FH Westküste
Wirtschaft und Technik

HAW HAMBURG

Masterarbeit

Pascal Gutthardt

Deep Reinforcement Learning zur autonomen Steuerung eines Flipperautomaten mit kamerabasierter Umgebungserfassung

Fachhochschule Westküste
Fachbereich Technik

FH Westküste - University of Applied Sciences
Faculty of Engineering

Hochschule für Angewandte Wissenschaften Hamburg
Fakultät Elektro-, Medien- und Informationstechnik
Hamburg University of Applied Sciences
Faculty of Electrical Engineering,
Media and Information Technology

Pascal Gutthardt

Deep Reinforcement Learning zur autonomen Steuerung eines Flipperautomaten mit kamerabasierter Umgebungserfassung

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang *Mikroelektronische Systeme*
der Fakultät Elektro-, Medien- und Informationstechnik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Ing. Marc Hensel
Zweitprüfer: Prof. Dr. Ing. Stephan Hußmann

Abgabedatum: 01.10.2025

Kurzreferat

Pascal Gutthardt

Thema der Arbeit

Deep Reinforcement Learning zur autonomen Steuerung eines Flipperautomaten mit kamerabasierter Umgebungserfassung

Stichworte

Künstliche Intelligenz, Reinforcement Learning, Deep Reinforcement Learning, Deep Q-Network, Bildverarbeitung, Objekterkennung, Virtuelle Umgebung / Digitaler Zwilling, Simulation und Training, Autonomer Flipperautomat, Echtzeitsteuerung

Kurzzusammenfassung

Diese Masterarbeit untersucht die Entwicklung eines autonomen Flipperautomaten, der durch den Einsatz von Deep Reinforcement Learning (DRL) und Bildverarbeitung selbstständig agieren kann. Ziel war es, ein System zu realisieren, das die Position der Kugel mithilfe einer Kamera erfasst und in Echtzeit optimale Steuerungsentscheidungen für die Flipperarme trifft. Hierfür wurde ein digitaler Zwilling als Trainingsumgebung aufgebaut, um den Agenten effizient und risikoarm zu trainieren, bevor das erlernte Verhalten auf einen physischen Demonstrator übertragen wurde. Im Rahmen der Arbeit wurde ein Deep Q-Network (DQN) implementiert, dessen Leistungsfähigkeit anhand verschiedener Experimente und Parameterstudien evaluiert wurde. Die Ergebnisse zeigen, dass der entwickelte Agent in der Lage ist, robuste Strategien zu erlernen, um die Kugel möglichst lange im Spiel zu halten. Damit leistet die Arbeit einen Beitrag zum praktischen Einsatz moderner KI-Methoden in interaktiven, realweltlichen Szenarien.

Pascal Gutthardt

Title of Thesis

Deep Reinforcement Learning for Autonomous Control of a Pinball Machine with Camera-Based Environment Perception

Keywords

Artificial Intelligence, Reinforcement Learning, Deep Reinforcement Learning, Deep Q-Network, Computer Vision, Object Detection, Virtual Environment / Digital Twin, Simulation and Training, Autonomous Pinball Machine, Real-Time Control

Abstract

This master's thesis explores the development of an autonomous pinball machine controlled by Deep Reinforcement Learning (DRL) and computer vision. The

objective was to create a system capable of detecting the ball's position using a camera and making real-time control decisions for the flipper arms. To achieve this, a digital twin was designed as a training environment, enabling efficient and safe training of the agent before transferring the learned behavior to a physical demonstrator. A Deep Q-Network (DQN) was implemented and its performance evaluated through a series of experiments and parameter studies. The results demonstrate that the developed agent is capable of learning robust strategies to keep the ball in play for extended periods. Thus, the work contributes to the practical application of modern AI methods in interactive, real-world scenarios.

Inhaltsverzeichnis

Abbildungsverzeichnis **IV**

Tabellenverzeichnis **VI**

Abkürzungen **VII**

1 Einleitung **1**

1.1 Motivation 1
1.2 Zielsetzung 2
1.3 Aufbau der Arbeit 2

2 Grundlagen **4**

2.1 Funktionsweise einer Pinball-Maschine 4
2.2 Bildverarbeitung 6
2.2.1 OpenCV 7
2.2.2 Digitale Bilder und Farbräume 8
2.2.3 Bildvorverarbeitung 8
2.3 Maschinelles Lernen 11
2.4 Künstliche neuronale Netzwerke 12
2.4.1 Deep Learning 13
2.4.2 Architekturen von künstlichen neuronalen Netzen 13
2.4.3 Aktivierungsfunktionen in neuronalen Netzwerken 14
2.4.4 Gradientenverfahren 16
2.4.5 Backpropagation (Fehlerrückführung) 19
2.4.6 Verlustfunktionen 20
2.4.7 Optimierer 21
2.4.8 Gewichtsinitialisierung 22
2.4.9 Batch-Normalisierung 24
2.5 Grundlagen des Reinforcement Learning 25
2.6 Q-Learning 29
2.7 Deep Q-Learning 30
2.8 Vergleich zwischen Q-Learning und Deep Q-Learning 30

3 Stand der Technik **32**

3.1 Physische Flipperautomaten 32
3.2 Digitale Flipperautomaten 32
3.3 Erste Schritte der KI gesteuerten Flipperautomaten 33
3.3.1 Visuelle Systeme und Automatisierung 33

3.3.2 PinBot – Reinforcement Learning mit einem digitalen Flipperautomaten . . . 33
3.3.3 Laufendes Forschungsprojekt: KI-gesteuerter Flipperautomat an der Helmut-Schmidt-Universität . . . 34
3.4 Vorhandener Fortschritt . . . 35

4 Anforderungsanalyse 38
4.1 Systemumgebung . . . 38
4.2 Stakeholder des Projekts . . . 38
4.3 Anforderungen . . . 41
4.3.1 Anforderungen an die Bildverarbeitung (BV) . . . 41
4.3.2 Anforderungen an die virtuelle Umgebung . . . 43
4.3.3 Anforderungen an die künstliche Intelligenz (KI) . . . 45
4.3.4 Anforderungen an den physischen Demonstrator (PD) . . . 47

5 Konzept 48
5.1 Systemarchitektur . . . 49
5.2 Kamera . . . 49
5.3 Virtuelle Umgebung . . . 51
5.3.1 Design . . . 51
5.3.2 Struktur . . . 52
5.4 Physische Umgebung . . . 53
5.5 Reinforcement Learning . . . 54
5.5.1 DQN-Agenten . . . 54
5.5.2 Zustands- und Aktionsraum . . . 56
5.5.3 Belohnungssystem . . . 57

6 Entwicklung 59
6.1 Nacharbeiten am physischen Demonstrator . . . 59
6.2 Entwicklung der physischen Umgebung . . . 60
6.2.1 Korrekturen von Verzerrungen . . . 60
6.2.2 Auswahl des Verfahrens für die Bildverarbeitung . . . 62
6.2.3 Parameter der Kamera . . . 66
6.2.4 Begründung der Belichtungszeit . . . 66
6.2.5 Extraktion des Spielfelds . . . 67
6.2.6 Perspektivische Transformation des Spielfeldes . . . 69
6.2.7 Erfassung der Winkel der Flipperarme für Kalibrierungsmaßnahmen 70
6.2.8 Korrektur des Startarms . . . 75
6.2.9 Detektion der Flipperkugel . . . 77
6.2.10 Zeitvorgaben der physischen Umgebung . . . 80

6.2.11 Sequenzdiagramm der Unterklasse `PinballCamera` 82
6.3 Entwicklung der virtuellen Umgebung 83
6.3.1 3D-Modell für den Rendermodus 84
6.3.2 Physikalische Eigenschaften 86
6.3.3 Rollverhalten der Kugel mit der Funktion `step` 89
6.3.4 Kollisionskarte für statische Objekte 93
6.3.5 Kollisionserkennung mit Spielfeldobjekten 94
6.3.6 Kollision mit den Flipperarmen 96
6.3.7 Allgemeine Kollisionsberechnung 99
6.4 Entwicklung des RL-Agenten 101
6.4.1 Wahl des RL-Frameworks 101
6.4.2 Wahl der API 103
6.4.3 Entwicklung des Agenten 103
6.4.4 Architektur und Funktionsweise des tiefen Q-Netzwerks (DQN) . 106
6.4.5 Hauptprogramm `TrainAgent` 108
6.4.6 Ermittelung der optimalen Belohnungen 109
6.4.7 Tests der RL-Parameterbestimmung 113
6.4.8 Optimalste Hyperparameter für die virtuelle Umgebung 120
6.4.9 Fortlaufendes Training in der physischen Umgebung 121
6.4.10 Probleme während des Trainings 121
6.4.11 Allgemeines Ergebnis des Trainings am physischen Demonstrator 122

7 Evaluierung des Systems 124
7.1 Anforderungsabgleich 124
7.1.1 Anforderungsabgleich der Bildverarbeitung (BV) 124
7.1.2 Anforderungsabgleich der virtuellen Umgebung (VU) 125
7.1.3 Anforderungsabgleich der künstlichen Intelligenz (KI) 126
7.1.4 Anforderungsabgleich des physischen Demonstrator (PD) 127
7.1.5 Zusammenfassung der Evaluierung 128

8 Fazit und Ausblick 130

9 Danksagung 131

Literaturverzeichnis 132

Abbildungsverzeichnis

2.1 Spielfeld eines Flipperautomaten 6
2.2 Darstellung eines RGB-Bildes (links), das in seine drei Farbkanäle rot, grün und blau zerlegt wird (rechts), jeweils mit einer schematischen 3x3-Matrix 7
2.3 Links das Originalbild und rechts nach dem Anwenden des Filters `GaussianBlur()` 9
2.4 Links das Originalbild und rechts nach dem Anwenden des Filters `Canny()` 9
2.5 Differenzbild: Flipper inaktiv (links), Flipper aktiv (mittig) und Differenzbild (rechts) 10
2.6 Darstellung eines neuronalen Netzes mit zwei verborgenen Schichten . . . 12
2.7 Schematische Darstellung eines Perzeptrons mit drei Eingängen, Bias b, Gewichten w_i und Aktivierungsfunktion φ. 14
2.8 Sigmoid- und Tanh-Aktivierungsfunktion 15
2.9 ReLU- und Leaky ReLU-Aktivierungsfunktion 16
2.10 RL-Framework des Reinforcement Learning [BZ20] 26
2.11 Standardmodell des Reinforcement Learning [BZ20] 28
2.12 Schematischer Vergleich zwischen Q-Learning und Deep Q-Learning . . 31
3.1 Bereits vorhandene Simlulation 37
4.1 Vereinfachtes Schema der Systemumgebung 39
4.2 Anwendungsfalldiagramm 42
5.1 Darstellung der Systemarchitektur 49
5.2 Mögliche Grundstruktur der virtuellen Umgebung 52
6.1 Physischer Aufbau mit der montierten Kamera 59
6.2 ChArUco Board 60
6.3 Analyse der ChArUco-Marker 61
6.4 Kalibrierung der Rohbilder: Rohbild (links) und entzerrtes Bild (rechts) . 62
6.5 Schematische Darstellung des LED-Flackerns als gleichgerichteter Sinusfunktion und eines Belichtungsfensters von 10 ms und 4,3 ms. 67
6.6 Extrahierung des Spielfelds 67
6.7 Interaktive Auswahl der Spielfeld-Ecken mithilfe des FieldCroppers . . . 68
6.8 Umwandlung des Bildes 70
6.9 Skizze des unteren Flipperspielfelds 70
6.10 Erkennung des Bereichs der Flipperarme 71
6.11 Differenzbild: Flipper inaktiv (links), Flipper aktiv (mittig) und Differenzbild (rechts) 71
6.12 Aufnahme des ermittelten Flipperbereichs 73
6.13 Ausgabe wie die Winkel der Flipperarme erkannt werden 75
6.14 Seitenansicht der beiden möglichen Szenarien des Startarms 75

6.15 Erkennung der grünen Markierung am Startarm 77
6.16 Vereinfachtes Ablaufdiagramm der Startarmkalibrierung 78
6.17 Detektion der Kugel mit markiertem Bereich der Flipperarme 80
6.18 Vereinfachter zeitlicher Ablauf einer Zustandsverarbeitung 81
6.19 Vereinfachtes Sequenzdiagramm der Klasse `PinballCamera` 83
6.20 Vereinfachtes Klassendiagramm der physischen Umgebung 84
6.21 Klassendiagramm der 3D-Modellierung 87
6.22 Laufende Simulation mit allen Objekten inkl. Achsenmarkierung 88
6.23 Konzept der Kartengestaltung für Hindernisse 94
6.24 Vereinfachte Darstellung der Kollisionüberprüfung mit statischen Objekten 95
6.25 Vereinfachte Darstellung einer Kollision mit eingetragenen Grundgrößen . 97
6.26 Vereinfachtes Ablaufdiagramm der virtuellen Umgebung 101
6.27 Vereinfachtes Klassendiagramm der virtuellen Umgebung 102
6.28 Vereinfachtes Modell des tiefen Q-Netzwerks 107
6.29 Vereinfachtes Klassendiagramm des Hauptprogramms `TrainAgent` . . . 109
6.30 Konzept der Staffelung, markierte Bereiche auf dem Spielfeld 112
6.31 Ablaufdiagramm des Belohnungssystem 113
6.32 Erster Testdurchlauf eines Trainings mit den Starthyperparametern 114
6.33 Erzielte Belohnungen über 1000 Episoden mit den besten Hyperparametern 121
6.34 Erzielte Belohnungen über 500 Episoden mit den besten Hyperparametern
am physischen Demonstrator 122

Tabellenverzeichnis

3.2 Vergleich der Ergebnisse zwischen menschlichem Spieler, trainiertem Agenten, untrainiertem Agenten und keiner Agentenunterstützung. 34
3.3 Bauteile des Flipper-Demonstrators 36
5.1 Vergleich der vorhandenen Kameras für den Flipper-Prototyp 50
6.1 Bewertungskriterien und Gewichtung 64
6.2 Bewertung der Methoden (1: schlecht bis 5: exzellent) 65
6.3 Nutzwertanalyse: klassische, KI-basierte und hybride Bildverarbeitung . . 66
6.4 Gemessene Zeiten der wichtigsten Sequenzen 81
6.5 Ergebnisse für die Trainingstestdurchläufe mit verschiedenen verbogenen Schichten im DQN 115
6.6 Ergebnisse für die Trainingstestdurchläufe mit drei Verlustfunktionen . . 116
6.7 Ergebnisse für die Trainingstestdurchläufe mit drei Aktivierungsfunktionen 117
6.8 Ergebnisse für die Trainingstestdurchläufe mit unterschiedlichen Werten für Epsilon_{min} und Epsilon_{decay} 118
6.9 Ergebnisse für die Trainingstestdurchläufe mit unterschiedlichen Werten für die Lernperiode und die Batchsize 119
6.10 Ergebnisse für die Trainingstestdurchläufe für unterschiedliche Gammawerte 119

Abkürzungen

Abkürzung	Bedeutung
AdaDelta	Adaptive Delta (Optimierungsverfahren)
API	Application Programming Interface (Programmierschnittstelle)
BN	Batch Normalization (Normalisierung in neuronalen Netzen)
BV	Bildverarbeitung
ChArUco	ChArUco Board (Charuco-Marker für Kamerakalibrierung)
CNN	Convolutional Neural Network (Faltungsneuronales Netz)
CSRT	Discriminative Correlation Filter with Channel and Spatial Reliability (Tracking-Algorithmus)
CUDA	Compute Unified Device Architecture (NVIDIA Parallel Computing Plattform)
DQL	Deep Q-Learning
DQN	Deep Q-Network
GPU	Graphics Processing Unit (Grafikprozessor)
GRU	Gated Recurrent Unit (Variante von LSTM-Netzen)
HSV	Hue-Saturation-Value Farbraum
JAX	Just After eXecution (Maschinelles Lernen Framework von Google)
JSON	JavaScript Object Notation (JSON-Datei/Format)
KI	Künstliche Intelligenz
KNN	k-Nearest Neighbor (Klassifikationsverfahren)
LSTM	Long Short-Term Memory (Spezialform rekurrenter Netze)
LUT	Look-Up Table (Nachschlagetabelle für Transformationen)
MAE	Mean Absolute Error (Mittlerer absoluter Fehler)
MDP	Markov Decision Process (Markow-Entscheidungsprozess)
ML	Machine Learning (Maschinelles Lernen)
MLP	Multilayer Perceptron (Mehrschichtiges Perzeptron)
MSE	Mean Squared Error (Mittlerer quadratischer Fehler)
OBB	Oriented Bounding Box (Orientierte Begrenzungsbox)

Abkürzung	**Bedeutung**
OpenCV	Open Source Computer Vision Library (OpenCV-Bibliothek)
PD	Proportional-Differential (Regleransatz in Steuerungssystemen)
ReLU	Rectified Linear Unit (Aktivierungsfunktion)
RGB	Rot-Grün-Blau Farbraum
RL	Reinforcement Learning (Bestärkendes Lernen)
RL-Agent	Agent im Reinforcement Learning
RL-Framework	Reinforcement Learning Framework
RL-System	Reinforcement Learning System
RMSProp	Root Mean Square Propagation (Optimierungsverfahren)
RNN	Recurrent Neural Network (Rekurrentes neuronales Netz)
TNA	Tracking-by-Normalized-Advantage (Tracking-Ansatz)
TPU	Tensor Processing Unit (Spezialhardware für ML)
UML	Unified Modeling Language (Modellierungssprache)
YAML	YAML Ain't Markup Language (Datenserialisierungsformat)
YOLO	You Only Look Once (Echtzeit-Objekterkennung)

1 Einleitung

Dieses Kapitel führt in die Thematik der Arbeit ein und stellt den Zusammenhang zwischen Künstlicher Intelligenz und dem gewählten Anwendungsbeispiel – einem KI-gesteuerten Flipperautomaten – her. Zunächst wird die Motivation für die Bearbeitung dieser Fragestellung erläutert, anschließend die konkreten Ziele der Arbeit beschrieben. Abschließend wird der Aufbau der Arbeit vorgestellt, um den inhaltlichen Rahmen für die folgenden Kapitel zu schaffen.

1.1 Motivation

Künstliche Intelligenz (KI) ist heutzutage eines der dynamischsten Forschungsfelder, welches in vielen Bereichen unseres täglichen Lebens zunehmend an Bedeutung gewinnt. Ihre Fähigkeit, aus Daten zu lernen und eigenständig komplexe Probleme zu lösen, macht sie unverzichtbar in Anwendungsgebieten wie der Robotik, bei autonomen Fahrzeugen, bei medizinischen Diagnosen und sogar bei der Optimierung von Energieverbrauch. Durch den rasanten Fortschritt in diesen Technologien steigt das Interesse der Öffentlichkeit, und der Drang die innovativen Lösungen zu entwickeln und umzusetzen.

Faszinierend ist die Fähigkeit der KI anspruchsvolle Aufgaben und Herausforderungen in Umgebungen zu meistern, die auf den ersten Blick sehr unstrukturiert oder unberechenbar wirken – wie in Spielen. Bereits in der Vergangenheit übertrafen KI-Systeme Menschen in strategisch komplexen Spielen, wie Schach oder Go[1]. Hierbei bieten Spiele eine ideale Testumgebung, um die Leistungsfähigkeit von KI greifbar und verständlich zu machen.

Ein Flipper-Spiel auf Basis von künstlicher Intelligenz stellt eine ausgezeichnete Gelegenheit dar die Prinzipien des maschinellen Lernens und der intelligenten Steuerung in einem praktischen, unterhaltsamen und interaktiven Rahmen zu demonstrieren. Dabei kommen Disziplinen wie Informatik, Mathematik und Sensorik zusammen, während gleichzeitig Mechanik und Elektronik ein integraler Bestandteil der Umsetzung sind. Mit einem KI-gesteuerten Flipper können komplexe Algorithmen zur Entscheidungsfindung und Bewegungssteuerung visualisiert und für ein breites Publikum greifbar gemacht werden. Diese Art von Projekt bietet die Möglichkeit, nicht nur theoretische Konzepte in die Praxis umzusetzen, sondern auch das Potenzial von KI in alltäglichen, spielerischen Anwendungen zu veranschaulichen.

[1] `https://arxiv.org/abs/1712.01815`, abgerufen am 30. September 2025

1.2 Zielsetzung

Ziel dieser Masterarbeit ist die Entwicklung eines autonomen KI-gesteuerten Flippers, der mittels Reinforcement Learning in der Lage ist, eigenständig zu spielen. Das System soll die Position der Kugel mithilfe von Bildverarbeitung erkennen und daraufhin die Steuerung der Flipperarme anpassen. Hierdurch soll verhindert werden, dass die Kugel durch den Drain fällt. Die zentrale Aufgabe der Arbeit besteht darin, eine KI zu entwickeln, die durch das Lernen aus Erfahrungen in der Lage ist, die optimalen Aktionen für die Steuerung der Motoren der Flipperarme zu bestimmen.
Hierfür wird Reinforcement Learning als Lernmethode eingesetzt, sodass der Flipper durch wiederholte Interaktionen mit dem Spielumfeld lernt, seine Entscheidungen zu verbessern. Dabei sollen Methoden wie das Deep Q-Learning (DQN) zum Einsatz kommen, um die Steuerung der Flipperarme zu optimieren. Ein wesentliches Ziel ist es, eine stabile und leistungsfähige Strategie zu entwickeln, die es der KI ermöglicht, in Echtzeit auf die Kugelbewegung zu reagieren und eine möglichst hohe Spielleistung zu erzielen.
Zusätzlich wird der Einsatz von Bildverarbeitung untersucht, um die Position der Kugel präzise und schnell zu erfassen. Durch die Kombination von Bildverarbeitung und Reinforcement Learning soll der Flipper in der Lage sein, sich kontinuierlich an verschiedene Spielsituationen anzupassen und ohne externe Steuerung erfolgreich zu agieren. Ziel dieser Masterarbeit ist es ein vollständig autonomes Flippersystem zu schaffen, das nicht nur die technischen Herausforderungen eines dynamischen Spiels bewältigt, sondern auch die Effektivität moderner KI-Methoden in einem praktischen und interaktiven Szenario demonstriert.

