9581075.html>. – Zugriffsdatum: 26.03.2024
- [16] CLEVERT, DJORK-ARNÉ; UNTERTHINER, THOMAS; HOCHREITER, SEPP: *Fast and Accurate Deep Network Learning by Exponential Linear Units (ELUs)*, Johannes Kepler University, Paper, 05.05.2015
- [17] DEUTSCHES FORSCHUNGSZENTRUM FÜR KÜNSTLICHE INTELLIGENZ GMBH: BRIO Labyrinth - Testbed für die Entwicklung von Lernarchitekturen. (07.11.2023). – URL <https://robotik.dfki-bremen.de/de/forschung/robotersysteme/brio-labyrinth>. – Zugriffsdatum: 26.03.2024
- [18] DOBRINSKI, PAUL; KRAKAU, GUNTER; VOGEL, ANSELM: *Physik für Ingenieure*. 12., aktualisierte Aufl. Vieweg + Teubner, 2010. – ISBN 9783834805805
- [19] DOLD MECHATRONIK GMBH: *Zahnriemen - GT2 6mm breit.* – URL <https://www.dold-mechatronik.de/Zahnriemen-GT2-6mm-breit>. – Zugriffsdatum: 25.07.2024

- [20] DUCHI, JOHN; HAZAN, ELAD; SINGER, YORAM: *Adaptive Subgradient Methods for Online Learning and Stochastic Optimization*, paper, 07.2011
- [21] EICH-SOELLNER, Edda: *Kleine mathematische Formelsammlung*. Haufe, 2006. – ISBN 3448071978
- [22] ELGENDY, Mohamed: *Deep Learning for Vision Systems*. Manning Publications Co. LLC, 2020. – ISBN 9781638350415
- [23] ERIKSSON, OLLE; MALMBERG, AXEL: *Labyrinth navigation using reinforcement learning with a high fidelity simulation environment*, Linköping University, Masterarbeit, 2022
- [24] FRID, EMIL; NILSSON, FREDERIK: *Path Following Using Gain Scheduled LQR Control with applications to a labyrinth game*, Linköping University, Masterarbeit, 2020
- [25] GEHRKE, Winfried ; WINZKER, Marco ; WOITOWITZ, Roland ; URBANSKI, Klaus: *Digitaltechnik: Grundlagen, VHDL, FPGAs, Mikrocontroller*. Springer Vieweg, 2017. – ISBN 3662497301
- [26] GÉRON, AURÉLIEN; ROTHER, KRISTIAN: *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme*. 1. Auflage. dpunkt.verlag GmbH, 2018. – ISBN 978-3-96010-114-7
- [27] GLOROT, XAVIER; BENGIO, YOSHUA: *Understanding the difficulty of training deep feedforward neural networks*, Universite de Montreal, Paper, 2010
- [28] GRIDIN, Ivan: *Practical Deep Reinforcement Learning with Python: Concise Implementation of Algorithmus, Simplified Maths, and Effective Use of TensorFlow an PyTorch*. BPB Online, 2022. – ISBN 9789355512062
- [29] HAFNER, DANIJAR; PASUKONIS, JURGIS; BA, JIMMY; LILLICRAP, TIMOTHY: *Mastering Diverse Domains through World Models*, Paper, 10.01.2023
- [30] HAUN, Matthias: *Simulation Neuronaler Netze: Eine praxisorientierte Einführung*. Renningen-Malmsheim : expert-Verlag, 1998. – ISBN 3816915442
- [31] HE, KAIMING; ZHANG, XIANGYU; REN, SHAOQING; SUN, JIAN: *Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification*, Paper, 06.02.2015