1.3 Aufbau der Arbeit

Die Masterarbeit gliedert sich in acht Kapitel, die inhaltlich aufeinander aufbauen. Nach der Einleitung, in der Motivation und Zielsetzung der Arbeit dargelegt werden, folgt in Kapitel die Darstellung der theoretischen Grundlagen. Hier werden zunächst die Funktionsweise einer Pinball-Maschine und der vorhandene Demonstrator beschrieben, bevor zentrale Konzepte der Bildverarbeitung sowie des maschinellen Lernens, neuronaler Netze und des Reinforcement Learnings erläutert werden. Damit wird das notwendige Fundament geschaffen, um die in dieser Arbeit eingesetzten Methoden und Verfahren nachvollziehen zu können.
Kapitel 3 widmet sich dem Stand der Technik. Es behandelt zunächst klassische und digitale Flipperautomaten und geht anschließend auf aktuelle Entwicklungen im Bereich der KI-gesteuerter Systeme ein. Darüber hinaus werden bereits existierende Forschungsarbeiten und Vorarbeiten aufgegriffen, die in direktem Zusammenhang mit dem hier verfolgten Ansatz stehen.
Im Anschluss daran wird in Kapitel 4 eine detaillierte Anforderungsanalyse durchgeführt.

Diese umfasst die Systemumgebung, die relevanten Stakeholder sowie die funktionalen und nicht-funktionalen Anforderungen, die an die Bildverarbeitung, die virtuelle Umgebung, die künstliche Intelligenz und den physischen Demonstrator gestellt werden.
Auf Grundlage dieser Analyse wird in Kapitel 5 das konzeptionelle Vorgehen entwickelt. Neben der Wahl der Programmierumgebung wird die Konzeption der Bildverarbeitung, der virtuellen Umgebung und der Reinforcement-Learning-Strategie beschrieben.
Die eigentliche Umsetzung wird in Kapitel 6 dargestellt. Hier werden die Entwicklungsschritte der Bildverarbeitung, der Aufbau der virtuellen Umgebung mitsamt physikalischen Eigenschaften und Kollisionsberechnungen sowie die Implementierung und das Training des RL-Agenten erläutert. Abschließend wird gezeigt, wie der Agent aus der Simulation auf den physischen Demonstrator übertragen werden kann.
Kapitel 7 dient der Evaluierung der Arbeit. Hierbei werden die Ergebnisse der Bildverarbeitung, des virtuellen Trainingsprozesses und des praktischen Einsatzes am realen Demonstrator untersucht und kritisch reflektiert.
Den Abschluss bildet Kapitel 8, in dem die erzielten Ergebnisse zusammengefasst und in den Gesamtkontext eingeordnet werden. Zudem wird ein Ausblick auf mögliche Weiterentwicklungen sowie auf weiterführende Forschungsperspektiven gegeben.

2 Grundlagen

Dieses Kapitel bietet eine Einführung in die theoretischen Grundlagen, die für das Verständnis und die Umsetzung eines durch Reinforcement Learning gesteuerten Flipperspiels notwendig sind.
Zunächst wird das klassische Flipperspiel beschrieben, welches die Grundlage für die Simulation und den physischen Demonstrator bildet. Dabei wird auf die Mechanik des Spiels sowie die Herausforderungen in Bezug auf die Steuerung der Flipperarme eingegangen.
Des Weiteren wird auf den bereits vorhandenen physischen Demonstrator, der von Dr. Hensel zur Verfügung gestellt wird, eingegangen.

Darauf folgt ein Überblick über das maschinelle Lernen und dessen Anwendungsgebiete, insbesondere im Kontext von Spielen. Dabei wird der Fokus auf die Unterschiede zwischen überwachtem Lernen, unüberwachtem Lernen und verstärkendem Lernen (Reinforcement Learning) gelegt. Darauf aufbauend werden die Konzepte künstlicher neuronaler Netzwerke erklärt, welche eine Schlüsselrolle bei der Entscheidungsfindung und Steuerung des Flipperspiels haben. Es wird auf die Architektur solcher Netzwerke sowie auf ihre wesentlichen Komponenten wie Aktivierungsfunktionen, Schichten und auf ihre Lernprozesse eingegangen.

Ebenfalls wird eine ausführliche Einführung in das Reinforcement Learning gegeben, welche die Grundlage für die Steuerung des Flipperspiels bildet. Es werden grundlegende Methoden wie die Q-Learning-Methode und die bei diesem Projekt genutzte Methode des Deep Q-Networks (DQN) vorgestellt, welche es der KI ermöglichen aus Interaktionen mit der Umgebung zu lernen und ihre Entscheidungen zu optimieren. Zudem werden Herausforderungen wie die Balance zwischen Exploration und Exploitation, die Skalierbarkeit und die Stabilität des Lernprozesses diskutiert. Diese sind insbesondere bei der Steuerung von dynamischen und physikalischen Systemen, wie einem Flipper, relevant.

2.1 Funktionsweise einer Pinball-Maschine

Eine Pinball-Maschine (auch bekannt und fortan genannt Flipper), wie in Abbildung 2.1[2] gezeigt, ist ein mechanisches Geschicklichkeitsspiel, bei dem der Spieler eine Kugel durch ein geneigtes Spielfeld steuert und dabei versucht, durch geschicktes Abschlagen der Kugel möglichst viele Punkte zu erzielen. Zentrales Ziel ist es, die Kugel so lange wie möglich im Spiel zu halten. Erreicht werden soll dies durch physische Interaktionen mit den Flipperarmen, Bumpern und anderen Spielkomponenten.

[2] `https://www.pinball-dreams.com/flipper-automaten-pinball-kaufen/custom-flipperbau-sonderanfertigungen/insm-customflipper.html`, abgerufen am 16. Juni 2025.

Das grundlegende Funktionsprinzip eines Flippers lässt sich in verschiedene mechanische und elektronische Komponenten unterteilen:

- **Spielfeld und Kugelbewegung**:
  Das Spielfeld ist geneigt, sodass die Kugel durch die Schwerkraft nach unten rollt. Zu Beginn des Spiels wird die Kugel mit einem mechanischen Kolben, dem sogenannten „Plunger“, in das Spielfeld geschossen. Fortan bewegt sich die Kugel frei auf dem Spielfeld und interagiert mit verschiedenen Hindernissen, Zielscheiben und Bumpern, die beim Aufprall Punkte generieren.

- **Flipperarme (Flipper)**:
  Die Flipperarme sind die wichtigsten Steuerungselemente im Spiel. Sie befinden sich am unteren Ende des Spielfelds und werden manuell über Tasten an der Seite des Flipperkastens betätigt. Durch das Drücken der Tasten wird ein elektromagnetischer Mechanismus aktiviert, der die Flipperarme nach oben bewegt. Der Spieler nutzt die Flipperarme, um die Kugel abzuschlagen und sie in bestimmte Bereiche des Spielfeldes zu lenken. Hierdurch soll die Kugel im Spiel gehalten und Punkte erzielt werden.

- **Bumper, Ramps und Targets**:
  Das Spielfeld ist mit verschiedenen Hindernissen und Zielen ausgestattet, die das Spielgeschehen beeinflussen. Zu den wichtigsten Elementen gehören Bumper, die die Kugel bei Kontakt mit hoher Geschwindigkeit zurückstoßen. Darüber hinaus gibt es Zielscheiben (*Targets*) und Rampen, die zusätzliche Punkte oder Boni freischalten können. Diese Elemente erhöhen die Komplexität und Dynamik des Spiels. Auch werden dem Spieler verschiedene Möglichkeiten zum Erzielen von Punkten geboten.

- **Punktesystem**:
  Das Flipperspiel verfügt über ein Punktesystem, bei dem jede Interaktion mit den Spielfeldobjekten Punkte generiert. Bestimmte Ziele oder Spielereignisse - das Treffen von Bumpern, Rampen oder speziellen Targets - können Extrapunkte, Multiball-Modi oder Bonusrunden aktivieren. Hierdurch wird das Spiel abwechslungsreich und zugleich herausfordernd.

- **Spielziel und Kugelverlust**:
  Das Hauptziel des Spiels besteht darin, möglichst viele Punkte zu sammeln, bevor die Kugel durch die untere Öffnung, der sogenannte *Drain*, fällt und das Spiel endet. Der Spieler versucht dies zu verhindern, indem er die Kugel mit den Flipperarmen immer wieder zurück ins Spielfeld befördert. Wenn die Kugel in den Drain fällt,

Abbildung 2.1: Spielfeld eines Flipperautomaten

verliert der Spieler eine Kugel. Insgesamt stehen dem Spieler meist drei Kugeln zur Verfügung. Das Spiel endet, wenn alle Kugeln verbraucht sind.

Zusammengefasst basiert das Flipperspiel auf der geschickten Steuerung der Kugel durch den Spieler, der durch gezielte Aktionen mit den Flipperarmen das Spielgeschehen beeinflusst. Die Herausforderung besteht darin, die Kugel möglichst lange im Spiel zu halten und währenddessen durch Interaktionen mit den Spielkomponenten möglichst viele Punkte erzielen.

2.2 Bildverarbeitung

Die visuelle Wahrnehmung ist ein zentrales Element für die autonome Steuerung physischer Systeme durch künstliche Intelligenz. In dieser Masterarbeit wird eine Kamera genutzt, um den Zustand eines Flipperautomaten zu erfassen und auf Basis der Bilddaten eine Entscheidungsfindung durch einen Reinforcement-Learning-Agenten zu ermöglichen. Hierzu bietet die OpenCV-Bibliothek (Open Source Computer Vision Library) ein umfassendes Set an Werkzeugen zur effizienten Verarbeitung solcher Bilddaten.

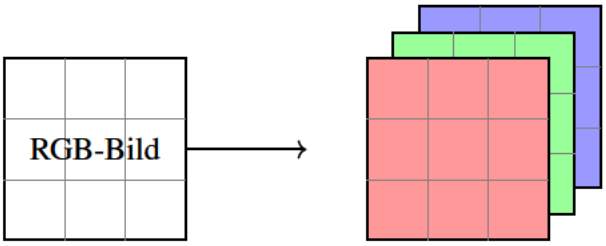


Abbildung 2.2: Darstellung eines RGB-Bildes (links), das in seine drei Farbkanäle rot, grün und blau zerlegt wird (rechts), jeweils mit einer schematischen 3x3-Matrix

2.2.1 OpenCV

OpenCV ist eine der bekanntesten und am weitesten verbreiteten Bibliotheken für Bildverarbeitung und Computer Vision. Die Entwicklung begann im Jahr 1999 bei Intel mit dem Ziel, eine frei verfügbare, performante und plattformübergreifende Sammlung von Werkzeugen für die Bildanalyse und das maschinelle Sehen bereitzustellen. Seitdem hat sich OpenCV kontinuierlich weiterentwickelt und wird heute von einer aktiven Open-Source-Community gepflegt [Fou25]. Die Bibliothek ist in C++ geschrieben, bietet aber Schnittstellen zu vielen anderen Programmiersprachen wie Python, Java und MATLAB, was ihre Verbreitung im wissenschaftlichen und industriellen Umfeld stark gefördert hat [PEM+20].

Die Einsatzmöglichkeiten von OpenCV sind äußerst vielfältig. Sie reichen von grundlegenden Operationen der Bildverarbeitung, wie Glättung, Filterung oder Farbtransformationen, über die Segmentierung und Objekterkennung bis hin zu hochkomplexen Anwendungen wie Gesichtserkennung, Bewegungserfassung, 3D-Rekonstruktion oder Deep-Learning-basierter Objektdetektion. Darüber hinaus kann OpenCV auch in ressourcenkritischen Anwendungen wie Robotik, autonomen Fahrzeugen oder Augmented Reality eingesetzt werden[3].

Das Fundament von OpenCV bildet die Behandlung von Bilddaten in Form von Matrizen. Praktisch jede Operation in OpenCV basiert auf Matrixberechnungen[4]. Ein digitales Bild wird dabei als zweidimensionale oder, im Falle von Farbbildern, dreidimensionale Matrix gespeichert. Jedes Element dieser Matrix repräsentiert einen Bildpunkt, also ein Pixel, und enthält dessen Intensitäts- oder Farbwerte.

Im einfachsten Fall eines Graustufenbildes handelt es sich um eine zweidimensionale Matrix, bei der jede Matrixzelle einen Wert zwischen 0 und 255 speichert. Der Wert 0 steht für Schwarz, 255 für Weiß und die Werte dazwischen für verschiedene Graustufen. Farbbilder, wie sie im üblichen RGB-Format vorliegen, werden hingegen als dreidimensionale Matrizen dargestellt. Wie in Abbildung 2.2 enthält jede Pixelposition drei Werte – einen

[3] `https://opencv.org/about/`, abgerufen am 15. September 2025.

[4] `https://docs.opencv.org/4.x/d6/d6d/tutorial_mat_the_basic_image_container.html`, abgerufen am 15. September 2025.

für den Rot-, einen für den Grün- und einen für den Blauanteil. Diese drei Werte bestimmen gemeinsam die endgültige Farbe des Pixels[5].
Durch diese Matrixdarstellung können Bildoperationen sehr effizient formuliert und ausgeführt werden, da sich viele Verfahren der Bildverarbeitung auf lineare Algebra und Signalverarbeitung zurückführen lassen. Beispielsweise entspricht die Anwendung eines Filters einer Faltung (Convolution) der Bildmatrix mit einem kleineren Filterkernel, ebenfalls einer Matrix. Diese mathematische Grundlage macht OpenCV sowohl flexibel als auch leistungsfähig, da sie den direkten Zugriff auf Pixelwerte erlaubt und gleichzeitig hochoptimierte Matrixoperationen nutzt [PEM+20].

2.2.2 Digitale Bilder und Farbräume

Digitale Bilder bestehen aus Pixeln, deren Farbwerte typischerweise in Farbräumen wie RGB, Grayscale oder HSV repräsentiert werden. Für viele bildverarbeitende Aufgaben, wie z. B. die Objekterkennung oder Segmentierung, wird gegebenenfalls eine Umwandlung in andere Farbräume vorgenommen, um bestimmte Merkmale hervorzuheben.

2.2.3 Bildvorverarbeitung

Die Bildvorverarbeitung ist ein essenzieller Schritt in der visuellen Datenanalyse. Sie ist nützlich, um Rohdaten aus der Kamera in eine Form zu überführen, die für nachfolgende Prozesse wie Objekterkennung, Segmentierung oder Zustandsableitung durch einen Reinforcement-Learning-Agenten geeignet ist. Hierbei sollen Störungen reduziert, relevante Bildmerkmale verstärkt und die Konsistenz der Daten erhöht werde.
Im Kontext eines Flipperautomaten ergeben sich nachfolgende typische Herausforderungen:

- Bildrauschen durch schwankende Lichtverhältnisse,
- Bewegungsunschärfe bei schneller Kugelbewegung,
- Reflexionen durch glänzende Spielflächen und
- Geringer Kontrast zwischen Kugel und Hintergrund.

Rauschunterdrückung (Blurring):
Ein Gaußscher Weichzeichner wird verwendet, um hochfrequentes Rauschen zu reduzieren. Dieser stabilisiert insbesondere die nachfolgende Kanten- oder Konturenerkennung. Abbildung 2.3[6] zeigt die Rauschunterdrückung durch die Funktion `cv2.GaussianBlur()`.

[5] `https://docs.opencv.org/3.4/de/d25/imgproc_color_conversions.html`, abgerufen am 15. September 2025.
[6] `https://docs.opencv.org/4.x/d4/d13/tutorial_py_filtering.html`, abgerufen am 15. September 2025.

Abbildung 2.3: Links das Originalbild und rechts nach dem Anwenden des Filters `GaussianBlur()`

Abbildung 2.4: Links das Originalbild und rechts nach dem Anwenden des Filters `Canny()`

Kantenerkennung (z. B. Canny-Filter):
Der Canny-Filter ist ein Kanten-Detektionsalgorithmus in der Bildverarbeitung. Er dient dazu, die Ränder bzw. Konturen von Objekten in einem Bild präzise zu erkennen. Das ist besonders nützlich, wenn man z. B. Objekte isolieren, zählen oder weiterverarbeiten möchte. Der Canny-Algorithmus detektiert, wie in Abbildung 2.4[7] gezeigt, signifikante Helligkeitsübergänge.

Morphologische Operationen:
Mit Erosion, Dilatation und Kombinationen wie Öffnung (z. B.: `MORPH_OPEN`) können kleine Störungen entfernt und Objektkonturen geglättet werden.

Die vorgestellten Bildvorverarbeitungsschritte bilden eine robuste Grundlage für die folgende visuelle Erkennung der Flipperkugel sowie weiterer relevanter Spielfeldobjekte. Für

[7] `https://www.bogotobogo.com/python/OpenCV_Python/python_opencv3_Image_Canny_Edge_Detection.php`, abgerufen am 15. September 2025.

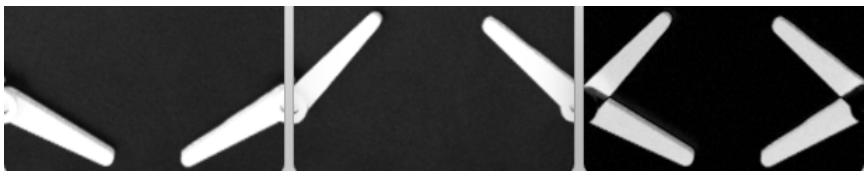


Abbildung 2.5: Differenzbild: Flipper inaktiv (links), Flipper aktiv (mittig) und Differenzbild (rechts)

ein durch Reinforcement Learning unterstütztes System ist die Qualität der Zustandserkennung entscheidend. Eine gut abgestimmte Bildverarbeitung trägt daher zur Lern- und Spielperformance bei.

Objekterkennung und Tracking

Ein zentrales Ziel der Bildverarbeitung im vorliegenden Projekt ist die zuverlässige Erkennung und Verfolgung der Flipperkugel. Die Position der Kugel stellt eine wesentliche Komponente des Systemzustands dar, die dem Reinforcement-Learning-Agenten als Eingabe zu darauf aufbauenden Maßnahmen zur Entscheidungsfindung dient.

Kugelerkennung mittels Hough-Transformation

Eine robuste Methode zur Detektion kreisförmiger Objekte wie Flipperkugeln ist die Hough-Kreis-Transformation.

Konturbasierte Segmentierung

Durch Konturerkennung lassen sich zusammenhängende, kontrastreiche Bildbereiche identifizieren.

Tracking der Kugelposition

Alternativ kann für eine kontinuierliche Zustandsschätzung anstelle der selbständigen Kugelerkennung und -verfolgung auch ein Objekttracker (z. B.: CSRT) verwendet werden.

Differenzbilder

Eine weitere Möglichkeit, bewegte Objekte zu erkennen, ist die Berechnung von Differenzbildern. Hierbei wird das aktuelle Kamerabild, in dem sich die Flipperarme bewegen, mit einem Referenzbild (z. B. Ruhelage der Flipperarme) subtrahiert. Abbildung 6.11 zeigt das Differenzbild zwischen Flipperarmen in der Ruhelage und aktivierten Position. Helle Pixel im Differenzbild markieren Regionen, die sich seit dem Referenzbild verändert haben, was typischerweise bewegte Objekte wie die Flipperkugel hervorhebt.
Differenzbilder sind besonders nützlich, wenn sich Objekte dynamisch bewegen, der Hintergrund statisch ist und eine schnelle, einfache Erkennung ohne komplexe Merkmalsextraktion erfolgen soll.

2.3 Maschinelles Lernen

Maschinelles Lernen (ML) ist ein Teilgebiet der Künstlichen Intelligenz. Diese zielt darauf ab Systeme zu entwickeln, die aus Daten lernen und auf dieser Basis Entscheidungen treffen oder Vorhersagen machen können. Im Gegensatz zu klassischen, regelbasierten Programmen beruht maschinelles Lernen auf statistischen Verfahren, welche aus Beispielen lernen. Es ist somit nicht notwendig explizite Anweisungen für jede denkbare Situation vorzugeben [Bis06].
Grundlegend unterscheidet man drei Arten von Lernverfahren: überwachtes Lernen, unüberwachtes Lernen und bestärkendes Lernen [Dom12].

Beim **überwachten Lernen** wird ein Modell mit Daten trainiert, bei denen sowohl die Eingaben als auch die zugehörigen Ausgaben (Labels) bekannt sind. Ziel ist es, eine Funktion zu erlernen, die auch für neue, bisher ungesehene Daten korrekte Ausgaben liefern kann. Typische Anwendungen sind Klassifikations- und Regressionsaufgaben [MRT18].

Das **unüberwachte Lernen** kommt ohne bekannte Ausgaben aus. Es wird genutzt, um Muster, Strukturen oder Gruppierungen innerhalb von Daten zu identifizieren. Ein typisches Beispiel hierfür ist das Clustering, bei dem ähnliche Datenpunkte zu Gruppen zusammengefasst werden.

Das **bestärkende Lernen** beschreibt einen Lernprozess, bei dem ein Agent durch Interaktion mit einer Umgebung lernt. Der Agent erhält Belohnungen oder Bestrafungen in Abhängigkeit von seinen Handlungen und versucht so, eine optimale Strategie zu erlernen. Dieses Verfahren findet beispielsweise in der Robotik oder bei der Entwicklung intelligenter Spielstrategien Anwendung [GBC16a].

Zu den klassischen Algorithmen im maschinellen Lernen zählen Entscheidungsbäume, Support Vector Machines (SVM), k-Nearest-Neighbor (k-NN) sowie lineare und logistische Regression. In den letzten Jahren haben sich jedoch auch komplexere Modelle wie künstliche neuronale Netze und Deep Learning durchgesetzt, insbesondere bei Aufgaben mit großen Datenmengen und hoher Komplexität [LBH15a].

Zu beachtende Aspekte sind die Datenvorverarbeitung, wie Normalisierung oder Feature-Engineering, sowie die Auswahl geeigneter Metriken zur Bewertung der Modellleistung, wie etwa Genauigkeit (Accuracy), Precision, Recall oder der F1-Score.

Ein tieferes Verständnis dieser Grundlagen ist essenziell, um die Stärken und Schwächen, sowie die Einsatzmöglichkeiten von maschinellen Lernverfahren korrekt einschätzen und

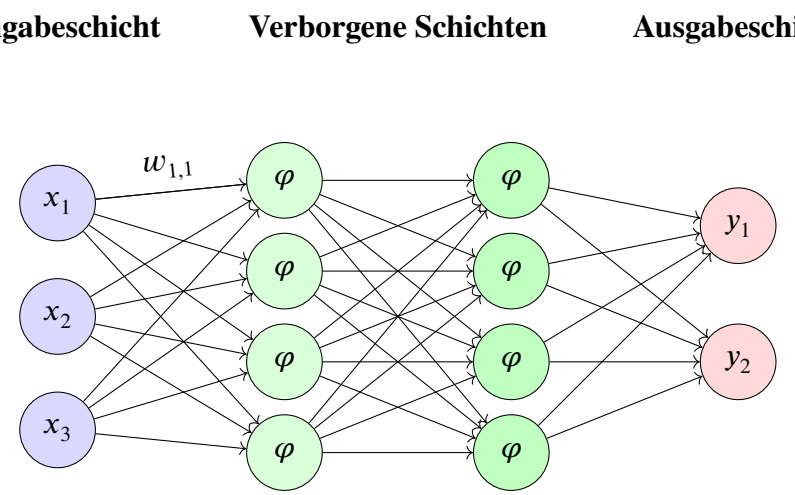


Abbildung 2.6: Darstellung eines neuronalen Netzes mit zwei verborgenen Schichten

gezielt anwenden zu können [Bis06; GBC16a; Dom12].

2.4 Künstliche neuronale Netzwerke

Künstliche neuronale Netzwerke (KNN) sind mathematische Modelle, die von der Funktionsweise biologischer Nervenzellen inspiriert sind. Ein einzelnes künstliches Neuron erhält eine Menge an Eingaben $x_1, x_2, \ldots, x_n$, die jeweils mit einem Gewicht $w_1, w_2, \ldots, w_n$ multipliziert werden. Die gewichtete Summe der Eingaben wird um einen Bias-Wert b ergänzt und anschließend durch eine nichtlineare Aktivierungsfunktion $\varphi(\cdot)$ transformiert:

$$y = \varphi\left(\sum_{i=1}^{n} w_i x_i + b\right). \tag{2.1}$$

Die Aktivierungsfunktion bestimmt, ob das Neuron „feuert“ und erlaubt es, nichtlineare Zusammenhänge zu modellieren. Typische Funktionen sind die Sigmoid-, Tanh- oder ReLU-Funktion.

Durch die Zusammenschaltung vieler Neuronen entstehen Schichten. Ein Feedforward - Netzwerk besteht aus einer Eingabeschicht, mehreren verborgenen Schichten und einer Ausgabeschicht. Informationen fließen dabei von links nach rechts durch das Netzwerk. Während des Trainings werden die Gewichte so angepasst, dass der Fehler zwischen den vorhergesagten und den tatsächlichen Ausgaben minimiert wird. Dies geschieht mithilfe des Backpropagation-Algorithmus in Kombination mit Gradientenabstiegsverfahren, die die Gewichte schrittweise optimieren [DY20].

Tiefe neuronale Netze (Deep Neural Networks, DNN) bestehen häufig aus vielen hintereinander geschalteten Schichten und sind in der Lage, hochkomplexe Muster und Repräsentationen zu lernen. Als Beispiel zeigt Abbildung 2.6 eine vereinfachte Darstellung eines neuronalen Netzes. Sie bilden die Grundlage moderner Anwendungen in der Bild- und Sprachverarbeitung, Robotik oder Medizin [SHP23]. Gleichzeitig stellt die geringe Inter-

pretierbarkeit der Modelle eine Herausforderung dar, weshalb Methoden der erklärbaren KI zunehmend in den Fokus rücken [Ras+22].

2.4.1 Deep Learning

Deep Learning ist ein Teilgebiet des maschinellen Lernens, das auf tiefen neuronalen Netzwerken basiert. Der Begriff „Deep“bezeichnet die Tatsache, dass es mindestens zwei verborgenen Schichten im neuronalen Netz gibt, die zwischen Eingabe- und Ausgabeschicht liegen. Durch die Verwendung vieler Schichten können hierarchische Merkmalsrepräsentationen gelernt werden: niedrige Schichten erfassen einfache Merkmale, höhere Schichten abstrahieren komplexe Konzepte [DY20].
Typische Architekturen sind Feedforward-Netze (FNN), Convolutional Neural Networks (CNN), Recurrent Neural Networks (RNN) und Transformer. Der Lernprozess basiert auf Forward-Propagation, Fehlerberechnung mittels Loss-Funktion, Backpropagation und Gradientenabstieg [SHP23].
Deep Learning ermöglicht automatische Feature-Extraktion, hohe Genauigkeit und vielseitige Anwendungen, ist jedoch daten- und rechenintensiv und weniger interpretierbar [Ras+22].

2.4.2 Architekturen von künstlichen neuronalen Netzen

Die Architektur eines künstlichen neuronalen Netzes beschreibt den strukturellen Aufbau des Modells, d. h. die Anordnung der Neuronen in Schichten sowie die Art ihrer Verbindungen. Unterschiedliche Architekturen eignen sich für unterschiedliche Problemstellungen und haben die Entwicklung des Deep Learnings entscheidend geprägt.

Das Perzeptron

Eines der frühesten Modelle ist das von Rosenblatt [Ros58] entwickelte *Perzeptron*. Wie in Abbildung 2.7 gezeigt, besteht das Perzeptron aus einer Eingabeschicht, deren Werte mit Gewichten multipliziert und aufsummiert werden. Anschließend erfolgt eine Transformation durch eine Aktivierungsfunktion, die das Neuron aktiviert oder inaktiv lässt. Mathematisch lässt sich das Perzeptron wie folgt beschreiben:

$$y = \varphi\left(\sum_{i=1}^{n} w_i x_i + b\right) \tag{2.2}$$

Hierbei stellen x_i die Eingaben, w_i die Gewichte, b der Bias und φ die Aktivierungsfunktion dar. Das Perzeptron ist jedoch nur in der Lage, linear trennbare Probleme zu lösen, was seine Anwendbarkeit einschränkt.

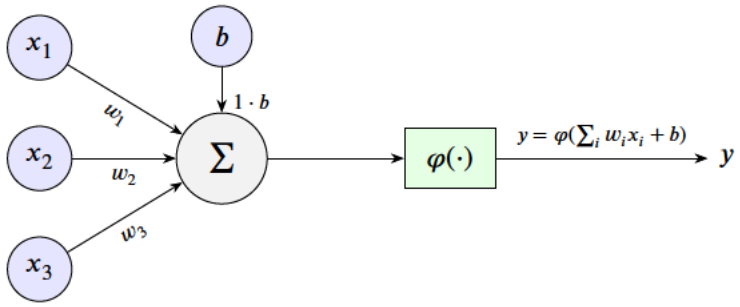


Abbildung 2.7: Schematische Darstellung eines Perzeptrons mit drei Eingängen, Bias b, Gewichten w_i und Aktivierungsfunktion φ.

Multilayer-Perzeptron (MLP)

Um auch nichtlinear trennbare Probleme zu bewältigen, wurde das Perzeptron durch das *Multilayer-Perzeptron (MLP)* erweitert. Hierbei werden mehrere Schichten von Neuronen hintereinandergeschaltet, sodass eine Hierarchie von Merkmalsrepräsentationen entsteht. MLPs bilden die Grundlage moderner Feedforward-Netzwerke und sind in vielen klassischen Klassifikations- und Regressionsaufgaben einsetzbar [GBC16b].

Convolutional Neural Networks (CNN)

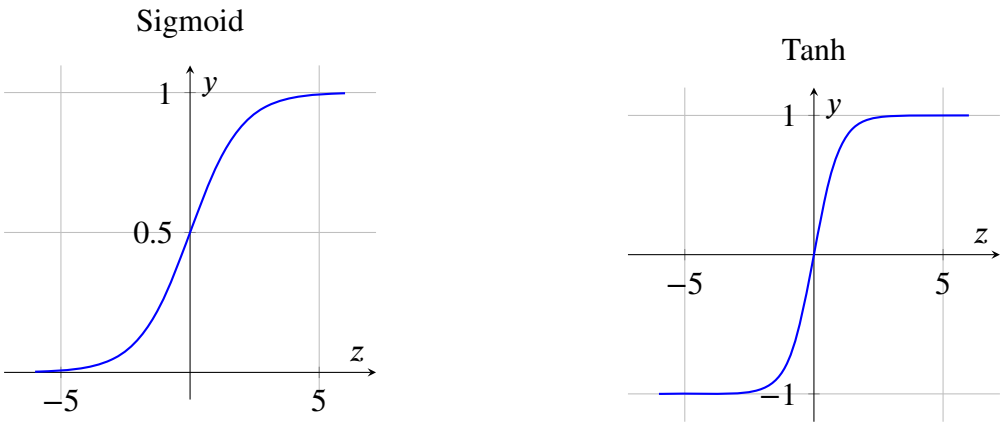
Für Bild- und Videoverarbeitung haben sich *Convolutional Neural Networks (CNNs)* als besonders leistungsfähig erwiesen. Sie nutzen Faltungsoperationen, um lokale Merkmale in Daten zu extrahieren und sind durch Gewichts-Sharing und lokale Rezeptive Felder besonders effizient. CNNs stellen die Grundlage für viele Anwendungen in der Computer Vision dar, wie Objekterkennung, Segmentierung und Bildklassifikation [LBH15b].

Recurrent Neural Networks (RNN)

Zur Verarbeitung sequentieller Daten wurden *Recurrent Neural Networks (RNNs)* entwickelt. Sie besitzen Rückkopplungen, die Informationen über frühere Zeitpunkte speichern können. Erweiterungen wie Long Short-Term Memory (LSTM) und Gated Recurrent Units (GRU) adressieren das Problem verschwindender Gradienten und haben insbesondere in der Sprachverarbeitung große Bedeutung [HS97; Cho+14].

2.4.3 Aktivierungsfunktionen in neuronalen Netzwerken

Aktivierungsfunktionen sind ein zentrales Element neuronaler Netzwerke, da sie den Neuronen ermöglichen *nichtlineare Zusammenhänge* zu modellieren. Ohne Aktivierungsfunktionen wären Netzwerke aus mehreren Schichten lediglich eine lineare Kombination der Eingaben und könnten keine komplexen Muster lernen [GBC16b; LBH15b].



(a) Sigmoid (b) Tanh

Abbildung 2.8: Sigmoid- und Tanh-Aktivierungsfunktion

Eine Aktivierungsfunktion transformiert die gewichtete Summe der Eingaben eines Neurons in die Ausgabe $y = \varphi(z)$, wobei w_i die Gewichte, x_i die Eingaben und b der Bias des Neurons sind:

$$z = \sum_{i=1}^{n} w_i x_i + b. \tag{2.3}$$

Sigmoid:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{2.4}$$

Die Sigmoidfunktion in Abbildung 2.8 mit dem Wertebereich (0,1) ist nützlich zur Ausgabe von Wahrscheinlichkeiten, jedoch anfällig für das *vanishing-gradient*-Problem, insbesondere in tiefen Netzwerken [GB10a].

Tanh (Hyperbolische Tangens):

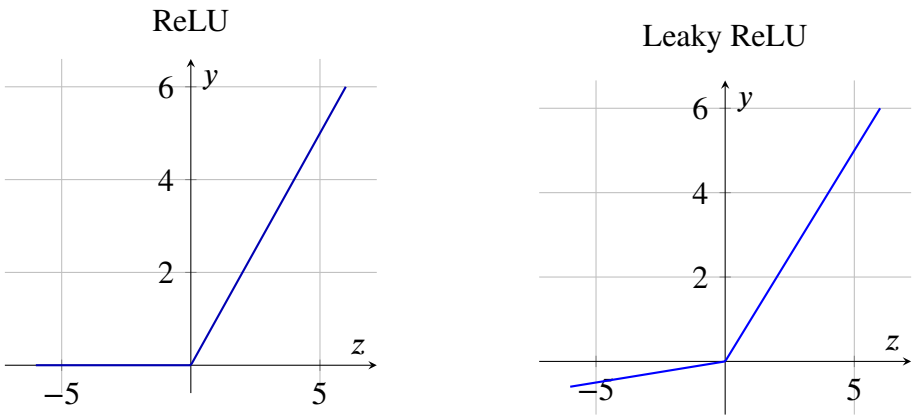
$$\tanh(z) = \frac{e^{z} - e^{-z}}{e^{z} + e^{-z}} \tag{2.5}$$

Mit einem Wertebereich von $(-1,1)$ hat der hyperbolische Tangens rechts in Abbildung 2.8 einen mittleren Wert nahe 0. Hierdurch wird das Training stabilisiert, ist jedoch anfällig für das Verschwinden der Gradienten bei tiefen Netzen. [GBC16b].

ReLU (Rectified Linear Unit):

$$\mathrm{ReLU}(z) = \max(0, z). \tag{2.6}$$

Links in der Abbildung 2.9 wird die Funktion ReLU gezeigt, welche einen Wertebereich von $[0, \infty)$ aufweist. Diese ist in modernen Netzwerken sehr beliebt, da die Berechnung



(a) ReLU (b) Leaky ReLU

Abbildung 2.9: ReLU- und Leaky ReLU-Aktivierungsfunktion

einfach ist und das Vanishing-Gradient-Problem reduziert wird. Nachteil sind jedoch die „Dead Neurons“, die dauerhaft inaktiv bleiben können [NH10].

Leaky ReLU / Parametric ReLU:

$$\text{LeakyReLU}(z) = \begin{cases} z, & z > 0 \\ \alpha z, & z \leq 0 \end{cases} \tag{2.7}$$

Bei der Funktion Leaky ReLU in Abbildung 2.9 (rechts) werden kleine negative Werte weitergeleitet, wodurch tote Neuronen vermieden werden. α ist ein kleiner Parameter (z. B. 0.1) [He+15].

Zusammenfassung

Die Wahl der Aktivierungsfunktion hängt stark von der Problemstellung ab. In den versteckten Schichten moderner Netzwerke werden meist ReLU oder Varianten (Leaky ReLU, PReLU) verwendet, während am Ausgang Sigmoid (binäre Klassifikation) oder Softmax (Multi-Klassen-Klassifikation) üblich sind. Die Aktivierungsfunktionen spielen eine entscheidende Rolle, um die Lernfähigkeit und Stabilität des Netzwerks zu gewährleisten.

2.4.4 Gradientenverfahren

Das Gradientenverfahren (engl. *Gradient Descent*) ist ein zentrales Optimierungsverfahren beim Training neuronaler Netzwerke. Ziel ist es, die Parameter des Netzwerks, also Gewichte W und Biases b, so anzupassen, dass die Verlustfunktion $L(\theta)$ minimal wird. Diese Verlustfunktion misst, wie gut das Netzwerk die Trainingsdaten modelliert [GBC16b].

Die Methode basiert auf der Information, die der Gradient der Verlustfunktion liefert. Für den Parametervektor θ gilt die iterative Aktualisierung:

$$\theta_{t+1} = \theta_t - \eta \nabla_\theta L(\theta_t). \tag{2.8}$$

Hierbei ist η die Lernrate, die die Schrittweite der Aktualisierung bestimmt und $\nabla_\theta L(\theta_t)$ den Gradienten der Verlustfunktion zum Zeitpunkt t bezeichnet. Der Gradient zeigt die Richtung des steilsten Anstiegs der Funktion, weshalb durch Subtraktion des Gradienten die Bewegung in Richtung Minimum erfolgt. Auf diese Weise „rollt“ der Parametervektor quasi den Talboden der Verlustlandschaft entlang, bis ein Minimum erreicht ist.
Je nach Berechnung des Gradienten unterscheidet man verschiedene Varianten. Beim *Batch Gradient Descent* wird der Gradient über alle Trainingsdaten berechnet, sodass stabile, aber langsame Updates die Folge sind. Beim *Stochastic Gradient Descent (SGD)* wird der Gradient für jeweils ein einzelnes Trainingsbeispiel berechnet. Dies ermöglicht schnellere Updates, führt jedoch zu stärker schwankenden Bewegungen. Eine gängige Praxis ist das *Mini-Batch Gradient Descent*, bei dem der Gradient über kleine Teilmengen der Trainingsdaten berechnet wird. Diese Methode vereint Vorteile von Batch- und Stochastic Gradient Descent und wird häufig in modernen neuronalen Netzwerken eingesetzt.
Das Gradientenverfahren ist für neuronale Netzwerke von zentraler Bedeutung, da es das Optimieren der Gewichte ermöglicht und damit die Lernfähigkeit der Netzwerke sicherstellt. Durch die iterative Anpassung der Parameter können komplexe nichtlineare Funktionen approximiert werden, die für Deep Learning und viele Anwendungen in der Bild- oder Sprachverarbeitung essentiell sind [LBH15b; GBC16b].

Mini-Batch Gradient Descent

Beim *Mini-Batch Gradient Descent* wird der Gradient der Verlustfunktion nicht über alle Trainingsdaten berechnet, wie beim klassischen *Batch Gradient Descent*, sondern über kleine Teilmengen der Daten, sogenannte *Mini-Batches* [GBC16b]. Ein Mini-Batch besteht typischerweise aus 32 bis 256 Trainingsbeispielen, je nach Größe des Datensatzes und Speicherkapazität der Hardware.

Durch diese Aufteilung wird das Training beschleunigt, da die Berechnung des Gradienten für eine Teilmenge wesentlich schneller ist als für den gesamten Datensatz. Gleichzeitig bleiben die Updates relativ stabil, da der Mittelwert über die Mini-Batch-Daten den Gradienten verrauscht, aber nicht zu stark schwankt. Dieses leichte Rauschen kann sogar vorteilhaft sein, da es hilft, kleine lokale Minima zu überwinden und die Konvergenz zu verbessern.

Formal wird die Parameteraktualisierung in einem Mini-Batch B_t wie folgt beschrieben:

$$\theta_{t+1} = \theta_t - \eta \frac{1}{|B_t|} \sum_{i \in B_t} \nabla_\theta L_i(\theta_t) \tag{2.9}$$

Hierbei ist $|B_t|$ die Anzahl der Trainingsbeispiele im Mini-Batch und $L_i(\theta_t)$ bezeichnet den Verlust für das einzelne Beispiel i.

Lokale Minima

Ein zentrales Problem beim Gradientenverfahren ist das Auftreten von lokalen Minima. Ein lokales Minimum ist ein Punkt in der Verlustlandschaft, an dem die Funktion einen kleineren Wert hat als in seiner unmittelbaren Umgebung, aber nicht unbedingt den global minimalen Wert. Das Gradientenverfahren folgt stets dem steilsten Abstieg. Dies bedeutet, dass das Netzwerk möglicherweise in einem lokalen Minimum „hängenbleibt“, ohne das globale Minimum zu erreichen [GBC16b].

Das Gradientenverfahren kann in steilen oder flachen Regionen der Verlustlandschaft sehr langsam konvergieren oder stark schwanken. Um dies zu mildern, wird häufig das Konzept des *Moment* eingesetzt [**qian_Moment_1999**].

Das Moment fügt einen Anteil der vorherigen Parameteraktualisierung zur aktuellen hinzu, wodurch die Bewegung in Richtung Minimum „beschleunigt“ und Oszillationen in steilen Tälern reduziert werden. Formal wird die Aktualisierung der Parameter θ mit dem Moment durch folgende Gleichungen beschrieben:

$$v_{t+1} = \beta v_t - \eta \nabla_\theta L(\theta_t) \tag{2.10}$$

$$\theta_{t+1} = \theta_t + v_{t+1} \tag{2.11}$$

Hier ist v_t die geschwindigkeitsähnliche Variable, die sich über die Iterationen akkumuliert, η die Lernrate und $\beta \in [0,1)$ der Moment-Faktor ist. Typische Werte für β liegen zwischen 0.8 und 0.99.
Dadurch wird das Lernen in flachen Regionen beschleunigt und die Parameter „überrollen“ kleine lokale Minima oder Sattelpunkte, an denen sonst der Gradient nahe Null wäre. Das Moment ist besonders effektiv bei tiefen neuronalen Netzwerken mit komplexen, nichtkonvexen Verlustlandschaften [GBC16b].

2.4.5 Backpropagation (Fehlerrückführung)

Backpropagation ist der zentrale Algorithmus, mit dem neuronale Netzwerke lernen, die Gewichte und Biases so anzupassen, dass Fehler auf den Trainingsdaten minimiert werden [GBC16b]. Der Algorithmus basiert auf der *Kettenregel* der Differentialrechnung und ermöglicht eine effiziente Berechnung der Gradienten der Verlustfunktion.

Forward Pass

Zunächst erfolgt der *Forward Pass*. Für ein Neuron j in Schicht l:

$$z_j^{(l)} = \sum_i w_{ji}^{(l)} a_i^{(l-1)} + b_j^{(l)}, \tag{2.12}$$

$$a_j^{(l)} = f(z_j^{(l)}) \tag{2.13}$$

Hierbei ist $w_{ji}^{(l)}$ das Gewicht von Neuron i in Schicht $(l-1)$ zu Neuron j in Schicht l, $b_j^{(l)}$ der Bias, $a_i^{(l-1)}$ die Aktivierung des vorherigen Neurons und f die Aktivierungsfunktion.

Fehlerberechnung an der Ausgabeschicht

Für die Ausgabeschicht L wird der Fehler $\delta_j^{(L)}$ wie folgt berechnet:

$$\delta_j^{(L)} = \frac{\partial L}{\partial z_j^{(L)}} = \frac{\partial L}{\partial a_j^{(L)}} \cdot f'(z_j^{(L)}), \tag{2.14}$$

wobei L die Verlustfunktion ist und f' die Ableitung der Aktivierungsfunktion.

Backpropagation durch verborgene Schichten

Für jede verborgene Schicht l wird der Fehler von der nächsten Schicht $(l+1)$ zurückpropagiert:

$$\delta_j^{(l)} = \left(\sum_k w_{kj}^{(l+1)} \delta_k^{(l+1)} \right) \cdot f'(z_j^{(l)}), \tag{2.15}$$

wobei $w_{kj}^{(l+1)}$ die Verbindung von Neuron j in Schicht l zu Neuron k in Schicht $(l+1)$ ist. So wird der Beitrag jedes Neurons zum Gesamtausgabe-Fehler nachvollzogen.