- [32] HERING, EKBERT; MARTIN, ROLF; STOHRER, MARTIN: *Physik für Ingenieure*. 9. Auflage. Springer-Verlag, 2004. – ISBN 9783540351481
- [33] HERING, EKBERT; MARTIN, ROLF; STOHRER, MARTIN: *Physik für Ingenieure*. 12. Auflage. Springer Vieweg, 2016. – ISBN 9783662493540
- [34] HINTON, Geoffrey: *Neuronal Networks for Machine Learning: Lecture 6a Overview of mini-batch gradient descent*, University of Toronto, lecture
- [35] HYDE, Randall: *The Book of  $I^2C$ : A Guide for Adventurers*. No Starch Press, 2022. – ISBN 9781718502475
- [36] IOFFE, SERGEY; SZEGEDY, CHRISTIAN: *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, Paper, 11.02.2015
- [37] JELLY COMB: *W06 1080P HD Webcam Pro*. – URL <https://www.jellycomb.com/products/w06-webcam>. – Zugriffssdatum: 13.09.2024
- [38] JELLY COMB: Web Camera user manual. . – URL [https://cdn.shopify.com/s/files/1/0053/8263/5610/files/WGBG-006\\_H606\\_W06.pdf?](https://cdn.shopify.com/s/files/1/0053/8263/5610/files/WGBG-006_H606_W06.pdf?) – Zugriffssdatum: 14.09.2024
- [39] JOY-IT: *Bread Power Spannungsversorgung für Breadboards*. – URL [https://cdn-reichelt.de/documents/datenblatt/A300/DATENBLATT\\_SBC-POW-BB.pdf](https://cdn-reichelt.de/documents/datenblatt/A300/DATENBLATT_SBC-POW-BB.pdf). – Zugriffssdatum: 14.09.2024
- [40] JOY-IT: *Servomotor: Technische Spezifikationen*. – URL <https://www.polin.de/media/47/33/52/1701729876/D820581-D.pdf>. – Zugriffssdatum: 14.09.2024
- [41] KINGMA, DIEDERIK P.; BA, JIMMY: *Adam: A Method for Stochastic Optimization*, conference paper at International Conference on Learning Representations, 22.12.2014
- [42] KLEMENT, Joachim: *Werkzeugmaschinen-Nebenbaugruppen: Automation und Energieeffizienz*. Tübingen : expert verlag, 2019. – ISBN 9783816984573
- [43] KLESSASCHECK, Mario: Funktionale Anforderungen versus nicht-funktionale Anforderungen. In: *Johner Institut GmbH* (18.04.2023). – URL <https://www.johner-institut.de/blog/iec-62304-medizinische-software/funktionale-und-nicht-funktionale-anforderungen/>. – Zugriffssdatum: 02.04.2024

- [44] KLOSTERMEIER, ROBIN; HAAG, STEFFI; BENLIAN, ALEXANDER: *Geschäftsmodelle digitaler Zwillinge: HMD Best Paper Award 2018*. Springer Vieweg, 2020. – ISBN 9783658283537
- [45] KONRADIN-VERLAG ROBERT KOHLHAMMER GMBH: Was sind Schrittmotoren, welche Typen gibt es und wie funktionieren sie? (01.07.2021). – URL <https://kem.industrie.de/elektromotoren/was-sind-schrittmotoren-welche-typen-gibt-es-und-wie-funktionieren-sie/>. – Zugriffsdatum: 30.03.2024
- [46] KONRADIN-VERLAG ROBERT KOHLHAMMER GMBH: Was ist ein Servomotor und wie funktioniert er? (06.05.2021). – URL <https://kem.industrie.de/elektromotoren/was-sind-servomotoren-und-wie-funktionieren-sie/>. – Zugriffsdatum: 30.03.2024
- [47] KUMAR, NEERAJ; TEKCHANDANI, RAJKUMAR: *Applied Deep Learning: Design and implement your own Neural Networks to solve real-world problems*. 1st ed. BPB Publications, 2023. – ISBN 9789355513700
- [48] LI, Shengbo E.: *Reinforcement Learning for Sequential Decision and Optimal Control*. 1st edition. Springer Nature Singapore, 2023. – ISBN 9789811977848
- [49] MAURER, WERNER: *Kugel auf schiefer Ebene*. 30.10.2012. – URL <https://www.youtube.com/watch?app=desktop&v=OT-73z3zC30>. – Zugriffsdatum: 04.04.2024
- [50] METZEN, JAN HENDRIK; KIRCHNER, ELSA ; ABDENEBAOUI, LARBI ; KIRCHNER, FRANK: *The Brio Labyrinth Game - A Testbed for Reinforcement Learning and for Studies on Sensorimotor Learning*, Deutsches Forschungszentrum für Künstliche Intelligenz GmbH, Conference Paper, 2009
- [51] MNIIH, VOLODYMYR; KAVUKCUOGLU, KORAY; SILVER, DAVID; GRAVES, ALEX; ANTONOGLOU, IOANNIS; WIERSTRA, DAAN; RIEDMILLER, MARTIN: *Playing Atari with Deep Reinforcement Learning*, paper, 19.12.2013
- [52] NIEDRIG, HEINZ; STERNBERG, MARTIN: *Das Ingenieurwissen: Physik*. Springer Vieweg, 2014. – ISBN 9783642411274
- [53] NIELSEN, MICHAEL A.: *Neural Networks and Deep Learning*. Determination Press, 2015