Gradienten und Gewichtsupdate

Die Gradienten der Gewichte und Biases ergeben sich aus:

$$\frac{\partial L}{\partial w_{ji}^{(l)}} = a_i^{(l-1)} \cdot \delta_j^{(l)}, \quad \frac{\partial L}{\partial b_j^{(l)}} = \delta_j^{(l)}. \tag{2.16}$$

Die Aktualisierung der Gewichte erfolgt dann typischerweise mit Gradient Descent:

$$w_{ji}^{(l)} \leftarrow w_{ji}^{(l)} - \eta \frac{\partial L}{\partial w_{ji}^{(l)}}, \quad b_j^{(l)} \leftarrow b_j^{(l)} - \eta \frac{\partial L}{\partial b_j^{(l)}}, \tag{2.17}$$

wobei η die Lernrate ist.

2.4.6 Verlustfunktionen

Verlustfunktionen (engl. *loss functions*) spielen eine zentrale Rolle beim Training neuronaler Netzwerke. Sie quantifizieren den Unterschied zwischen den vom Modell vorhergesagten Ausgaben und den tatsächlichen Zielwerten. Der Wert der Verlustfunktion dient als Maß für die Genauigkeit des Modells und bestimmt die Richtung, in die die Gewichte während des Trainings angepasst werden [GBC16b].

Mean Squared Error (MSE)

Für Regressionsprobleme, bei denen kontinuierliche Zielwerte vorhergesagt werden, ist die *Mean Squared Error* (MSE) sehr gebräuchlich. Sie berechnet den durchschnittlichen quadratischen Unterschied zwischen den vorhergesagten Werten $\hat{y}_i$ und den tatsächlichen Werten y_i:

$$L_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2. \tag{2.18}$$

MSE ist sensitiv gegenüber Ausreißern, da große Fehler quadratisch bestraft werden.

Cross-Entropy Loss

Bei Klassifikationsproblemen, bei denen das Modell Kategorien vorhersagt, ist die *Cross-Entropy*-Lossfunktion weit verbreitet. Für ein Beispiel i mit C Klassen lautet sie:

$$L_{\text{CE}} = -\sum_{c=1}^{C} y_{i,c} \log(\hat{y}_{i,c}), \tag{2.19}$$

wobei $y_{i,c}$ der binäre Indikator ist, ob Klasse c korrekt ist, und $\hat{y}_{i,c}$ die vorhergesagte Wahrscheinlichkeit für Klasse c. Cross-Entropy misst, wie gut die Wahrscheinlichkeitsverteilung des Modells die tatsächlichen Labels approximiert und wird häufig in neuronalen Netzen für Klassifikationsaufgaben genutzt.

Mean Absolute Error (MAE)

Mean Absolute Error (MAE) ist eine Alternative zur MSE für Regressionen, die weniger empfindlich gegenüber Ausreißern ist:

$$L_{\text{MAE}} = \frac{1}{n} \sum_{i=1}^{n} |\hat{y}_i - y_i|. \tag{2.20}$$

MAE bestraft große Fehler linear statt quadratisch und wird eingesetzt, wenn robuste Schätzungen gewünscht sind.

2.4.7 Optimierer

Optimierer sind Algorithmen, die die Gewichte und Biases neuronaler Netzwerke während des Trainings anpassen, um die Verlustfunktion zu minimieren. Sie bestimmen, wie die Gradienten, die durch Backpropagation berechnet werden, in konkrete Parameteraktualisierungen umgesetzt werden. Die Wahl des Optimierers beeinflusst die Konvergenzgeschwindigkeit, Stabilität und letztlich die Genauigkeit des Modells [GBC16b].

RMSProp

RMSProp passt die Lernrate adaptiv für jedes Gewicht an, basierend auf dem gleitenden Mittelwert der quadrierten Gradienten:

$$v_t = \beta v_{t-1} + (1 - \beta)(\nabla_\theta L_{B_t}(\theta_t))^2, \tag{2.21}$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{v_t + \epsilon}} \nabla_\theta L_{B_t}(\theta_t), \tag{2.22}$$

wobei ϵ eine kleine Zahl zur Stabilisierung ist. RMSProp verhindert, dass Parameter mit häufig großen Gradienten zu schnell aktualisiert werden [TH12].

AdaDelta

AdaDelta ist eine Weiterentwicklung von RMSProp und verzichtet auf die explizite Lernrate. Sie verwendet die gleitende Mittelung der vergangenen Updates, um die Schrittgröße dynamisch anzupassen:

$$\Delta\theta_t = -\frac{\sqrt{E[\Delta\theta^2]_{t-1} + \epsilon}}{\sqrt{E[g^2]_t + \epsilon}} g_t, \tag{2.23}$$

wobei g_t der aktuelle Gradient ist. Dadurch wird die Notwendigkeit einer manuellen Lernrate reduziert [Zei12].

Adam

Adam kombiniert die Vorteile von Moment und RMSProp, indem es sowohl die ersten Momente (Mittelwert) als auch die zweiten Momente (Varianz) der Gradienten verfolgt:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t, \quad v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2, \tag{2.24}$$

$$\hat{m}_t = \frac{m_t}{1-\beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1-\beta_2^t}, \tag{2.25}$$

$$\theta_{t+1} = \theta_t - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}. \tag{2.26}$$

Adam ist robust gegenüber verrauschten Gradienten und wird in vielen modernen neuronalen Netzwerken standardmäßig eingesetzt [KB15].

2.4.8 Gewichtsinitialisierung

Die Initialisierung der Gewichte ist ein entscheidender Faktor für den erfolgreichen Trainingsprozess neuronaler Netze und spielt im Reinforcement Learning eine besonders wichtige Rolle. Da Rückmeldungen aus der Umgebung häufig spärlich oder verzögert auftreten, ist das Lernsignal weniger stabil als in klassischen überwachten Lernverfahren. Eine ungeeignete Wahl der Startgewichte kann dazu führen, dass Gradienten verschwinden oder explodieren, was den Agenten daran hindert, effektive Strategien zu erlernen [LeC+12; GBC16a].

Nullinitialisierung

Eine naheliegende, aber problematische Methode ist die Nullinitialisierung, bei der sämtliche Gewichte eines Netzes zu Beginn auf den Wert null gesetzt werden. Mathematisch lässt sich dies wie folgt formulieren:

$$W_{ij} = 0 \quad \forall i,j. \tag{2.27}$$

Dieser Ansatz führt jedoch dazu, dass alle Neuronen innerhalb einer Schicht identische Ausgaben berechnen und während des Trainings identische Gradienten erhalten. Das Netzwerk verliert dadurch seine Ausdrucksstärke, da keine Symmetrie zwischen den Neuronen gebrochen wird. Aus diesem Grund ist die Nullinitialisierung in der Praxis unbrauchbar und wird nur zu didaktischen Zwecken diskutiert.

Zufällige Initialisierung

Ein einfacher, historisch oft genutzter Ansatz ist die zufällige Initialisierung mit kleinen Werten, typischerweise aus einer Normal- oder Gleichverteilung:

$$W_{ij} \sim \mathcal{N}(0, \sigma^2) \quad \text{oder} \quad W_{ij} \sim U(-\epsilon, \epsilon). \tag{2.28}$$

Dabei ist σ eine kleine Standardabweichung und ϵ ein kleiner Schwellwert, der die Werte einschränkt. Diese Methode wurde vor allem in den frühen Phasen der Entwicklung neuronaler Netze eingesetzt, da sie einfach implementierbar ist und die Symmetrie zwischen den Neuronen bricht. In tiefen Netzwerken kann sie jedoch zu instabilen Gradienten führen, insbesondere wenn σ oder ϵ ungünstig gewählt sind. Im Kontext von Reinforcement Learning kann dies dazu führen, dass ein Agent nur langsam lernt oder in lokalen Optima gefangen bleibt[8].

Xavier-Initialisierung

Um die Probleme zufälliger Startwerte abzumildern, wurde die Xavier-Initialisierung vorgeschlagen [GB10b]. Ziel ist es, die Varianz der Eingaben und Ausgaben pro Schicht im Gleichgewicht zu halten. Die Gewichte werden dazu nach einer Gleichverteilung gewählt:

$$W_{ij} \sim U\left(-\sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}, \sqrt{\frac{6}{n_{\text{in}} + n_{\text{out}}}}\right), \tag{2.29}$$

wobei n_{in} und n_{out} die Anzahl der Eingänge und Ausgänge der jeweiligen Schicht darstellen. Alternativ kann auch eine Normalverteilung mit Varianz

$$\text{Var}(W_{ij}) = \frac{2}{n_{\text{in}} + n_{\text{out}}} \tag{2.30}$$

verwendet werden. Diese Strategie verhindert eine übermäßige Abschwächung oder Verstärkung von Signalen und hat sich besonders bei sigmoidalen und tanh-Aktivierungsfunktionen bewährt.

He-Initialisierung

Für Netzwerke mit ReLU-Aktivierungen wurde die He-Initialisierung eingeführt [He+15]. Sie basiert auf der Beobachtung, dass bei ReLU nur etwa die Hälfte der Neuronen aktiv ist, sodass eine höhere Varianz der Gewichte erforderlich ist, um eine konstante Signalstärke

[8] `https://machinelearningmastery.com/`, abgerufen am 23. September 2025.

zu gewährleisten. Formal ergibt sich:

$$W_{ij} \sim \mathcal{N}\left(0, \frac{2}{n_{\text{in}}}\right) \quad \text{oder} \quad W_{ij} \sim U\left(-\sqrt{\frac{6}{n_{\text{in}}}}, \sqrt{\frac{6}{n_{\text{in}}}}\right). \tag{2.31}$$

Diese Methode ist heute Standard in tiefen Architekturen, die ReLU oder Varianten davon verwenden, und trägt entscheidend zu stabilen Gradienten bei.

Anwendungen im Reinforcement Learning

Die Bedeutung der Gewichtsinitialisierung zeigt sich besonders deutlich in modernen Reinforcement-Learning-Algorithmen. In Deep-Q-Networks (DQN) [Mni+15] können ungeeignete Startgewichte dazu führen, dass die approximierten Q-Werte stark verzerrt sind, was die Exploration behindert und die Konvergenz verzögert.

2.4.9 Batch-Normalisierung

Batch-Normalisierung (Batch Normalization, BN) wurde von Ioffe und Szegedy eingeführt, um die Trainingsdynamik tiefer neuronaler Netze zu verbessern [IS15]. Die Grundidee besteht darin, die Aktivierungen innerhalb einer Schicht so zu normalisieren, dass sie über ein Mini-Batch hinweg einen stabilen Mittelwert und eine konstante Varianz aufweisen. Auf diese Weise wird verhindert, dass sich die Verteilungen der Zwischenschichten während des Trainings zu stark verschieben, ein Phänomen, das als *Internal Covariate Shift* bezeichnet wird. Dies führt zu einer schnelleren Konvergenz und einer robusteren Optimierung.

Mittelwert und Varianz

Gegeben sei ein Mini-Batch $B = \{x_1, x_2, \ldots, x_m\}$ von Eingaben einer Schicht mit Batchgröße m. Zunächst wird der Mittelwert des Batches berechnet:

$$\mu_B = \frac{1}{m} \sum_{i=1}^{m} x_i. \tag{2.32}$$

Anschließend wird die Varianz innerhalb des Batches bestimmt:

$$\sigma_B^2 = \frac{1}{m} \sum_{i=1}^{m} \left(x_i - \mu_B\right)^2. \tag{2.33}$$

Hierbei bezeichnet x_i die Aktivierung der i-ten Instanz im Batch, μ_B den Mittelwert und σ_B^2 die Varianz.

Normalisierung der Aktivierungen

Die einzelnen Aktivierungen werden nun mit Hilfe des Mittelwerts und der Varianz normalisiert:

$$\hat{x}_i = \frac{x_i - \mu_B}{\sqrt{\sigma_B^2 + \epsilon}}, \tag{2.34}$$

wobei ϵ eine kleine Konstante ist, die numerische Stabilität gewährleistet und eine Division durch Null verhindert. Das Ergebnis $\hat{x}_i$ ist eine normalisierte Aktivierung mit Mittelwert null und Varianz eins innerhalb des Mini-Batches.

Skalierung und Verschiebung

Um die Ausdrucksstärke des Netzes nicht durch die Normalisierung einzuschränken, werden die normalisierten Werte zusätzlich durch zwei lernbare Parameter transformiert:

$$y_i = \gamma \hat{x}_i + \beta. \tag{2.35}$$

Hierbei ist γ ein Skalierungsparameter und β ein Verschiebungsparameter, die beide während des Trainings optimiert werden. Auf diese Weise kann das Netzwerk die Normalisierung im Extremfall sogar rückgängig machen, falls dies für die jeweilige Aufgabe von Vorteil ist. Empirisch zeigt sich jedoch, dass die meisten Modelle von der stabilisierten Verteilung profitieren und dadurch schneller sowie robuster trainieren.

Bedeutung für Reinforcement Learning

Im Bereich des Reinforcement Learning spielt Batch-Normalisierung eine besondere Rolle, da die Verteilungen von Zuständen und Belohnungen dynamisch sind und sich im Laufe des Trainings verändern. Ohne Normalisierung können neuronale Netze unter diesen Bedingungen stark schwankende Gradienten aufweisen, was den Lernprozess instabil macht. Durch die Anwendung von Batch-Normalisierung werden die Aktivierungen stabilisiert, was insbesondere in tiefen Netzwerken mit Q-Funktion-Approximationen oder in Policy-Gradient-Methoden zu einer signifikant verbesserten Konvergenz führt [IS15; San+18].

2.5 Grundlagen des Reinforcement Learning

Reinforcement Learning (RL), auf Deutsch bestärkendes Lernen, ist ein Teilgebiet des maschinellen Lernens, das sich deutlich von den etablierten Paradigmen des überwachten und unüberwachten Lernens unterscheidet. Während beim überwachten Lernen ein Modell auf Basis von annotierten Daten trainiert wird, deren Eingaben und Zielwerte bekannt sind, und beim unüberwachten Lernen Strukturen oder Cluster in unbeschrifteten Daten identifiziert werden, basiert Reinforcement Learning auf einem anderen Prinzip. Hier steht

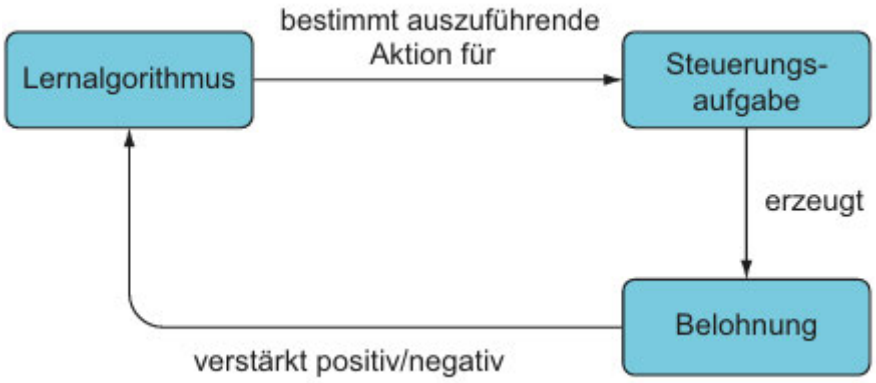


Abbildung 2.10: RL-Framework des Reinforcement Learning [BZ20]

die Interaktion eines Agenten mit einer Umgebung im Vordergrund, bei der der Agent eigenständig durch Versuch und Irrtum eine Strategie, die sogenannte *Policy*, erlernt. Diese Policy soll so optimiert werden, dass die erwartete kumulierte Belohnung über die Zeit maximiert wird [BZ20; SB18].
In einem RL-Framework, wie in Abbildung 2.10, wählt der Lernalgorithmus die Aktionen aus, die zur Bewältigung einer Steuerungsaufgabe notwendig sind. Jede ausgeführte Aktion wird mit einer Belohnung oder Bestrafung bewertet, wodurch der Algorithmus lernt, welche Handlungen vorteilhaft sind und welche vermieden werden sollten.

Ein RL-System wird üblicherweise in Form eines Markov-Entscheidungsprozesses (MDP) modelliert. Ein MDP beschreibt die dynamische Interaktion zwischen Agent und Umgebung durch eine Menge von Zuständen S, eine Menge von Aktionen A, eine Übergangsfunktion $P(s'|s,a)$, die die Wahrscheinlichkeit des Übergangs vom Zustand s in den Folgezustand s' unter Ausführung der Aktion a angibt, eine Belohnungsfunktion $R(s,a)$, die das Feedback der Umgebung repräsentiert, sowie einen Discountfaktor $\gamma \in [0,1]$, der zukünftige Belohnungen abwertet. Die Markov-Eigenschaft impliziert dabei, dass die Zukunft nur vom aktuellen Zustand und der gewählten Aktion abhängt und nicht von der gesamten Historie der Zustandsfolge.

Der Lernprozess lässt sich als zyklische Interaktion beschreiben. Der Agent befindet sich zu Beginn in einem Zustand s_t und wählt gemäß seiner Policy $\pi(a|s_t)$ eine Aktion[9] a_t. Die Umgebung reagiert auf diese Aktion, indem sie in einen neuen Zustand s_{t+1} übergeht und dem Agenten eine Belohnung R_{t+1} zurückmeldet. Diese Rückmeldungen sind für den Agenten die einzige Informationsquelle darüber, wie gut oder schlecht seine Entscheidungen im Hinblick auf das langfristige Ziel waren. Ziel ist es nicht, in jedem einzelnen Schritt die maximale Belohnung zu erzielen, sondern über eine Sequenz von Schritten hinweg den

[9] `https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf?utm_source=chatgpt.com`, abgerufen am 30. September 2025.

kumulierten Reward zu maximieren. Dieser sogenannte Return G_t wird formal definiert als

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \tag{2.36}$$

wobei γ als Discountfaktor sicherstellt, dass unmittelbare Belohnungen stärker gewichtet werden als weiter in der Zukunft liegende.
Die Qualität einer Policy wird durch Wertfunktionen (*Value Functions*) beschrieben. Die Zustandswertfunktion $V^\pi(s)$ gibt an, wie hoch der erwartete Return ist, wenn der Agent im Zustand s startet und anschließend der Policy π folgt:

$$V^\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]. \tag{2.37}$$

Diese Funktion erlaubt es einzuschätzen, welche Zustände langfristig vorteilhaft sind[10]. Noch detaillierter ist die Aktionswertfunktion $Q^\pi(s,a)$, die zusätzlich die Auswahl einer bestimmten Aktion berücksichtigt:

$$Q^\pi(s,a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]. \tag{2.38}$$

Während $V^\pi(s)$ die Güte eines Zustands bewertet, erlaubt $Q^\pi(s,a)$ eine direkte Einschätzung, welche Handlung in einem Zustand vorteilhaft ist. Die optimalen Wertfunktionen $V^*(s)$ und $Q^*(s,a)$ charakterisieren die optimale Policy π^*, die den maximalen erwarteten Return für alle Zustände garantiert.

Der Kern des Reinforcement Learning besteht in der Balance zwischen Exploration und Exploitation. Einerseits muss der Agent die Umgebung ausreichend erkunden, um herauszufinden, welche Zustände und Aktionen vorteilhaft sind. Andererseits muss er das bereits Gelernte ausnutzen, um hohe Rewards zu erzielen.
Der im Ablaufdiagramm dargestellte Zyklus der Abbildung 2.11, beginnt mit der Wahrnehmung des aktuellen Zustands durch den Agenten. Auf Grundlage dieses Zustands wählt er eine Aktion, die seiner aktuellen Policy entspricht. Die Umgebung verarbeitet diese Aktion, verändert ihren Zustand und gibt sowohl den neuen Zustand als auch eine Belohnung an den Agenten zurück. Diese Information nutzt der Agent, um seine Policy oder seine Wertfunktionen zu aktualisieren, sodass künftige Entscheidungen besser auf die Maximierung des langfristigen Returns ausgerichtet sind. Dieser Prozess wiederholt sich fortlaufend und bildet das Fundament des Lernens im RL. Über viele Episoden hinweg verfeinert der Agent seine Policy und nähert sich schrittweise einer optimalen Strategie an.

[10] `https://incompleteideas.net/book/first/ebook/node34.html?utm_source=chatgpt.com`, abgerufen am 30. September 2025.

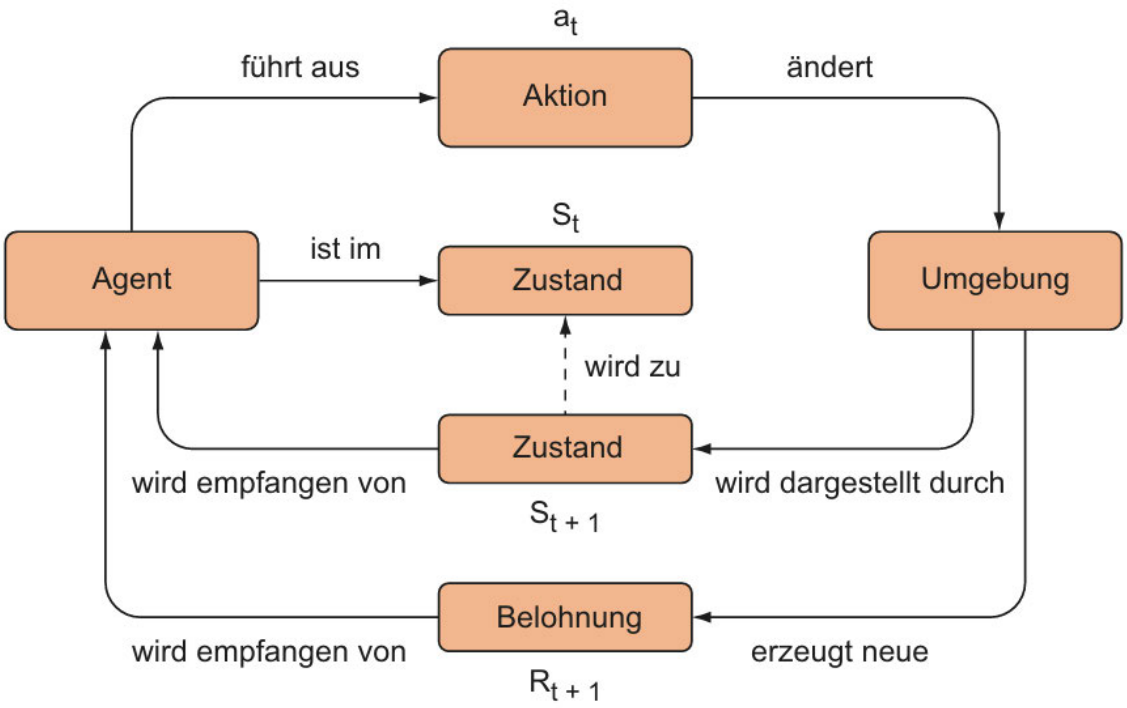


Abbildung 2.11: Standardmodell des Reinforcement Learning [BZ20]

Häufige Probleme beim Trainieren von neuronalen Netzen im Reinforcement Learning

Das Training von neuronalen Netzen im Reinforcement Learning (RL) ist mit einer Reihe spezifischer Herausforderungen verbunden, die über die bekannten Schwierigkeiten des überwachten Lernens hinausgehen. Diese Probleme entstehen insbesondere durch die enge Verzahnung von Datenverteilung, Agentenpolitik und Umweltinteraktion, was zu Instabilitäten und ineffizientem Lernen führen kann [Mni+15; Hen+18].

Instabilität durch Korrelationen

Ein weiteres Problem entsteht durch starke Korrelationen zwischen aufeinanderfolgenden Beobachtungen. Da Agenten typischerweise Zustandsfolgen durch Interaktion mit der Umwelt erzeugen, sind die Datenpunkte zeitlich abhängig. Werden diese direkt zum Training eines neuronalen Netzes verwendet, kann dies zu verzerrten Gradienten und unstabilen Lernprozessen führen. In Methoden wie dem Deep Q-Network (DQN) wurde deshalb Experience Replay eingeführt, um Korrelationen zu reduzieren und die Datenverteilung zu glätten [Mni+15].

Exploration vs. Exploitation

Ein fundamentales Dilemma im RL besteht in der Balance zwischen Exploration und Exploitation. Neuronale Netze tendieren dazu, frühe Muster in den Daten stark zu verstärken,

was dazu führen kann, dass der Agent bestimmte Aktionsbereiche übermäßig ausnutzt und andere kaum erkundet. Insbesondere bei komplexen Umgebungen führt eine unzureichende Exploration zu suboptimalen Politiken, die durch bloßes weiteres Training nur schwer überwunden werden können.

Sparse Rewards

In vielen RL-Umgebungen sind Belohnungssignale spärlich oder treten erst nach einer langen Sequenz von Handlungen auf. Neuronale Netze, die auf solche Signale angewiesen sind, haben Schwierigkeiten, nützliche Gradienten zu erhalten, da der Zusammenhang zwischen Aktionen und Belohnungen schwer zu erfassen ist. Dies führt oft zu ineffizientem Training und erfordert spezielle Techniken wie Reward Shaping oder Hierarchical Reinforcement Learning.

Verschwindende und explodierende Gradienten

Wie in anderen tiefen Architekturen können auch im RL verschwindende und explodierende Gradienten auftreten. Diese Probleme entstehen insbesondere in tiefen Netzwerken oder rekurrenten Architekturen, die in RL häufig für sequentielle Entscheidungsprozesse eingesetzt werden. Während verschwindende Gradienten dazu führen, dass frühere Zustandsinformationen nicht effektiv genutzt werden können, resultieren explodierende Gradienten in numerischer Instabilität und stark schwankenden Updates [PMB13].

Hyperparameter-Sensitivität

Schließlich ist das Training von RL-Agenten mit neuronalen Netzen oft stark von den gewählten Hyperparametern abhängig. Lernrate, Discount-Faktor, Netzwerktopologie und Batchgröße beeinflussen die Stabilität und die Geschwindigkeit der Konvergenz erheblich. Im Gegensatz zu überwachten Lernaufgaben ist es im RL besonders schwierig, robuste Kombinationen von Hyperparametern zu finden, die in unterschiedlichen Umgebungen zuverlässig funktionieren [Hen+18].

2.6 Q-Learning

Eine weit verbreitete Methode zum Erlernen der optimalen Aktionswertfunktion ist das Q-Learning. Dabei wird die Q-Funktion iterativ mithilfe der Bellman-Gleichung aktualisiert. Das Verfahren benötigt keine Modellierung der Übergangswahrscheinlichkeiten der Umgebung, sondern basiert ausschließlich auf den erlebten Zustands-Transitions- und Belohnungssignalen. Die Aktualisierungsregel lautet:

$$Q(s_t,a_t) \leftarrow Q(s_t,a_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(s_{t+1},a) - Q(s_t,a_t)\Big], \tag{2.39}$$

wobei α die Lernrate ist[11].
Das klassische Q-Learning speichert alle Q-Werte in einer Tabelle. Dies funktioniert gut für kleine, diskrete Zustands- und Aktionsräume, stößt jedoch bei hochdimensionalen oder kontinuierlichen Umgebungen schnell an seine Grenzen. Hier setzt Deep Q-Learning an, das neuronale Netze zur Approximation der Q-Funktion nutzt.

2.7 Deep Q-Learning

Um die Einschränkungen von Q-Learning zu überwinden, wurde Deep Q-Learning (DQL) entwickelt. Anstelle einer Q-Tabelle wird ein tiefes neuronales Netz eingesetzt, das die Q-Funktion approximiert. Dieses Netz erhält den aktuellen Zustand als Eingabe und liefert für jede mögliche Aktion einen approximierten Q-Wert. Dadurch lassen sich auch hochdimensionale oder kontinuierliche Zustandsräume handhaben, die mit klassischem Q-Learning nicht mehr effizient lösbar wären.
Zudem werden in DQL Techniken wie *Experience Replay* (Wiederverwendung von vergangenen Erfahrungen) und *Target Networks* (separates Zielnetz zur Stabilisierung) eingesetzt, um das Lernen robuster und stabiler zu machen.

2.8 Vergleich zwischen Q-Learning und Deep Q-Learning

Der Unterschied zwischen den beiden Ansätzen liegt primär in der Repräsentation der Q-Funktion:

- **Q-Learning:** Speicherung der Q-Werte in einer Tabelle. Funktioniert gut bei kleinen, diskreten Zustands- und Aktionsräumen.
- **Deep Q-Learning:** Approximation der Q-Werte durch ein neuronales Netz. Eignet sich für komplexe, hochdimensionale Probleme und ermöglicht die Anwendung von Reinforcement Learning in realistischen Szenarien, etwa bei Spielen (z. B. Atari) oder der Robotik.

Abbildung 2.12 zeigt einen vereinfachten Vergleich zwischen dem Q-Learning und dem Deep Q-Learning

[11] `https://incompleteideas.net/book/the-book-2nd.html`, abgerufen am 30. September 2025.

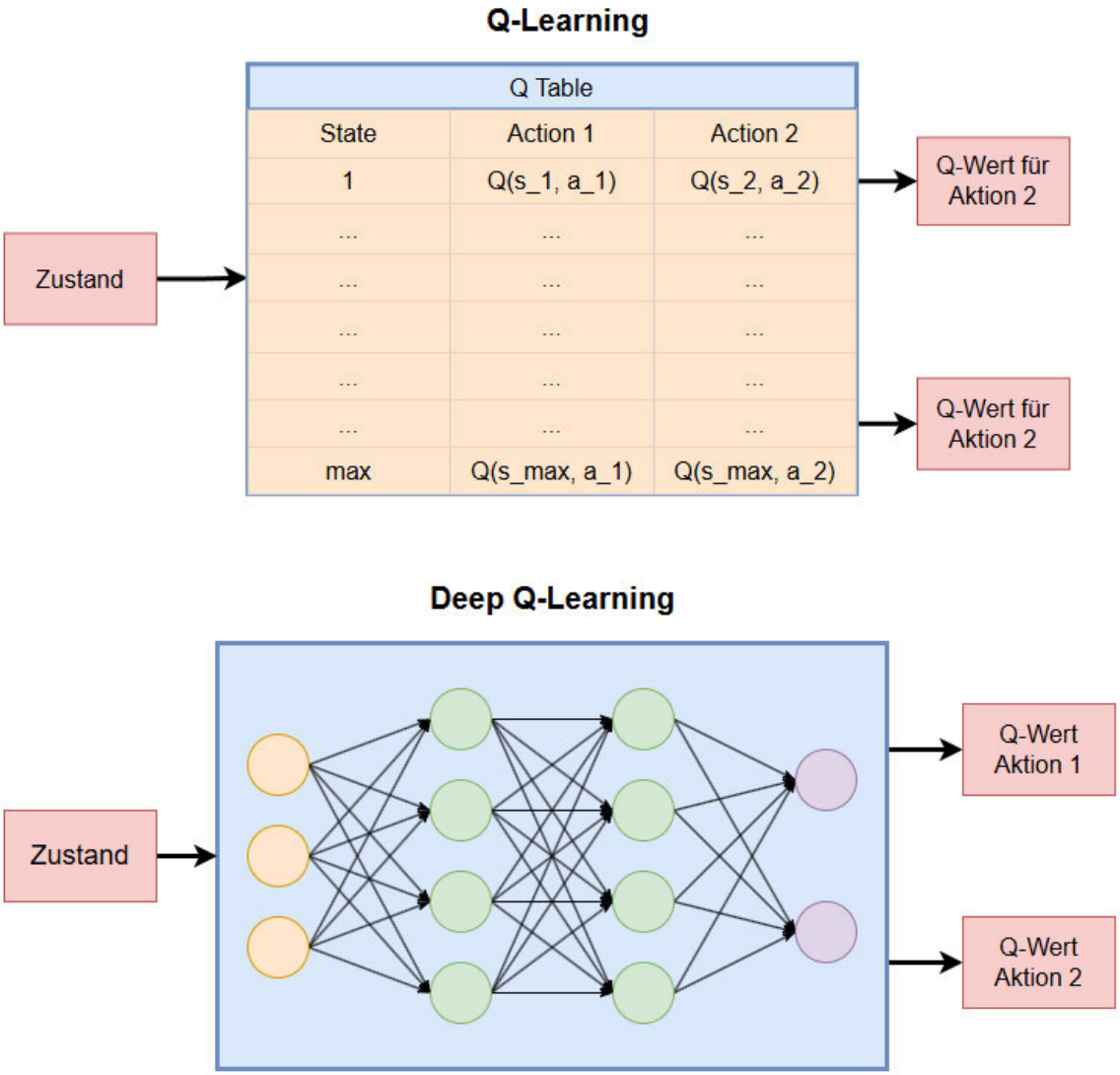


Abbildung 2.12: Schematischer Vergleich zwischen Q-Learning und Deep Q-Learning

3 Stand der Technik

Die Entwicklung von Flipperautomaten und deren digitale Adaptionen bieten einen spannenden Überblick über den technologischen Fortschritt von rein mechanischen Systemen hin zu hybriden, KI-gestützten Spielumgebungen. Für die Einordnung aktueller Forschungsarbeiten im Bereich der KI-gesteuerten Flipperautomaten ist es daher sinnvoll, sowohl die historische Entwicklung physischer Geräte als auch den Aufstieg digitaler Simulationen zu betrachten. Im Folgenden werden zentrale Meilensteine und bestehende Ansätze vorgestellt, die den Stand der Technik in diesem Bereich widerspiegeln.

3.1 Physische Flipperautomaten

Erste elektromechanische Modelle entstanden in den 1930er Jahren, während ab den 1970er Jahren elektronische Steuerungssysteme in Serienproduktionen Einzug hielten [Ken01]. Moderne Geräte vereinen heute mechanische, elektrische und digitale Technologien in einem hybriden Spielsystem.

Elektronische Steuerung und Sensorik

Seit den 1980er Jahren verwenden Flipperautomaten mikrokontrollerbasierte Plattformen zur Spiellogik, Steuerung von Sound und Lichteffekten, sowie zur Verarbeitung von Eingabesignalen wie Schaltern oder optischen Sensoren [Sha13]. Kommerzielle Systeme wie Williams' WPC-Plattform oder Sterns Spike-System gelten als Meilensteine in der modularen Flippertechnologie[12].

3.2 Digitale Flipperautomaten

Neben mechanisch-physischen Flipperautomaten haben sich in den letzten Jahrzehnten zunehmend digitale Flippersimulationen und virtuelle Flipper-Systeme etabliert. Diese Systeme bilden die Mechanik und Spiellogik klassischer Flipperautomaten softwarebasiert ab und bieten eine wichtige Plattform für Spielentwicklung, Emulation und Forschung, sowie den Einsatz künstlicher Intelligenz.
Ein bekanntes Open-Source-Projekt in diesem Bereich ist **Visual Pinball X (VPX)**. Dies ist eine auf Windows basierende Simulationsumgebung, die realistische physikalische Modelle von Kugelbewegungen, Kollisionen und Spielflächenverhalten ermöglicht[13]. Die Umgebung wird häufig in Kombination mit **VPinMAME** verwendet. Hierbei handelt es sich um einen Emulator, welcher eine originalgetreue Nachbildung historischer Titel ermöglicht. Die Community stellt zahlreiche, teils lizenzierte und teils fiktive Spieltische bereit.

[12] `https://www.ipdb.org/`, abgerufen am 30. September 2025.
[13] `https://www.vpforums.org/`, abgerufen am 30. September 2025.

Kommerzielle digitale Flipperspiele wie **Pinball FX** von Zen Studios bieten darüber hinaus hochwertige 3D-Grafiken, dynamische Spielphysik und Online-Funktionen wie Bestenlisten oder Multiplayer-Modi [Stu22]. Diese Titel richten sich sowohl an Casual Gamer als auch an Flipper-Enthusiasten. Verfügbar sind diese auf verschiedenen Plattformen (PC, Konsole, Mobile).

3.3 Erste Schritte der KI gesteuerten Flipperautomaten

3.3.1 Visuelle Systeme und Automatisierung

In klassischen Flipperautomaten sind keine computergestützten visuellen Erfassungssysteme verbaut. Einige experimentelle Forschungsarbeiten und Maker-Projekte haben jedoch Bildverarbeitungstechnologien wie Kameras und maschinelles Sehen eingesetzt, um die Kugelbewegung in Echtzeit zu analysieren oder Flipperautomaten teilautomatisiert zu steuern [Mir+16; BZB20]. Diese Konzepte befinden sich nur im Prototyp-Stadium und sind nicht auf kommerziellen Geräten verbreitet.

3.3.2 PinBot – Reinforcement Learning mit einem digitalen Flipperautomaten

Ein aktuelles Beispiel stellt das Projekt „PinBot – Reinforcement Learning on a Pinball Machine[14]“ dar, welches im Rahmen eines Kurses an der Carnegie Mellon University (CMU) realisiert wurde. Ziel des Projekts war die Entwicklung eines RL-Agenten, welcher das Spiel Total Nuclear Annihilation (TNA) eigenständig erlernen und spielen kann. Hierbei wurde zunächst ein Simulationsmodell des Spiels in Unity basierend auf der Plattform Visual Pinball X erstellt. Der Agent wurde mithilfe des Proximal Policy Optimization (PPO)-Algorithmus trainiert. Die Beobachtungen umfassten ein qualitatives reduziertes Bild des Spielfelds sowie Positions- und Geschwindigkeitsdaten der Kugel. Die Aktionsauswahl bestand aus diskreten Befehlen zur Steuerung der Flipperarme. Die Definition des Belohnungssystems sah eine Belohnung mit hohen Spielpunkten für aktives Spielverhalten und eine starke negative Gewichtung für den Verlust der Kugel vor.

Das Training erfolgte über rund 90.000 Schritte. Tabelle 3.2 zeigt, dass der trainierte Agent eine zufällige Steuerung signifikant übertraf und erreichte ein nahezu vergleichbares Spielniveau wie menschliche Spieler. Die Architektur des Modells kombinierte ein kleines Convolutional Neural Network (CNN) mit einem Recurrent Neural Network (RNN), um dem Agenten eine temporale Gedächtnisfähigkeit über etwa 1,5 Sekunden zu geben. Hierdurch wurde ein entscheidender Aspekt für die dynamische Kontrolle im Spiel geschaffen.

Ein besonderes Merkmal des Projekts ist der Versuch der Transferierung des in der Simulation trainierten Modells auf eine reale physische Flippermaschine. Hierdurch soll die

[14] `https://sahiltchaudhary.github.io/portfolio/2_pinbot/`, abgerufen am 30. September 2025.

	Menschlicher Spieler	**Trainierter Agent**	**Untrainierter Agent**	**Kein Agent**
Punktzahl Mittelwert	38.952,86	33.952,00	14.420,00	8.358,00
Punktzahl Std.-Abw.	26.546,47	35.787,92	9.981,91	376,43
Zeit Mittelwert (s)	69,29	57,46	37,68	36,00
Zeit Std.-Abw.	15,18	25,77	7,88	11,81

Tabelle 3.2: Vergleich der Ergebnisse zwischen menschlichem Spieler, trainiertem Agenten, untrainiertem Agenten und keiner Agentenunterstützung.

Trainingszeit drastisch reduziert und das Modell auf echte Hardware angepasst werden. Diese sogenannte Sim2Real-Übertragung stellt derzeit eine der größten Herausforderungen im Bereich physisch interagierender RL-Systeme dar.
Das Projekt demonstriert eindrucksvoll das Potenzial von RL im Bereich physischer Spiele mit hoher Reaktionsdynamik. Gleichzeitig verdeutlicht es die bestehenden Herausforderungen bei der Übertragung von Lernmodellen aus simulierten Umgebungen in reale Systeme. Hierbei liegt ein besonderer Fokus auf der Sensorlatenz, der Robustheit der visuellen Wahrnehmung und der präzisen Aktorensteuerung.

3.3.3 Laufendes Forschungsprojekt: KI-gesteuerter Flipperautomat an der Helmut-Schmidt-Universität

Im Rahmen eines aktuellen Forschungsprojekts an der Helmut-Schmidt-Universität Hamburg wird die Automatisierung eines elektromechanischen Flipperautomaten mithilfe von Reinforcement Learning (RL) untersucht[15]. Ziel des Projekts ist es, verschiedene Strategien zur Steuerung der Flipperarme zu entwickeln und zu evaluieren – insbesondere im Hinblick auf eine Optimierung der Spieldauer und des Gewinnens von Punkten.
Im Zentrum des Projekts steht der Einsatz von Reinforcement Learning zur autonomen Entscheidungsfindung. Eine besondere Herausforderung ergibt sich dabei aus dem hohen Maß an mechanischer Toleranz und Variabilität im dynamischen Verhalten des historischen, elektromechanischen Flipperautomaten. Diese Systemungenauigkeiten stellen hohe Anforderungen an die Präzision und Robustheit der KI-gesteuerten Regelung.
Zur Erfassung der Kugelbewegung in Echtzeit wird ein optisches Motion-Capture-System der Firma VICON eingesetzt. Das System nutzt mehrere Infrarotkameras, die reflektierende Marker auf der Kugel gleichzeitig erfassen. Über Triangulation kann so die dreidimensionale Position der Kugel im Raum mit hoher Genauigkeit berechnet werden. Die gewonnenen Positionsdaten dienen als Eingangsinformationen für die RL-Agenten und werden durch geeignete Algorithmen in Steuerbefehle für die Flippermechanik übersetzt.

[15] `https://www.hsu-hh.de/rt/forschung/gottlieb_300`, abgerufen am 15. April 2021.

3.4 Vorhandener Fortschritt

Alexander Wessel und Prof. Dr. Marc Hensel haben zusammen mit der Bachelorarbeit „Entwicklung und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Flipper-Automaten“ große Vorarbeit für den KI gesteuerten Flipper geleistet. Prof. Dr. Marc Hensel hat den Aufbau der Bachelorarbeit nachgebaut und für den Erfolg dieser Masterarbeit zur Verfügung gestellt. Dadurch gibt es bereits bis auf die Kamera den gesamten Hardwareanteil. Zum Start dieser Masterarbeit wurde somit der physische Demonstrator mit der benötigten Software für die Motorsteuerung zu Verfügung gestellt.
Der im Rahmen der Arbeit entwickelte prototypische Flipper orientiert sich an klassischen Automaten. Grundlage ist eine Aluverbundplatte als Spielfeld, eingefasst in einen stabilen Rahmen aus Aluminiumprofilen [Wes24].

Spielfeld
Das Spielfeld besitzt eine Abmessung von 50 cm × 34 cm und ist in eine Gesamtplatte von 54 cm × 38 cm eingefasst. Um einen natürlichen Kugellauf durch die Schwerkraft zu gewährleisten, ist es um 7° geneigt.

Flipperarme
Die beiden Flipperarme sind jeweils 4,5 cm lang und in der Ausgangsstellung 35° zur Waagerechten geneigt. Der Bewegungsbereich beträgt 70° und wird durch Schrittmotoren realisiert. Die Flipperarme wurden als rote 3D-gedruckte Bauteile gefertigt und auf die Motorachsen montiert.

Kugel
Als Spielkugel wurde eine Kugel mit ca. 13,5 mm Durchmesser und 80 g Gewicht eingesetzt.

Rahmen und Basis
Der Rahmen besteht aus Aluminiumprofilen, verschraubt mit Innenwinkeln und Nutsteinen. Das Spielfeld selbst besteht aus einer 3 mm starken Aluverbundplatte. Der modulare Aufbau ermöglicht Wiederverwendbarkeit und einfache Erweiterung.

Antrieb und Steuerung
Als Antrieb wurden NEMA 17 Schrittmotoren (Modell `17HS19-2004S1`) eingesetzt, welche über ein Arduino Uno angesteuert werden. Diese erlauben präzise Teildrehungen und decken den geforderten Bewegungsbereich von 70° zuverlässig ab. Alternative Motoren mit geringerem Drehmoment erwiesen sich als ungeeignet.

3D-gedruckte Elemente
Zu den selbst gefertigten 3D-Bauteilen zählen:

- Flipper (2 Stück)
- Banden (links/rechts)
- Startflipper
- Starttunnel und Rückführungsrampe (Längen: 2×110 mm, 1×90 mm)
- Motorhalterungen und Stützen
- Einstellbare Füße für die Neigung

Übersicht der Maße

Bauteil	**Maß**	**Einheit**
Spielfeld	50×34	cm
Gesamtplatte	54×38	cm
Spielfeldneigung	7	°
Flipperarm Länge	4,5	cm
Flipper Bewegungsbereich	70	°
Kugel	12,7 (Ø), 9	mm, g
Plattenstärke	3	mm
Start- und Rückführungsrampe	110, 90	mm
Motor (NEMA 17)	Schrittweite 1,8	°/Schritt

Tabelle 3.3: Wichtige Bauteile des Flipper-Demonstrators mit Maßen

Damit wird ein maßstabsreduzierter, aber funktionaler Prototyp bereitgestellt, der sich durch Modularität, Wiederverwendbarkeit und präzise Steuerung auszeichnet. Tabelle 3.3 zählt die vorhandenen Bauteil mit Maßen auf.