- [54] OSINGA, Douwe: *Deep Learning Kochbuch: Praxisrezepte für einen schnellen Einstieg*. 1. Auflage. dpunkt.verlag GmbH, 2019. – ISBN 9783960102649
- [55] OTTE, Sebastian: *Künstliche neuronale Netze, Das Perzeptron*, Hochschule Rhein-Main, Paper, 12.2009
- [56] PALANISAMY, Praveen: *Hands-on intelligent agents with OpenAI Gym: Your guide to developing AI agents using deep reinforcement learning*. Packt Publishing, 2018. – ISBN 1788835131
- [57] PERROTTA, PAOLO: *Machine Learning für Softwareentwickler: Von der Python-Codezeile zur Deep-Learning-Anwendung*. 1.Auflage. dpunkt.verlag, 2020. – ISBN 9781680506600
- [58] POLLIN ELECTRONIC GMBH: *JOY-IT Servomotor, digital, bis 20 kg/cm, 40x20x40,5 mm.* – URL <https://www.pollin.de/p/joy-it-servomotor-digital-bis-20-kg-cm-40x20x40-5-mm-820581>. – Zugriffsdatum: 19.04.2024
- [59] POPOV, Valentin L.: *Kontaktmechanik und Reibung: Ein Lehr- und Anwendungs-buch von der Nanotribologie bis zur numerischen Simulation*. 1. Aufl. Springer-Verlag Berlin Heidelberg, 2009. – ISBN 9783540888369
- [60] PYTORCH CONTRIBUTORS: *Reproducibility*. 2023. – URL <https://pytorch.org/docs/stable/notes/randomness.html>. – Zugriffsdatum: 20.07.2024
- [61] RASCHKA, SEBASTIAN; MIRJALILI, VAHID: *Machine Learning mit Python und Keras, Tensorflow 2 und Scikit-learn: Das umfassende Praxis-Handbuch für Data Science, Deep Learning und Predictive Analytics*. 3. Auflage. mitp, 2021. – ISBN 9783747502136
- [62] RASCHKA, SEBASTIAN; PATTERSON, JOSHUA; NOLET, COREY: *Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence*, Paper, 2020
- [63] RAVENSBURGER AG: Kugeln von GraviTrax. (24.09.2020). – URL [https://service.ravensburger.de/Marken\\_und\\_Produkte/Ravensburger\\_Produkte/GraviTrax%C2%AE/Kugeln\\_von\\_GraviTrax%C2%AE](https://service.ravensburger.de/Marken_und_Produkte/Ravensburger_Produkte/GraviTrax%C2%AE/Kugeln_von_GraviTrax%C2%AE). – Zugriffsdatum: 02.04.2024

- [64] RAVICHANDIRAN, Sudharsan: *Deep Reinforcement Learning with Python: Master Classic RL, Deep RL, Distributional RL, Inverse RL, and More with OpenAI Gym and TensorFlow*. 2nd ed. Packt Publishing, 2020. – ISBN 1839215593
- [65] REICHELT ELEKTRONIK GMBH: *DEBO BREAD POWER*. – URL <https://www.reichelt.de/entwicklerboards-spannungsversorgung-f-steckboards-debo-bread-power-p202832.html>. – Zugriffsdatum: 19.04.2024
- [66] REITH, DIRK; SCHENK, MARTIN; STEINEBACH, GERD: *Modellbildung und Simulation: Eine anwendungsorientierte Einführung mit praktischen Beispielen in MATLAB und Julia*. 1st ed. 2024. Springer Fachmedien Wiesbaden, 2024. – ISBN 9783658442507
- [67] ROBBINS, HERBERT; MONRO SUTTON: *A stochastic approximation method*, University of North Carolina, paper, 1951
- [68] ROSENBLATT, FRANK: *The perceptron: a probabilistic model for information storage and organization in the brain*
- [69] RUDER, Sebastian: *An overview of gradient descent optimization algorithms*, Insight Centre for Data Analytics, NUI Galway, Paper, 15.06.2017
- [70] RUMMERY, G. A.; NIRANJAN, M.: *On-Line Q-Learning Using Connectionist Systems*, Cambridge University Engineering, paper, 09.1994
- [71] SAMMUT, CLAUDE; WEBB, GEOFFREY I.: *Encyclopedia of Machine Learning*. Springer, 2011. – ISBN 9780387301648
- [72] SCHWARZ, Helmut: *Simulationsgestützte CAD/CAM-Kopplung für die 3D-Laserbearbeitung mit integrierter Sensorik*. Springer-Verlag Berlin Heidelberg GmbH, 1994. – ISBN 9783540575771
- [73] SCHWEIZER-FN: *Reibwerte von verschiedenen Materialien*. 03.12.2022. – URL [https://www.schweizer-fn.de/stoff/reibwerte/reibwerte\\_sonstige.php#sonstiges](https://www.schweizer-fn.de/stoff/reibwerte/reibwerte_sonstige.php#sonstiges). – Zugriffsdatum: 16.04.2024
- [74] SUTTON, RICHARD S.; BARTO, ANDREW G.: *Reinforcement learning: An introduction*. 2. The MIT Press, 2018. – ISBN 0262193981
- [75] WATKINS, Christopher John Cornish H.: *Learning from Delayed Rewards*, King's Collage, Ph.D Thesis, 05.1989