Vorhandener Fortschritt - Virtuelle Umgebung
In der von Prof. Dr. Marc Hensel weiterentwickelten Vorarbeit wurde eine dreidimensionale Rendering-Umgebung für einen Flipperautomaten entwickelt, die als Grundlage für eine OpenAI-Gym-Umgebung dient. Die Implementierung, die in Abbildung 3.1 gezeigt wird, basiert auf der Klasse `PinballRender3D`, die die Geometrie des Spielfelds aus der Klasse `PinballGeometry` übernimmt. Bereits enthalten sind die wesentlichen statischen Bestandteile des Flipperautomaten: das Spielfeld mit Umrandung, der Startkanal

Abbildung 3.1: Bereits vorhandene Simlulation

(Launcher-Lane), die seitlichen und oberen Begrenzungen sowie die abgerundeten Ecken im oberen Bereich.

4 Anforderungsanalyse

Die Anforderungsanalyse ist ein zentraler Bestandteil dieser Masterarbeit, insbesondere bei technisch und wissenschaftlich anspruchsvollen Projekten wie einem KI-gesteuerten Flipperautomaten. Sie beschreibt den systematischen Prozess, der die Anforderungen an das Projekt identifiziert, dokumentiert und priorisiert. Ziel ist die Schaffung einer klaren Grundlage für die Planung, Umsetzung und Evaluation dieser Arbeit.
Bezogen auf einen KI-Flipperautomaten ergibt sich somit die Anforderungsnalyse in Form der Ermittlung aller funktionalem und nicht-funktionalem Anforderungen, die für die Entwicklung eines autonomen Systems notwendig sind. Das autonome System arbeitet hierbei mit einer Kamera, um den Spielverlauf zu analysieren und die Flipperarme in Echtzeit zu steuern.

4.1 Systemumgebung

Die Systemumgebung des KI-gesteuerten Flipperautomaten setzt sich aus mehreren, eng miteinander verknüpften Komponenten zusammen, welche sowohl den physischen Betrieb als auch den KI-Trainingsprozess unterstützen.
Im Zentrum steht der Flipperautomat, welcher durch präzise gesteuerte Motoren betrieben wird. Diese Motoransteuerung übernimmt die mechanischen Aktionen, welche für ein authentisches Spielgeschehen notwendig sind, beispielsweise das Auslösen der Flipperarme.
Ein wesentlicher Bestandteil des Systems ist die integrierte Kamera. Diese liefert präzise Echtzeitbilder, anhand derer die KI den aktuellen Status des Automaten erfasst. Hierbei werden relevante Parameter, wie die Position der Flipperarme, die Rollbahn der Kugel und andere kritische Systemzustände ermittelt. Diese visuellen Informationen fließen direkt in den Entscheidungsprozess der KI ein und ermöglichen eine adaptive Steuerung, welche in Echtzeit auf Veränderungen im Spielgeschehen reagiert.
Zur Beschleunigung des Lernprozesses der KI, wird ein digitaler Zwilling eingesetzt. Dieser bildet den Flipperautomaten virtuell ab und simuliert realitätsnahe physikalische Dynamiken, insbesondere in der Anfangsphase des Trainings. Durch diese Simulation können große Datenmengen schnell generiert und analysiert werden, sodass die KI bereits in einer kontrollierten Umgebung grundlegende Strategien und Reaktionsmuster erlernen kann, bevor eine Übertragung auf die reale Hardware vorgenommen wird.
Abbildung 4.1 zeigt ein vereinfachtes Schema der Systemumgebung.

4.2 Stakeholder des Projekts

Der Erfolg dieses Projekts hängt nicht nur von der technischen Umsetzung und der Leistungsfähigkeit der KI ab, sondern auch von den Bedürfnissen und Erwartungen verschiedener Stakeholder. Diese Gruppen und Einzelpersonen beeinflussen das Projekt direkt

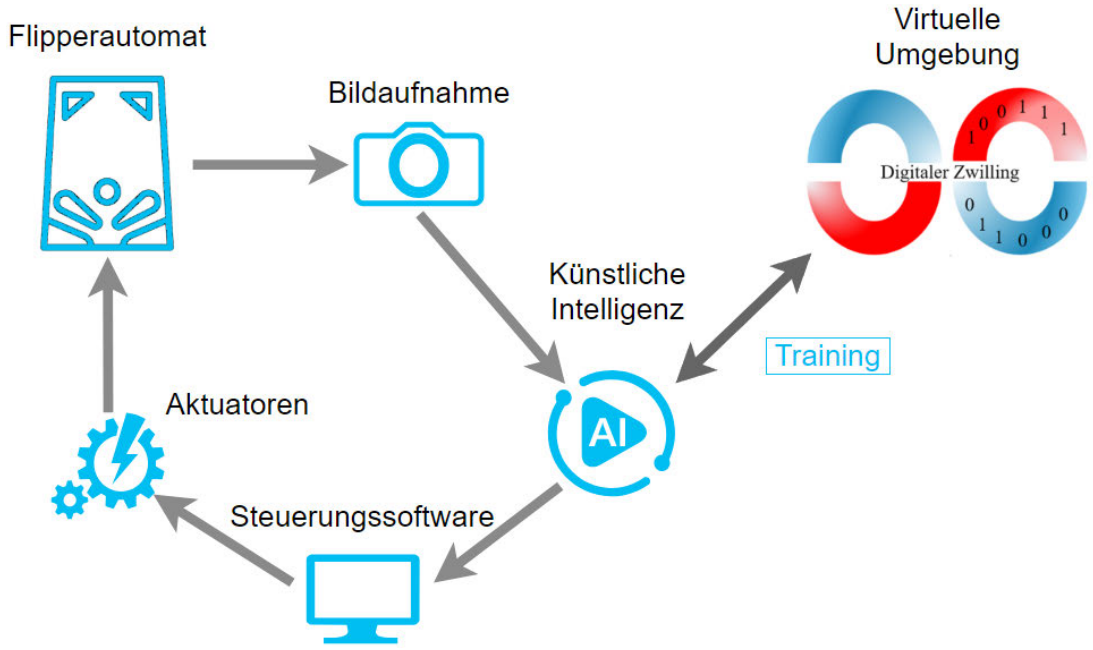


Abbildung 4.1: Vereinfachtes Schema der Systemumgebung

oder indirekt und haben ein Interesse an seinem Gelingen.
Im Folgenden werden die wichtigsten Stakeholder beschrieben.

Betreuer und Hochschule

Die betreuenden Professor:innen und die Hochschule sind zentrale Stakeholder, da sie die wissenschaftlichen Anforderungen an die Masterarbeit definieren. Sie erwarten eine methodisch fundierte, innovative und nachvollziehbare Umsetzung des Projekts sowie eine klare Dokumentation der Ergebnisse. Herr Prof. Dr. Hensel ist als Erstprüfer und Betreuer dieser Masterarbeit ein zentraler Stakeholder und übernimmt gleichzeitig die Rolle des Auftraggebers. Dabei verfolgt er verschiedene Ziele und Interessen.
Ein zentrales Anliegen ist die Entwicklung eines funktionsfähigen Flipperautomaten, welcher durch künstliche Intelligenz gesteuert wird. Dieses Projekt soll moderne KI-Technologien nicht nur theoretisch erforschen, sondern auch deren praktische Umsetzung aufzeigen. Der Fokus liegt auf dem Aufzeigen der Einsatzmöglichkeiten künstlicher Intelligenz zur Analyse von Spielmustern, zur Optimierung von Reaktionszeiten und zur autonomen Steuerung in Echtzeit.
Darüber hinaus dient der KI-gesteuerte Flipperautomat als Anschauungs- und Demonstrationsobjekt, um die Möglichkeiten intelligenter Systeme in interaktiven mechanischen Umgebungen zu veranschaulichen. Dies kann sowohl in wissenschaftlichen als auch in industriellen Kontexten von Interesse sein. Zudem bietet das Projekt eine Grundlage für zukünftige studentische Arbeiten, in denen unterschiedliche KI-Verfahren, Bildverarbeitungsmethoden und Ansätze für Softwareentwicklung weiterentwickelt und für ähnliche

Anwendungen optimiert werden können.
Für den Prüfer stehen neben der technischen Umsetzung, insbesondere der wissenschaftliche Erkenntnisgewinn und die Erweiterung des Fachwissens im Vordergrund. Dabei spielen sowohl theoretische Aspekte, wie die Funktionsweise und Entscheidungsmechanismen künstlicher Intelligenz, als auch praktische Erfahrungen mit der Implementierung und Optimierung von KI-gesteuerten Systemen eine wesentliche Rolle.

Entwickler:innen und Forscher:innen in den Bereichen KI und Robotik
Dieses Projekt bietet wertvolle Erkenntnisse für Entwickler:innen und Forscher:innen in den Bereichen künstliche Intelligenz, maschinelles Lernen und Robotik. Die Steuerung eines Flipperautomaten durch KI erfordert eine Kombination aus Echtzeit-Datenverarbeitung, Sensortechnologie und adaptiven Algorithmen. Hierdurch werden neue Ansätze für die Automatisierung mechanischer Systeme aufgezeigt.
Besonders interessant ist die Anwendung von Reinforcement Learning zur Entwicklung optimaler Spielstrategien. Hierbei wird zum Einen die Anpassungsfähigkeit/-fertigkeit der KI in einer physikalischen Umgebung untersucht; zum Anderen wird analysiert, welche hybriden Ansätze - regelbasierte Algorithmen kombiniert mit datengetriebenem Lernen – zu effizienteren Steuerungen führen.
Darüber hinaus liefert das Projekt Impulse für die Spielentwicklung, insbesondere im Bereich intelligenter Spielmechaniken und adaptiver Schwierigkeitsgrade. Die Erkenntnisse könnten nicht nur für Arcade-Spiele, sondern auch für interaktive Robotik und maschinelle Entscheidungsfindung von Bedeutung sein.

Studierende
Durch die Masterarbeit können Studierende wertvolle Einblicke in moderne Algorithmen und praxisnahe Anwendungen erhalten. Gleichzeitig profitieren sie von der Möglichkeit, die Forschungsergebnisse in Lernprojekten, Experimenten oder Seminaren praktisch zu erproben. Ihr Feedback kann helfen, die Benutzerfreundlichkeit, Verständlichkeit und den Lerneffekt der entwickelten Systeme zu verbessern. Zudem können Studierende als Multiplikatoren dienen, indem sie die gewonnenen Erkenntnisse in ihrem akademischen Umfeld verbreiten und damit die Relevanz und den Praxisbezug der Arbeit unterstreichen.

Technologieunternehmen und Industriepartner
Firmen, welche im Bereich der Entwicklung von Automaten, Robotik und KI-Systemen tätig sind, können von diesem Projekt profitieren. Besonders Unternehmen der Bildverarbeitungsindustrie finden in der integrierten Kamera, die Echtzeitdaten zur Steuerung des Flipperautomaten liefert, interessante Anwendungsmöglichkeiten.
Auch die Spielentwicklungsbranche kann von den Projektergebnissen profitieren. Adaptive KI, kombiniert mit präziser Bildverarbeitung, ermöglicht die Analyse von Spielerak-

tionen und das dynamische Anpassen des Spiels. Dieser Ansatz, führt zu immersiven und realitätsnahen Spielerlebnissen und eröffnet neue Perspektiven für personalisierte Spielmechaniken.

4.3 Anforderungen

Die Anforderungsanalyse klärt unter Anderem, welche Anforderungen an das System gestellt werden. Bei der Analyse der Systemanforderungen unterscheidet man grundsätzlich zwei Kategorien:

- **Funktionale Anforderungen (F):** Diese legen die konkreten Funktionen fest, die das System erfüllen muss. Sie definieren, was das System leisten soll.
- **Nicht funktionale Anforderungen (NF):** Diese beschreiben die Eigenschaften, Qualitätsmerkmale und Beschränkungen des Systems und geben an, wie oder unter welchen Bedingungen die Funktionen umgesetzt werden sollen.

Die Anforderungsanalyse dient als Orientierung für die praktische Umsetzung der Masterarbeit. Sie minimiert Risiken, Missverständnisse hinsichtlich des Umfangs oder der falschen Priorisierung von Aufgaben. Hiermit wird die Basis für den Erfolg der Masterarbeit begründet.
In Abbildung 4.2 wird das entsprechende Anwendungsfalldiagramm für das Gesamtsystem dargestellt, welches die zentralen Anwendungsfälle visualisiert, die für das Erreichen der definierten Ziele erforderlich sind.

4.3.1 Anforderungen an die Bildverarbeitung (BV)

Dieser Abschnitt befasst sich mit der Analyse der relevanten Anforderungen, die auf den Ergebnissen der vorherigen Stakeholderanalyse basieren.

Funktionale Anforderungen (F)

BV-F1: Das Spielfeld soll für die Weiterverarbeitung aus dem rohen Bildmaterial ausgeschnitten und zu einem rechtwinkligen Format transformiert werden.
Zur Vermeidung möglicher Fehler bei der Ausrichtung der Kamera oder des Spielfelds, soll das Spielfeld so ausgeschnitten und transformiert werden, dass die arbeitende künstliche Intelligenz eine stabile und sich nicht verändernde Umgebung für die Beobachtung des Zustands erhält.

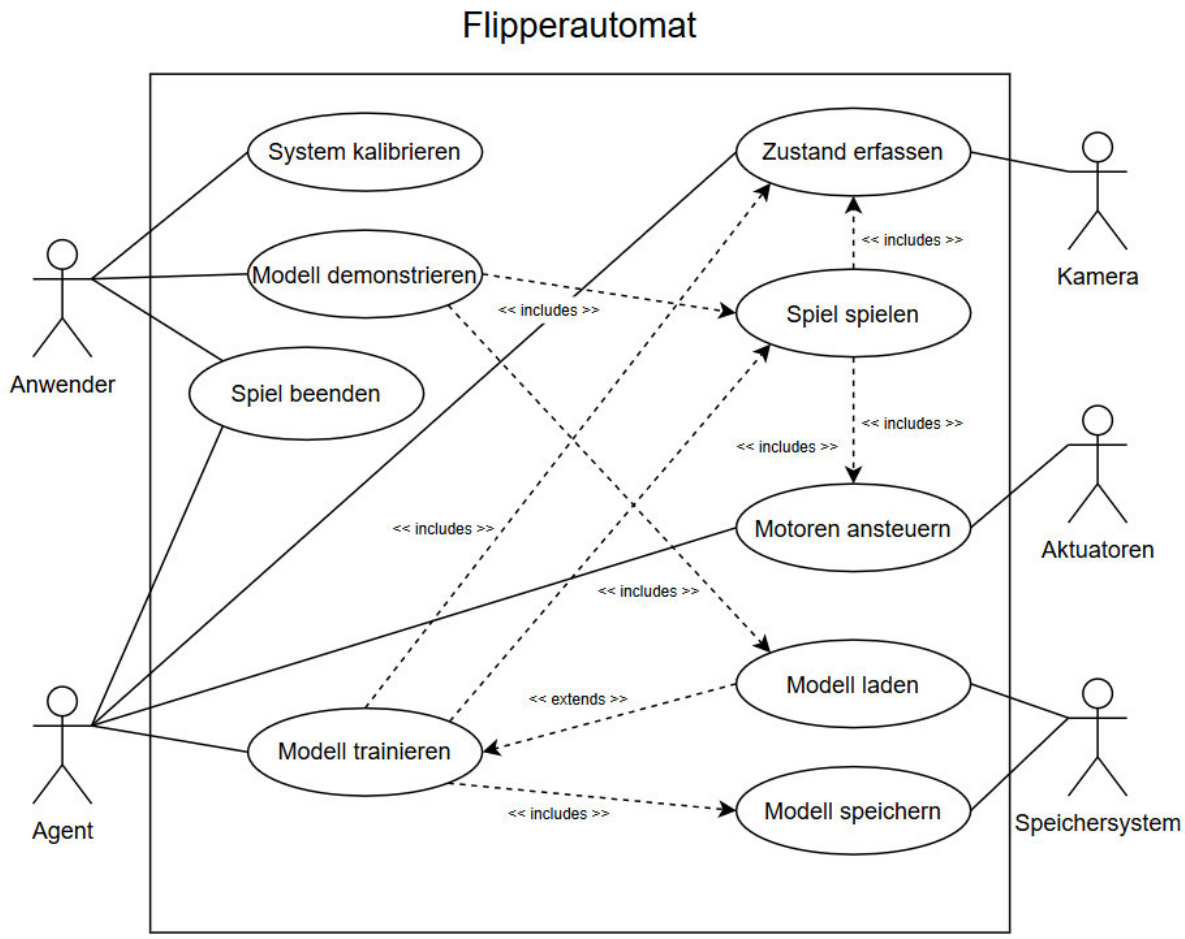


Abbildung 4.2: Anwendungsfalldiagramm

BV-F2: Das extrahierte Spielfeld soll den gesamten Interessensbereich abbilden.
Wenn der gesamte Interessensbereich extrahiert wurde, können alle möglichen Zustände fehlerfrei erkannt werden.

BV-F3: Die Position des Startarms muss erkannt werden.
Beim Startarm des Flipperautomatens können Schrittverluste auftreten. Dies hat zufolge, dass der Startarm selbst die rückführende Öffnung vom Drain in die Startrampe verdecken kann. Dies würde einen fehlerhaften Start des Spiels verursachen und mögliche Folgeprobleme erzeugen.

BV-F4: Die aktuellen Winkel der Flipperarme sollen erkannt werden.
Zur Durchführng von Kalibrierungsmaßnahmen müssen die aktuell vorliegenden Winkel der Flipperarme vorliegen. Hierdurch ist zu jeder Zeit eine Anpassung der Position der Flipperarme möglich. Beim Treffen der Kugel mit den Flipperarmen können Schrittfehler auftreten. Dies kann eine Positionsveränderung der Arme zur Folge haben. Langfristig

wird dies ohne Korrekturmaßnahme zu Fehlern führen.

BV-F5: Der Drain des Flippers soll erkannt werden und die Koordinaten sowie Dimensionierung weitergereicht werden.
Um zu erfassen, wann das Spiel verloren wurde, muss erkannt werden, wenn die Kugel im Drain erscheint.

BV-F6: Die Position der Kugel muss aus den Bilddaten extrahiert werden.
Die Position der Kugel ist für das System essenziell. Mit ihr kann der Agent die benötigten Aktionen auswählen.

Nicht-funktionale Anforderungen (NF)

BV-NF1: Die Bildrate für den Input zur Bildverarbeitung muss ausreichend an die Zeit der Belichtungszeit der Kamera, der Bildverarbeitungszeit und an die Aktionszeit der KI angepasst werden.
Das System gibt durch deine eigenen Beschränkungen vor wie hoch die maximal resultierende Bildrate sein wird. Es soll eine möglichst hohe Bildrate angestrebt werden, damit so viele Aktionen des Agenten wie möglich ausgeführt werden können.

BV-NF2: Um fehlerhafte Aktionen oder Weiterverarbeitungen zu verhindern, müssen die zur Verarbeitung genutzten Bilddaten vorverarbeitet werden.
Bei schlechter oder nicht vorhandener Vorverarbeitung können Zustände des Systems fehlerhaft erkannt werden, was zu falschen Reaktionen der KI führen kann.

4.3.2 Anforderungen an die virtuelle Umgebung

Funktionale Anforderungen (F)

VU-F1: Die Bewegung der Kugel soll den physikalischen Eigenschaften der Realität nachempfunden werden.
Das aus dem Training resultierende Modell kann durch die Erfahrungen aus der virtuellen

Umgebung besser die Bewegungsmuster der realen Flipperkugel bestimmen und voraussagen, wann dieses mit den gleichen physikalischen Gesetzen konfrontiert wird.

VU-F2: Wenn die Kugel in den Drain fällt, soll das Spiel automatisch neu gestartet werden, um mit der Simulation oder dem Training des Agenten fortzufahren.
Hierbei sollen alle bewegten Objekte auf ihren Ursprungspunkt zurück gesetzt werden.

VU-F3: Die Kollisionsberechnung soll den realen Bedingungen am physischen Demonstrator nachempfunden sein.
Gleichgestellt mit den physischen Bedingungen hinsichtlich des Rollverhaltens soll auch bei der Kollision gegen eine Wand oder einen Flipperarm der Austrittswinkel dem der Realität entsprechen.

VU-F4: Das reale Spielfeld des Flippers soll grafisch als virtuelle Umgebung in 3D simulierbar sein.
Die virtuelle Umgebung sollte den reale Flipper proportional abbilden, um eine einfache Vergleichbarkeit zwischen virtueller und realer Spielumgebung zu ermöglichen.

VU-F5: Beim Neustart des Spiels, sollen der Startpunkt und die Startgeschwindigkeit mit einem zufälligem Offset gesetzt werden.
Der zufällige Offset sorgt dafür, dass die Simulation nicht immer mit den gleichen Parametern startet und sich somit wiederholt.

Nicht-funktionale Anforderungen (NF)

VU-NF1: Die Implementierung der virtuellen Umgebung soll in Python erfolgen, um eine nahtlose Integration mit gängigen RL-Frameworks zu ermöglichen. Zur Erstellung der Simulation soll die Bibliothek VPython genutzt werden.

VU-NF2: Proportional zu den physikalischen Abmessungen der Kugel des echten Flipperspiels sollte die virtuell modellierte Kugel relativ erstellt werden, um eine realistische Simulation zu gewährleisten.
Falsche Proportionen könnten zu unnatürlichem Verhalten führen.

VU-NF3: Die Neigung des virtuellen Spielfeldes soll horizontal 7 Grad betragen.
Der physische Demonstrator weist jene Neigung auf. Hierdurch wird ein physikalisch korrektes Rollverhalten der virtuellen Kugel sichergestellt und zutreffende Berechnungen ermöglicht.

VU-NF4: Als virtuelle Umgebung soll das gesamte Spielfeld inklusive Rahmen und Kollisionsobjekten simuliert werden. Die Maße der virtuellen Umgebung müssen relativ zu dem physischen Demonstrator übereinstimmen.
Dadurch wird nach dem erfolgreichen Vortraining in der virtuellen Umgebung eine Basis geschaffen, welche sich in die Realität und später auf dem physischen Demonstrator leichter implementieren lässt.