## *Literaturverzeichnis*

---

- [76] WIEDEMANN, Christine: *Neuronale Netze und Fuzzy-Logik in der Neuprodukt-Erfolgsfaktorenforschung*. Deutscher Universitäts-Verlag, 1999. – ISBN 9783322952097
- [77] XU, BING; WANG, NAIYAN; CHEN, TIANQI; LI, MU: *Empirical Evaluation of Rectified Activations in Convolutional Network*, University of Alberta, Hong Kong University of Science and Technology, University of Washington, Carnegie Mellon University, Paper, 05.05.2015
- [78] ZAI, ALEX; BROWN, BRANDON: *Einstieg in Deep Reinforcement Learning: KI-Agenten mit Python und PyTorch programmieren*. Carl Hanser Verlag, 2020. – ISBN 9783446466081
- [79] ZEBEROXYZ 3D STORE: *2 Stück Set GT2-Synchronrad, 20 & 60 Zähne, 6.35mm Bohrung, Aluminium-Zahnriemenscheibe mit 2 Stück Länge 200 mm Breite 6mm Riemen (20-60T-6.35B-6)*. – URL <https://www.amazon.de/Zeberoxyz-Synchronrad-Aluminium-Zahnriemenscheibe-20-80T-6B-6/dp/B08X68JKS7/?th=1>. – Zugriffsdatum: 25.07.2024
- [80] ZEILER, Matthew D.: *ADADELTA: An Adaptive Learning Rate Method*, paper, 22.12.2012

# A Anhang

Für die Nutzung der im Rahmen dieser Abschlussarbeit entstandenen Software ist die Installation einige Softwarebibliotheken erforderlich. Die Installationen können beispielsweise mit Miniconda3 über den Conda-Paketmanager erfolgen. Das Installationsvorgehen wird nachfolgend anhand von Miniconda3 und PyCharm 2023.3.4 mit Python 3.10 genauer erläutert. Zuerst muss eine Conda Umgebung im Anaconda Prompt erstellt und aktiviert werden:

```
conda create -name rl_course  
conda activate rl_course
```

Nach erfolgreich erstellter Umgebung können die verschiedenen Pakete im Anaconda Prompt installiert werden (siehe Tabelle A.1). Die Pakete gym, vpython, pytorch und matplotlib werden für die virtuelle Umgebung verwendet. Tk, opencv und pyserial werden beim physischen Demonstrator genutzt. Für die Benutzeroberfläche, aus der das Training oder die Evaluation gestartet werden kann wird PyQt6 benötigt.

Um den Anaconda Interpreter in PyCharm nutzen zu können, müssen die Path Umgebungsvariablen um drei Pfade erweitert werden:

```
C:\Users\<username>\miniconda3  
C:\Users\<username>\miniconda3\Scripts
```

Tabelle A.1: Packetinstallation

Anaconda Prompt	PyCharm Terminal
conda install -c conda-forge gym-all	pip install gymnasium
conda install -c conda-forge vpython	pip install vpython
conda install pytorch::pytorch	pip install torch
conda install conda-forge::matplotlib	pip install matplotlib
conda install anaconda::tk	pip install tk
conda install -c conda-forge::opencv	pip install opencv-python
conda install conda-forge::pyserial	pip install pyserial
	pip install PyQt6

## A Anhang

---

*C:\Users\<username>\miniconda3\Library\bin*

Anschließend kann in PyCharm conda als Interpreter gewählt werden und die pip Befehle im Pycharm Terminal ausgeführt werden (siehe Tabelle A.1). Ausführlichere Installationserläuterungen sind unter<sup>1</sup> und<sup>2</sup> zu finden.

Für das Arduinoprogramm zur Servoansteuerung beim physischen Demonstrator wird noch eine Adafruit Bibliothek benötigt. Dazu muss in der Arduino IDE wie folgt vorgegangen werden:

1. Klicke auf: Sketch → include Library → Manage Libraries.
2. Suche im Textfeld nach Adafruit PWM Servo Driver Library und installiere diese Bibliothek

---

<sup>1</sup><https://ajpatel-bigdata.medium.com/configure-anaconda-python-interpreter-or-pycharm-ide-and-installation-of-gym-package-b4b6ba5717a6> - Zugriffsdatum: 15.05.2024

<sup>2</sup><https://www.youtube.com/watch?v=e3DyCg0fgx0> - Zugriffsdatum: 15.05.2024

### **Erklärung zur selbständigen Bearbeitung**

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

---

Ort

Datum

Unterschrift im Original