VU-NF5: Die Rotationsgeschwindigkeit der virtuellen Flipperarme soll an den physischen Demonstrator angepasst sein.
Damit das Reaktionsverhalten der KI in der virtuellen Umgebung sinnvoll trainiert werden kann, müssen die Flipper zeitgleich wie in der physischen Umgebung reagieren. So kann die KI das Spielverhalten in beiden Umgebungen anwenden.

4.3.3 Anforderungen an die künstliche Intelligenz (KI)

Für einen KI-gestützten Flipper, der auf Reinforcement Learning (RL) basiert, müssen spezielle Anforderungen an die KI gestellt werden, um ein realistisches und reaktionsschnelles Spielerlebnis zu gewährleisten. Die KI muss in der Lage sein, aus ihren eigenen Handlungen und den daraus resultierenden Spielzuständen zu lernen und sich kontinuierlich zu verbessern. Dabei ist es entscheidend, dass das Modell in der Lage ist, die richtigen Spielentscheidungen in Echtzeit zu treffen, um auf Veränderungen im Spielgeschehen schnell zu reagieren.

Durch die Anwendung von RL wird die KI darauf trainiert, optimale Strategien für das auf Belohnungen basierende Flipperspiel zu entwickeln. Belohnungen werden für erfolgreiche Aktionen ausgegeben. Zur Minimierung von Fehlern und zur beständigen Verbesserung ist eine präzise Handhabung der Zustände und der Übergänge zwischen den Zuständen nötig. Die KI sollte hierbei effizient arbeiten, um ihre benötigte Rechenleistung auf ein Minimum zu beschränken. Somit kann die KI auf der verfügbaren Hardware optimal laufen.
Durch die Anwendung von Reinforcement Learning wird die KI darauf trainiert, optimale Strategien für das Flipper-Spiel zu entwickeln, basierend auf den Belohnungen, die sie für erfolgreiche Aktionen erhält. Dies erfordert eine präzise Handhabung der Zustände und

der Übergänge zwischen ihnen, um Fehler zu minimieren und das Spielverhalten beständig zu verbessern. Die KI sollte dabei effizient arbeiten, um die benötigte Rechenleistung zu minimieren und auf der verfügbaren Hardware optimal zu laufen.

Funktionale Anforderungen (F)

KI-F1: Nach dem Training in der virtuellen Umgebung soll das entstandene Modell weiter in der realen Umgebung trainiert werden.
Sobald das Modell ausreichend genug trainiert wurde, soll das Modell weiter in der realen Umgebung trainiert werden, damit sich der Agent an die entsprechenden Bedingungen anpassen kann.

KI-F2: Es soll einen Trainingsmodus und einen Evaluations- bzw. Demonstratormodus geben.
Im Trainingsmodus soll der Agent trainiert werden können. Der Demonstrationmodus soll den Trainingsfortschritt und das Verhalten der KI wiedergeben.

KI-F3: Es soll ein Replay Buffer eingerichtet werden.
Ein Replay-Buffer sorgt dafür die Effzienz und Stabilität des Lernprozesses zu verbessern.

Nicht-funktionale Anforderungen (NF)

KI-NF1: Mit Hilfe von Reinforcement Learning soll der daraus resultierende Agent nach den Anforderungen der virtuellen Umgebung trainiert werden.
Reinforcement Learning eignet sich ideal für einen KI-gestützten Flipper, da der Agent durch Versuch und Irrtum selbstständig optimale Strategien erlernen kann. Er optimiert langfristige Spielentscheidungen, ohne dass Regeln manuell programmiert werden müssen. Zudem ermöglicht RL eine kontinuierliche Verbesserung und Echtzeitreaktionen, wodurch ein realistisches und effizientes Spielverhalten entsteht.

KI-NF2: Der Aktionsraum und der Zustandsraum, soll sowohl in der virtuellen Umgebung als auch in der realen Umgebung identisch sein.
So kann der Agent die gleichen Aktionen bei gleichen Zuständen in beiden Umgebungen ausführen.

KI-NF3: Die virtuelle Umgebung soll das gleiche Interface wie das der physischen Umgebung vorweisen.

Damit während der Evaluierung leicht zwischen den Umgebungen gewechselt werden kann, sollen diese das gleiche Interface anbieten können. Zudem kann so der Agent mit beiden Umgebungen gleichermaßen arbeiten.

KI-NF4: Die KI soll mittels des Deep-Q-Learning-Algorithmus trainiert werden.
DQN nutzt ein neuronales Netz als Funktion, um den Wert für jeden Zustand zu schätzen. So ist es möglich mit großen Zustandsräumen zu arbeiten.

4.3.4 Anforderungen an den physischen Demonstrator (PD)

Funktionale Anforderungen (F)

PD-F1: Der Agent muss ist der Lage sein die Flipperarme steuern zu können.
Durch die Bewegung der Flipperarme ist es dem Agenten möglich die Kugel im Spiel zu behalten.

PD-F2: Die Kugel muss nach dem Verlieren des Spiels wieder in die Startrampe gelangen.
Der Rücklauf der Kugel nach abgeschlossenem Spiel, ist nicht nur für den Spielfluss wichtig, sondern garantiert auch ein reibungsloses Training des Agenten.

PD-F3: Die nächste auszuführende Aktion soll mit Hilfe eines RL-Algorithmus bestimmt werden. **PD-F4:** Vor dem Beginn jedes Spieles sollen die Flipper zurück in die Ruhelage gesetzt werden.
Dies garantiert die gleichen Startbedingungen für jeden Durchlauf und verhindert fehlerhafte Motoransteuerungen.

Nicht-funktionale Anforderungen (NF)

PD-NF1: Die Winkel der Flipperarm müssen in einer Flucht zu den Inlanes liegen.
Die Inlanes besitzen einen Rotationwinkel von -35 Grad (links)/ +35 Grad (rechts). Somit müssen die Flipperarme die gleichen Winkel im Ruhezustand aufweisen.
Im aktivierten Zustand sollen die Flipperarme nicht mehr als um +70 Grad (links)/ -70 Grad (rechts) ausgeschlagen werden, sodass diese einen Endrotationwinkel von +35 Grad (links) und -35 Grad (rechts), bezogen auf die Rotationachse, aufweisen.

5 Konzept

Ziel dieser Masterarbeit ist es, einen KI-gestützten Flipperautomaten zu entwickeln, der durch die Kombination von Reinforcement Learning (RL), Bildverarbeitung und einem digitalen Zwilling ein autonomes Spielerlebnis ermöglicht. Der Flipperautomat soll in der Lage sein den Zustand des Spiels in Echtzeit zu analysieren und die Flipperarme strategisch zu steuern, sowie in Trainingsphasen kontinuierlich zu lernen und sich zu verbessern. Zentraler Bestandteil des Systems ist eine Kamera, die Informationen über den Zustand des Systems erfasst. Diese Daten werden an ein RL-Modell weitergegeben, welches optimale Steuerbefehle für die Flipperarme berechnet. Der digitaler Zwilling dient dazu als virtuellen Umgebung das Modell zu trainieren und anzupassen, bevor die Änderungen auf den physischen Automaten übertragen werden.
Die Bildverarbeitung dient dazu relevante Informationen in Echtzeit über den aktuellen Systemstatus zu extrahieren, beispielsweise die Position der Flipperkugel, die Ausrichtung der Flipperarme und weitere physikalische Parameter. Die vorverarbeiteten Bilddaten bilden die Grundlage für die weitere Entscheidungsfindung des KI-Systems.
Darüber hinaus fokussiert sich ein weiterer Teil auf die Definition des Lernsystems, zur Steuerung des KI-gesteuerten Flippers. Hierzu sind die nachfolgenden Zustands- und Aktionsräume zu definieren:

- *Zustandsraum:* Die verarbeiteten Bilddaten liefern detaillierte Informationen über den aktuellen Zustand des Flipperautomaten. Dieser Zustand umfasst die Position der Flipperkugel und den Winkel der Flipperarme

- *Aktionsraum:* Die möglichen Aktionen umfassen sämtliche Steuerbefehle, welche über die Motoransteuerung an den Flipperautomaten ausgegeben werden können, zum Beispiel das Aktivieren der Flipperarme.

Auf Basis dieser Definitionen wird ein RL-Agent aufgebaut, welcher lernt durch Interaktion mit der Umgebung optimale Spielstrategien zu entwickeln. Hierbei werden verschiedene Belohnungsstrategien erarbeitet, die das gewünschte Verhalten der KI fördern. Positive Belohnungen signalisieren erfolgreiche Aktionen. Negative Belohnungen signalisieren nicht-erfolgreiche Aktionen - Fehlentscheidungen oder ineffiziente Bewegungen. Die Nutzung eines digitalen Zwillings unterstützt zudem den Lernprozess. Hierdurch können in einer virtuellen Umgebung Simulationen zu einem schnelleren Lernerfolg führen.
Dieses dreigeteilte Konzept ermöglicht es durch robuste Bildverarbeitung eine präzise Umgebungswahrnehmung zu gewährleisten und mittels eines gezielten RL-Ansatzes eine adaptive und effektive Steuerung des Flipperautomaten zu realisieren.

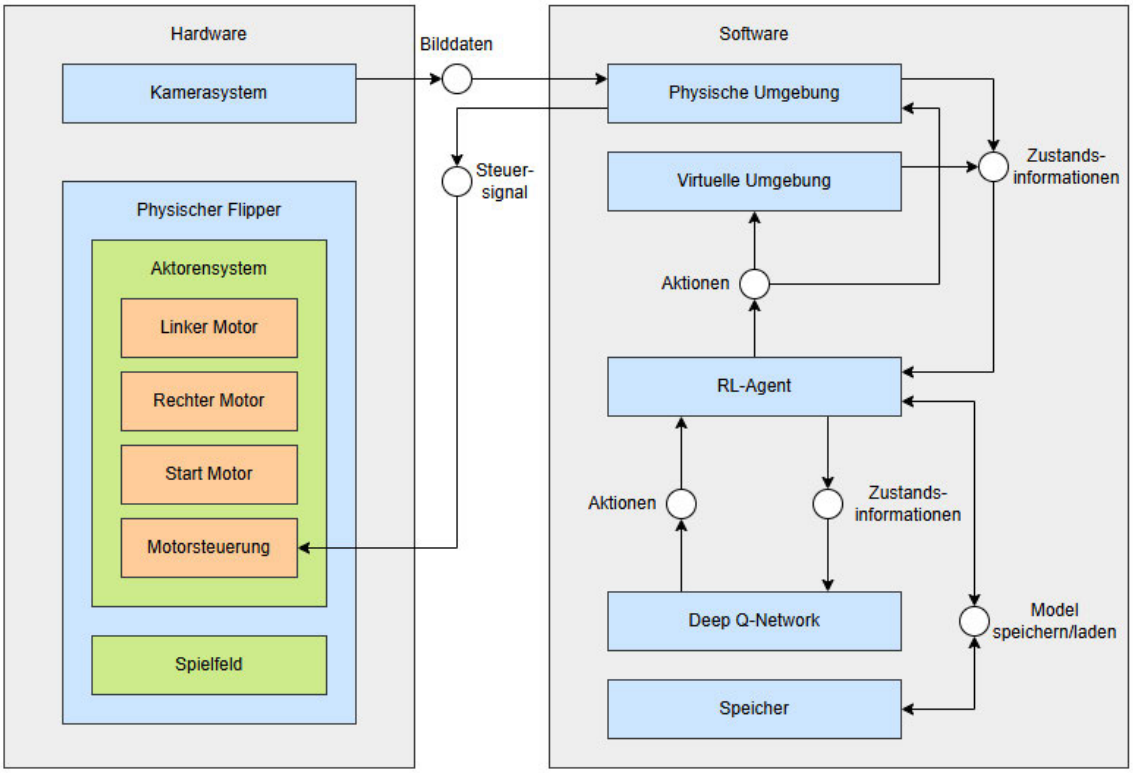


Abbildung 5.1: Darstellung der Systemarchitektur

5.1 Systemarchitektur

Abbildgun 5.1 zeigt, dass das System in zwei Bereiche aufgeteilt ist. Zum Einen der Bereich der Hardware, welcher dafür zuständig ist die benötigten Bilddaten zu erfassen und die ausgewählten Aktionen des Agenten umzusetzen. Zum Anderen der Bereich der Software, in welchem die Einrichtung der jeweiligen Umgebung erfolgt. Hierdurch sollen gleichermaßen die Zustandsinformationen des laufenden Sysems an den RL-Agenten ausgeben werden. Diese Daten werden vom Agenten an das Deep Q-Network weitergeleitet. In der Folge wird dann die resultierende Aktion ausgegeben und durch den Agenten an die jeweilige Umgebungen weitergeleitet. Im Falle der physischen Umgebung, wird die Aktion mittels Steuersignalen an die Motorsteuerung des Aktorensystems weitergeleitet, welche im Anschluss die jeweiligen Motoren ansteuert.

5.2 Kamera

Für die Bildverarbeitung des Flipper-Prototyps wurde die Kamera *Daheng Imaging Venus VEN-161-61U3C* ausgewählt. Diese bietet eine Auflösung von 1440×1080 Pixeln, eine Bildrate von 61 fps und einen Global Shutter, sowie einen 1/2,9-Zoll-Sensor. Der entscheidende Vorteil dieses Modells liegt in der Kombination aus hoher Bildrate und Global-Shutter-Technologie mit einem Preis von lediglich 149 €, welcher für eine Indus-

Kamera	Auflösung	Sensorgröße	FPS	Shutter	Preis
AV Alvium 1800 U-319c	2064x1544	1/1,8”	54	Global	540 €
AV Alvium 1800 U-500c	2592x1944	1/2,5”	68	Rolling	310 €
DI Venus VEN-161-61U3C	1440x1080	1/2,9”	61	Global	149 €
DI Venus VEN-505-36U3C	2592x1944	1/2,8”	36	Rolling	149 €

Tabelle 5.1: Vergleich der vorhandenen Kameras für den Flipper-Prototyp

triekamera einen niedrigeren Preis darstellt. Zudem ist es möglich mit dieser Industriekamera Wechselobjektive zu nutzen
Im direkten Vergleich zu den verfügbaren Alternativen ergeben sich eindeutige Gründe für die Auswahl:

- Die *Allied Vision Alvium 1800 U-319c* verfügt mit 2064×1544 Pixeln zwar über eine höhere Auflösung, eine Bildrate von 54 fps und einen größeren 1/1,8-Zoll-Sensor, ist jedoch mit 540 € etwa viermal teurer. Aufgrund der begrenzten Stückzahl (ein Sponsoring-Exemplar) kann dieses Modell nicht zur Verfügung gestellt werden.

- Die *Allied Vision Alvium 1800 U-500c* bietet eine noch höhere Auflösung von 2592×1944 Pixeln und 68 fps, setzt jedoch auf einen Rolling Shutter. Bei sich schnell bewegenden Objekten wie Flipperkugeln kann dies zu Bewegungsartefakte führen.

- Die *Daheng Imaging Venus VEN-505-36U3C* besitzt mit 2592×1944 Pixeln eine hohe Auflösung und ist preislich attraktiv (149 €), erreicht jedoch nur 36 fps und arbeitet ebenfalls mit Rolling Shutter. Für den Flippereinsatz ist die Bildrate damit zu gering, da die schnelle Kugelbewegung eine zeitlich hochaufgelöste Erfassung erfordert.

Damit stellt die *VEN-161-61U3C* aus den folgenden Gründen die beste Option für den geplanten Einsatz dar: Sie liefert bei niedrigen Kosten eine ausreichend hohe Auflösung, eine Bildrate oberhalb von 60 fps und einen Global Shutter, der eine verzerrungsfreie Erfassung der hochdynamischen Flipperkugel ermöglicht. Während die VEN-505-36U3C eine hohe Auflösung bei geringerer Bildrate bietet und somit für eine statische Anwendung geeignet ist, zeichnet sich die VEN-161-61U3C eher für hochdynamische Szenarien wie den Flipper aus. Auf diese Weise lässt sich trotz Budgetknappheit eine praxisnahe Arbeit mit Industriekameras in verschiedenen Projekten sicherstellen.

Für den KI-gesteuerten Flipperautomaten ist eine hohe Bildrate entscheidend, da die Kugelbewegungen sehr schnell und unvorhersehbar sind. Nur durch eine ausreichend hohe Bildrate kann die Kugel bei jeder Position präzise erfasst werden, sodass die Bildverarbeitung zuverlässig arbeitet. Eine zu geringe Framerate würde dazu führen, dass Bewegungen zwischen den Frames nicht abgebildet werden, was zu ungenauen Positionsinformationen

und Fehlern im Steueralgorithmus führen kann. Der Einsatz eines Global Shutters in Kombination mit über 60 fps ermöglicht zudem eine verzerrungsfreie Erfassung der Kugel, was insbesondere bei Hochgeschwindigkeitsbewegungen von zentraler Bedeutung ist.

5.3 Virtuelle Umgebung

Mithilfe einer Simulationsumgebung können Agenten unter kontrollierten und leicht reproduzierbaren Bedingungen getestet und trainiert werden.

5.3.1 Design

Für das Vortraining des KI-Agenten wird eine virtuelle Umgebung benötigt, die das Verhalten des Flippers realistisch simuliert. Ziel dieser Umgebung ist es, dem Agenten ein sicheres, kontrolliertes Umfeld zu bieten, in dem er verschiedene Strategien ausprobieren kann, ohne Risiken für einen physischen Automaten einzugehen. Die Umgebung soll eine realistische Interaktion der Kugel mit Flipperarmen, Hindernissen und Rahmen ermöglichen und gleichzeitig alle relevanten Zustände für das Reinforcement Learning bereitstellen.

Die virtuelle Umgebung umfasst ein Spielfeld, welches durch feste Rahmenstrukturen und Banden begrenzt wird. Diese stellen die physikalischen Grenzen des Flipperautomaten dar. Innerhalb des Spielfelds werden sowohl dynamische als auch statische Elemente simuliert. Die Kugel simuliert ein dynamisches Objekt, welches durch Gravitation, Kollisionen und die Flipperarme beeinflusst wird. Die Flipperarme selbst sind bewegliche Objekte mit definierten Rotationsachsen und begrenzten Winkelbereichen, die der Agent gezielt steuern kann. Weitere Elemente wie Inlanes, Launcher-Lane, Arc-Lane und Lose-Lane dienen zur Strukturierung der Bewegungsräume der Kugel und Erzeugung unterschiedlicher physikalischer Interaktionen.
Die visuelle Umsetzung erfolgt, wie beispielsweise in Abbildung 5.2 gezeigt, über eine Simulationsumgebung, die sowohl die statischen Rahmen und Bahnen als auch die dynamischen Objekte wie Kugel und Flipperarme rendert. In jedem Simulationsschritt werden die Positionen der Kugel und die Winkel der Flipper aktualisiert, sodass Bewegungen und Rotationen nachvollziehbar und konsistent dargestellt werden.
Physikalisch modelliert die Umgebung Effekte wie Gravitation, Kollisionen und die Wechselwirkung der Kugel mit den Flipperarmen. Die Rotationsbewegungen der Flipper übertragen Kräfte auf die Kugel und ermöglichen dem Agenten, die Kugel gezielt zu beschleunigen oder ihre Richtung zu verändern. Durch die Integration dieser physikalischen Mechanismen soll der Agent lernen, die Kugel aktiv im Spiel zu halten und Situationen zu erkennen, in denen ein Spielverlust droht.

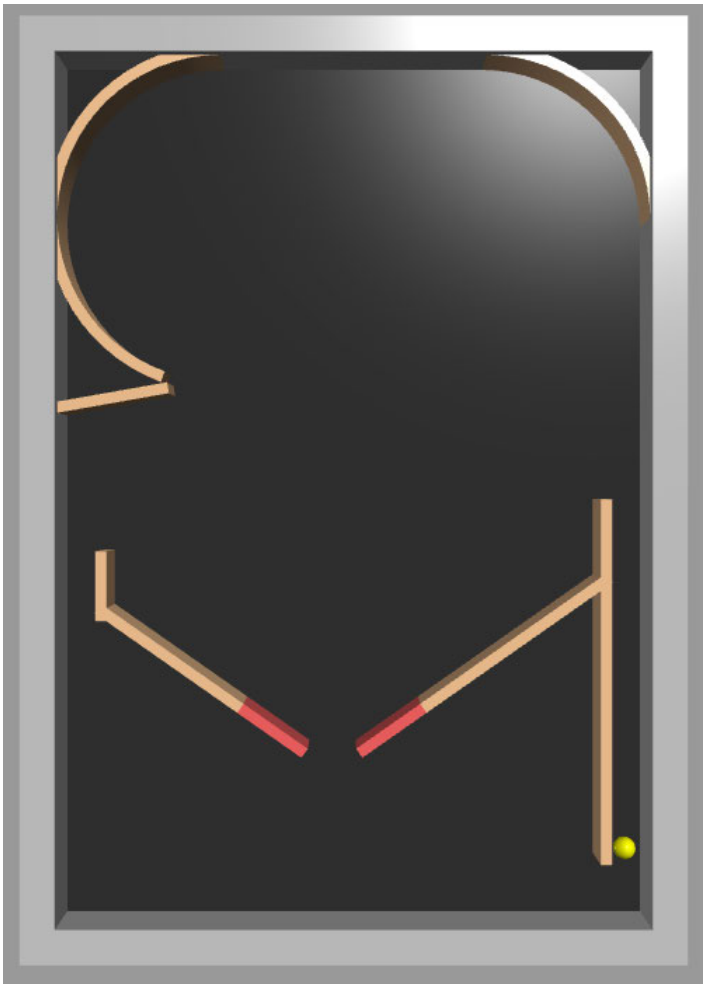


Abbildung 5.2: Mögliche Grundstruktur der virtuellen Umgebung

Für die weitere Entwicklung muss die Umgebung noch vollständig implementiert werden. Dazu gehören die präzise Berechnung von Kollisionen, die korrekte Umsetzung der Gravitation und die dynamische Aktualisierung aller beweglichen Objekte, sowie die Schnittstelle zur Ansteuerung durch den KI-Agenten. Zusätzlich sollten Mechanismen zur Erfassung von Zuständen und Ereignissen eingebaut werden, damit diese Informationen für die Belohnungsberechnung im Reinforcement Learning genutzt werden können. Ziel ist es eine robuste, realistische und flexibel konfigurierbare Trainingsumgebung zu schaffen, die den Agenten optimal auf reale oder detailliertere Simulationen vorbereitet.

5.3.2 Struktur

Im Zentrum steht die Implementierung einer Klasse, welche die Standardschnittstellen von `gym.Env` erfüllen soll. Sie umfasst die wesentlichen Methoden `step()` und `reset()`, über die die Interaktion zwischen Agent und Umgebung abgebildet wird.
Die `step()`-Methode soll so konzipiert werden, dass sie eine gewählte Aktion auf den physikalischen Zustand anwendet, mehrere Simulationsschritte zur Abbildung realistischer

Reaktionszeiten durchführt und anschließend den neuen Beobachtungsvektor, die ermittelte Belohnung sowie Informationen über Episodenende oder Abbruch zurückgibt.
Die `reset()`-Methode initialisiert die Umgebung neu, indem die Flipper auf die Ausgangsstellung gesetzt und die Kugel in einer definierten Startposition platziert wird. Zusätzlich sollen `render()`- und `close()`-Methoden implementiert werden, um die grafische Ausgabe zu aktualisieren bzw. Ressourcen nach Beendigung der Simulation freizugeben.

5.4 Physische Umgebung

Neben der virtuellen Simulation soll auch eine physische Umgebung entwickelt werden, in der ein realer Flipperautomat über eine Schnittstelle an das `gymnasium`-Framework angebunden wird. Dieses Konzept dient dazu, das zuvor im Simulator trainierte Modell in die reale Welt zu übertragen und dort weiter zu optimieren.
Kernstück wird eine Klasse, die – analog zur virtuellen Variante – die gym-typischen Methoden `step()` und `reset()` implementieren soll. Der entscheidende Unterschied liegt darin, dass nicht eine simulierte Physik-Engine, sondern reale Hardwarekomponenten angesteuert werden. Dazu zählen insbesondere der Flipperautomat selbst sowie die Kamera, welche die Position der Kugel und die Bewegungen der Flipper kontinuierlich erfasst.
Das Beobachtungsmodell (`observation_space`) wird so konzipiert, dass die durch die Kamera gewonnenen Informationen in normierter Form an den Agenten übergeben werden. Vorgesehen sind dabei ebenfalls die Winkelstellungen der Flipper sowie die aktuelle und vorherige Position der Kugel. Die Normierung dient dazu, die Daten direkt für neuronale Netze nutzbar zu machen und eine einheitliche Schnittstelle zwischen virtueller und physischer Umgebung zu gewährleisten.
Der Aktionsraum (`action_space`) soll – wie bei der Simulation – vier diskrete Aktionen umfassen: beide Flipperarme in Ruhelage, linker Flipper oben und rechter unten, rechter Flipper oben und linker unten oder beide oben. Diese Aktionen werden über die Ansteuerung der realen Flipperarme des Demonstrators umgesetzt.
In der `step()`-Methode soll eine vom Agenten gewählte Aktion in konkrete Hardwarebefehle übersetzt werden. Gleichzeitig erfasst die Kamera ein neues Bild, welches ausgewertet und in eine Beobachtung transformiert wird. Ergänzend wird über eine Belohnungsfunktion der Fortschritt des Agenten bewertet. Auf diese Weise entsteht ein kontinuierlicher Kreislauf aus Beobachtung, Aktion und Rückmeldung, der direkt auf der realen Maschine abläuft.
Die `reset()`-Methode wird dafür sorgen, dass der physische Aufbau in einen definierten Startzustand versetzt wird. Dies umfasst unter Anderem das Zurücksetzen der Flipper in die Ausgangsstellung und die Rückführung der Kugel durch Pausezeiten. Zusätzlich ist

vorgesehen Kalibrierungsfunktionen für das Kamerasystem einzubinden, damit eine zuverlässige Erkennung gewährleistet bleibt.
Das Konzept sieht vor, dass die physische Umgebung dieselbe Schnittstelle wie die virtuelle bietet, sodass ein trainierter Agent ohne Anpassungen in beiden Umgebungen eingesetzt werden kann. Dadurch wird ein nahtloser Übergang vom Training im Simulator hin zum Training am realen Demonstrator ermöglicht.

5.5 Reinforcement Learning

Das Reinforcement Learning (RL) bildet das methodische Fundament für die Steuerung des autonomen Flipperautomaten. Ziel ist es, einen Agenten zu entwickeln, der durch Interaktionen mit seiner Umgebung lernt, optimale Entscheidungen zu treffen und so die Kugel möglichst lange im Spiel zu halten. In diesem Kapitel werden zunächst der Aufbau und die Funktionsweise des geplanten DQN-Agenten beschrieben, gefolgt von der Definition von Zustands- und Aktionsraum. Abschließend wird das Belohnungssystem vorgestellt, das als zentrale Steuergröße das Lernverhalten des Agenten prägt.

5.5.1 DQN-Agenten

Der geplante Agent soll auf dem *Deep Q-Network* (DQN) Ansatz basieren, welcher neuronale Netze zur Approximation der Q-Funktion einsetzt. Im Folgenden wird der vorgesehene Aufbau sowie die zentralen Komponenten des Agenten beschrieben:

Hyperparameterverwaltung
Die Hyperparameter des Agenten (z. B. Lernrate, Batch-Größe, Epsilon-Strategie, Discountfaktor γ) sollen in einer externen YAML-Datei verwaltet und beim Initialisieren eingelesen werden. YAML bietet eine leicht lesbare, hierarchische Struktur und ist dadurch besser für die Organisation komplexer Konfigurationsdaten geeignet als beispielsweise reine Text- oder CSV-Dateien. Dies ermöglicht eine klare Trennung von Implementierungslogik und Parametrisierung, was insbesondere bei experimentellen Vergleichen und reproduzierbaren Trainingsläufen vorteilhaft ist.

Exploration vs. Exploitation
Zur Steuerung der Aktionsauswahl soll eine *epsilon-greedy Strategie* implementiert werden. Der Agent wählt mit Wahrscheinlichkeit ϵ eine zufällige Aktion (Exploration) und ansonsten diejenige Aktion, welche aktuell den höchsten Q-Wert besitzt (Exploitation). Diese Strategie ist einfach umsetzbar, rechenökonomisch und hat sich in der Literatur als Standardmethode zur Balancierung von Exploration und Exploitation etabliert. Der Wert von ϵ soll nach jeder Episode dekrementiert werden, bis ein Minimalwert erreicht ist, wo-

durch der Agent in frühen Phasen vermehrt exploriert und später stärker auf bekannte Strategien setzt.

Tiefes Q-Netzwerk

Das tiefe Q-Netzwerk (DQN) soll als künstliches neuronales Netz realisiert werden, dessen Architektur durch Hyperparameter definiert ist (bis zu drei vollverbundene Schichten). Die Verwendung vollverbundener Schichten bietet eine gute Balance zwischen Flexibilität und Implementierungsaufwand, während gleichzeitig eine ausreichende Ausdrucksstärke für typische RL-Umgebungen gewährleistet ist. Als Optimierungsverfahren soll der *Adam-Optimierer* eingesetzt werden, da er eine adaptive Lernratenanpassung erlaubt und sich insbesondere bei nichtstationären Problemen wie dem Reinforcement Learning als leistungsfähig erwiesen hat. Als Verlustfunktion ist die `Mean Squared Error (MSE)` vorgesehen, da diese einen direkten Vergleich zwischen den approximierten Q-Werten und den Zielwerten (basierend auf dem Temporal Difference Fehler) erlaubt.

Replay Buffer

Erfahrungen des Agenten (s, a, r, s', d) sollen in einem *Replay Buffer* gespeichert werden. Dieses Verfahren hat den Vorteil, dass die Daten beim Training nicht in der zeitlichen Reihenfolge der Interaktionen genutzt werden müssen. Stattdessen werden Batches zufällig aus der Vergangenheit gezogen, wodurch Korrelationen in den Daten reduziert und die statistische Effizienz verbessert wird. Zudem können durch den Replay Buffer seltene, aber wichtige Erfahrungen mehrfach für das Training genutzt werden und somit die Stabilität und Effizienz des Lernprozesses erhöhen.

Trainingsschritte (Temporal Difference Learning)

Der Agent soll in regelmäßigen Abständen Trainingsschritte durchführen, deren Frequenz über den Hyperparameter `learn_period` gesteuert wird. Für jedes Trainingsbatch wird der *Temporal Difference (TD)-Fehler* berechnet:

$$TD = r + \gamma \cdot \max_{a'} Q(s', a') - Q(s, a). \tag{5.1}$$

Dieses Verfahren basiert auf dem Prinzip der dynamischen Programmierung und erlaubt es, den Wert einer Handlung schrittweise zu verbessern, ohne dass ein vollständiges Modell der Umgebung benötigt wird. Für terminale Zustände entfällt der Zukunftsanteil $Q(s',a')$, sodass sich der Fehler zu $TD = r - Q(s,a)$ vereinfacht. Die Minimierung dieses Fehlers durch Backpropagation ist ein bewährter Ansatz, um die Q-Funktion stabil und effizient zu approximieren.

Speichern und Laden von Modellen

Zur Sicherung des Trainingsfortschritts ist vorgesehen Methoden zum Speichern und La-

den der Netzwerkgewichte zu implementieren. Dadurch wird eine Fortsetzung des Trainings, sowie eine Evaluierung eines trainierten Agenten ermöglicht. Die Wahl dieser Funktionalität ist insbesondere im Kontext von vergleichenden Experimenten und der Wiederverwendbarkeit von Modellen sinnvoll.

5.5.2 Zustands- und Aktionsraum

Für das Reinforcement Learning im Flipper-Szenario müssen Zustands- und Aktionsraum klar definiert sein. Diese Räume bestimmen, wie der Agent seine Umgebung wahrnimmt und welche Eingriffsmöglichkeiten er besitzt.

Zustandsraum
Der Zustandsraum repräsentiert den aktuellen Zustand der Spielfläche und der dynamischen Objekte. In der Implementierung ist der Zustand als kontinuierlicher Vektor zu modellieren, welcher aus folgenden Komponenten besteht:

- **Flipperarmwinkel:** Die Rotationswinkel beider Flipperarme werden in einem normierten Bereich von [0,1] angegeben. Dies bildet die momentane Stellung der Flipperarme ab.
- **Kugelposition:** Die zweidimensionale Position der Kugel wird in den Normalisierungsbereich [0,1] überführt. Damit werden Spielfeldgröße und Koordinatenlage abstrahiert.
- **Vorherige Kugelposition:** Zusätzlich wird die Position der Kugel im letzten Zeitschritt gespeichert und bereitgestellt. Damit kann der Agent Bewegungsrichtungen und Trends ableiten.

Hierdurch ergibt sich ein kontinuierlicher Zustandsraum von sechs Werten, der sowohl die Dynamik der Kugel als auch den Steuerungszustand der Flipper abbildet:

- Winkel des linkes Flipperarms und des rechten Flipperarms,
- X- und Y-Koordinate der Kugel,
- Letzte X- und Y-Koordinate der Kugel,

Durch die Normalisierung werden Wertebereiche vereinheitlicht und somit die Stabilität des Trainings verbessert.

Aktionsraum

Der Aktionsraum ist als diskreter Raum mit vier möglichen Aktionen zu modellieren. Er beschreibt die Steuerungsmöglichkeiten der beiden Flipperarme in binär codierter Form:

- 0: Beide Flipper unten,
- 1: Rechter Flipper oben, linker Flipper unten,
- 2: Linker Flipper oben, rechter Flipper unten,
- 3: Flipper beide oben.

Die Aktionen sind so gestaltet, dass sie einfach interpretierbar sind und dennoch ausreichend strategische Vielfalt bieten. Damit kann der Agent entscheiden, ob er gezielt mit einem einzelnen Flipper eingreift oder beide Flipper gleichzeitig aktiviert.

5.5.3 Belohnungssystem

Das Ziel des Belohnungssystems ist es, den Agenten so zu trainieren, dass er die Kugel möglichst lange im Spiel hält, sie aktiv bewegt und dabei eine effiziente Nutzung der Flipperarme erlernt. Das Belohnungssystem dient somit als zentrale Steuergröße, die das Verhalten des Agenten formt und eine Balance zwischen defensivem und aktivem Spielverhalten ermöglicht.

Kugel im Spiel halten

Der zentrale Bestandteil des Belohnungssystems ist die Aufrechterhaltung des Spielzustandes. Solange die Kugel nicht verloren geht, erhält der Agent in jedem Zeitschritt eine kleine positive Belohnung. Diese Funktion soll den Agenten dazu anregen, Strategien zu entwickeln, die die Kugel möglichst lange im Spielfeld halten. Im Falle eines Spielverlustes wird eine hohe negative Strafe vergeben.

Vertikale Position der Kugel

Die vertikale Position der Kugel stellt einen weiteren wesentlichen Belohnungsfaktor dar. Je höher sich die Kugel auf dem Spielfeld befindet, desto größer ist die Wahrscheinlichkeit, dass sie länger aktiv im Spiel bleibt und zusätzliche Interaktionen mit Spielelementen erzeugt. Daher erhält der Agent eine Belohnung, die proportional zur Höhe der Kugel steigt. Das Spielfeld wird dazu in Zonen unterteilt: Befindet sich die Kugel im oberen Drittel, wird eine hohe Belohnung vergeben, im mittleren Bereich eine moderate Belohnung und im unteren Drittel eine Strafe. Dieses Prinzip soll verhindern, dass der Agent lernt, die Kugel dauerhaft in den unteren Spielfeldregionen zu halten

Bewegung der Kugel

Neben der Position spielt auch die Dynamik der Kugel eine Rolle. Eine Kugel, die sich in horizontaler oder vertikaler Richtung aktiv bewegt, wird mit einer positiven Belohnung verstärkt. Im Gegensatz dazu wird eine Kugel, die ihre Position über mehrere Zeitschritte kaum verändert, mit einer Strafe belegt. Diese Komponente soll insbesondere verhindern, dass der Agent die Kugel zwischen den Flipperarmen einklemmt oder unbeweglich macht, da dies dem eigentlichen Spielprinzip widerspricht.

Nutzung der Flipper

Die Nutzung der Flipperarme wird ebenfalls in das Belohnungssystem integriert. Jeder Einsatz eines Flippers führt zunächst zu einer leichten Strafe, um ein dauerhaftes und ineffizientes Nutzen der Flipper zu vermeiden. Diese leichte Strafe soll progressiv ansteigen, wenn derselbe Flipper mehrfach hintereinander ohne erkennbaren Nutzen eingesetzt wird. Wird hingegen kein Flipper ausgelöst, soll der Agent eine neutrale bis leicht positive Belohnung erhalten. Damit wird ein Gleichgewicht zwischen notwendigem, gezieltem Einsatz der Flipper und der Vermeidung übermäßiger Aktivität geschaffen. Optional kann die Belohnung für Flipperaktionen auch mit der resultierenden Kugelbewegung korreliert werden, sodass nur solche Flips, die die Kugel effektiv beeinflussen, neutral oder leicht positiv gewertet werden.

Erwartetes Lernverhalten

Insgesamt soll der Agent durch diese Struktur folgende Strategien erlernen:

- die Kugel so lange wie möglich im Spiel zu halten,
- bevorzugt in höheren Spielfeldregionen zu agieren,
- die Kugel aktiv und dynamisch zu bewegen,
- die Flipper nur gezielt und effizient einzusetzen
- und schließlich das Risiko eines Spielverlustes insbesondere durch die Mitte zu minimieren.

Durch die Kombination von positiven und negativen Verstärkungen entsteht ein ausgewogenes Belohnungssystem, das sowohl defensives als auch aktives Verhalten des Agenten fördert und somit eine realistische, spielnahe Strategieentwicklung ermöglicht.

6 Entwicklung

6.1 Nacharbeiten am physischen Demonstrator

Der physische Demonstrator, verfügt über keine Kamera. Eine Kamera, welche im System als Sensor dient, ist jedoch notwendig die Eigenschaften der VEN-161-61U3C etablieren diese als beste Option, sodass diese an den Flipper montiert wurde. Hierzu wurde eine aus Aluprofilen gefertigte Halterung an den Rahmen des Spielfeld befestigt. Abbildung 6.1 zeigt den physischen Flipper ohne Motorsteuerung.
Um eine erfolgreiche Umsetzung dieser Masterarbeit zu ermöglichen stellte Prof. Dr. Marc Hensel die Halterung zur Verfügung. Diese bildet nunmehr zusammen mit dem Demonstrator das Flippersystem. Die Kamera wurde mit der Software über ein USB3.0 Interface verbunden. Das Interface wurde vom Hersteller durch die Hardware zur Verfügung gestellt und garantiert die reibungslose Datenübertragung der Aufnahmen.

Abbildung 6.1: Physischer Aufbau mit der montierten Kamera

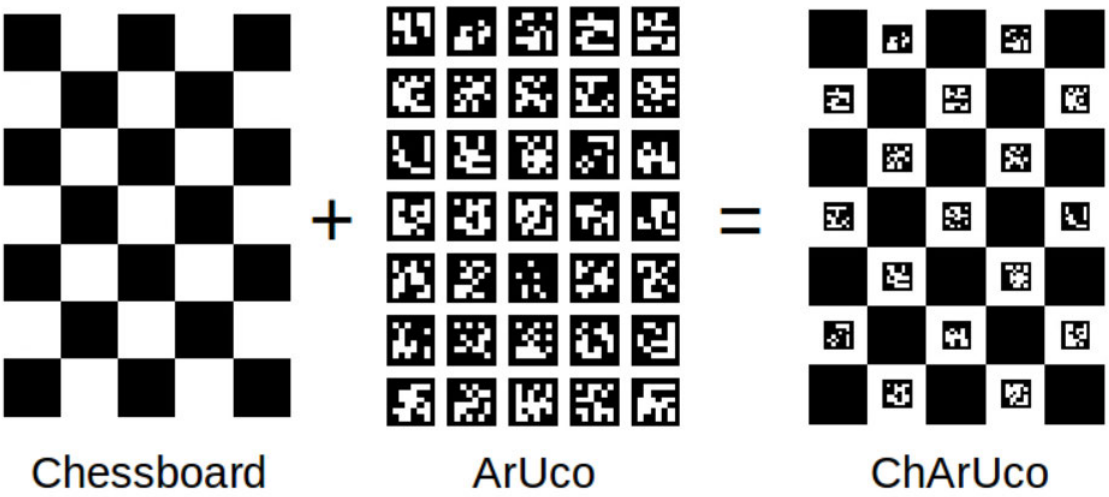


Abbildung 6.2: ChArUco Board

6.2 Entwicklung der physischen Umgebung

6.2.1 Korrekturen von Verzerrungen

Je nach Verwendung der Kamera können Verzerrungen im Bild entstehen oder auftreten. Am Bekanntesten ist der Fischaugeneffekt oder Weitwinkeleffekt. Der Fischaugeneffekt ist eine unerwünschte Verzerrung, welche die Kameras im Bild hervorrufen. Sie lassen die Szene rundlich, in der Mitte „aufgeblasen“ und an den Seiten „gequetscht“ erscheinen[16]. Neben der Möglichkeit die Kamera von dem Fokuspunkt zu entfernen, ist es softwareseitig möglich ungewollte Effekte zu entzerren.

ChArUco Boards
Ein ChArUco Board ist eine Kombination aus Charuco-Markern und ArUco-Markern, welche für präzise Kamerakalibrierung und Posenschätzung in der Computer Vision verwendet wird. Es besteht aus einem Schachbrettmuster mit eingebetteten ArUco-Markern, wodurch es möglich ist, auch teilweise sichtbare Muster zu erkennen und genaue Eckpunkte zu bestimmen.

Mit einem ChArUco Board, wie in Abbildung 6.2[17] zu sehen, kann eine hochpräzise Kamerakalibrierung durchgeführt werden, da die Eckpunkte der Schachbrettfelder mithilfe der ArUco-Marker exakt lokalisiert werden können. Dies verbessert die Genauigkeit gegenüber herkömmlichen Schachbrettmustern.
Mittels des ChArUco Boards soll der Fischaugeneffekt entzerrt werden. In der Folge kön-

[16] `https://storungssuche.com/`, abgerufen am 31. März 2025.
[17] `https://docs.opencv.org/3.4/df/d4a/tutorial_charuco_detection.html`, abgerufen am 31. März 2025.

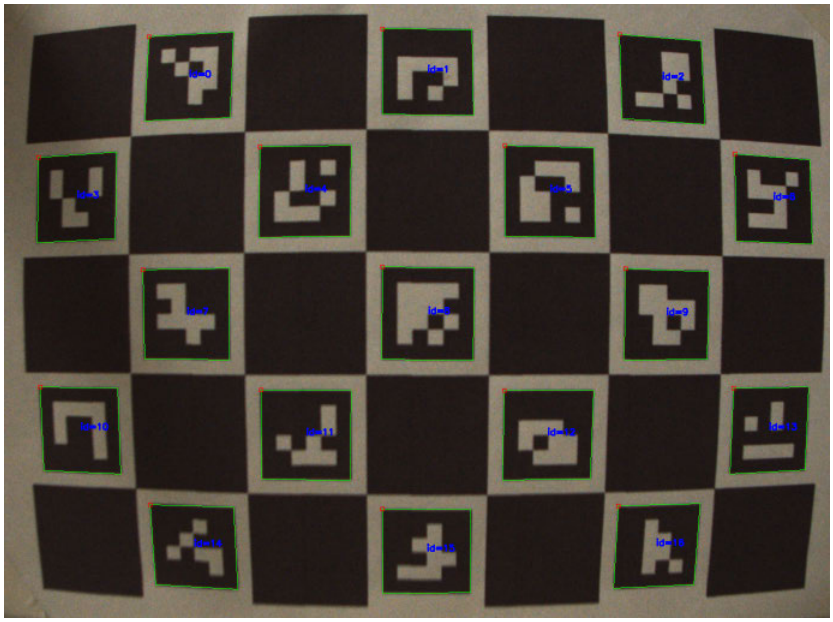


Abbildung 6.3: Analyse der ChArUco-Marker

nen Kanten und Formen durch die Bildverarbeitung einfacher erkannt werden.

Für die Kamerakalibrierung wurde eine bestehende Python-Implementierung[18] verwendet, die auf den Funktionen der OpenCV-Bibliothek basiert. Diese stellt eine Klasse zur Verfügung, mit der sich der gesamte Prozess – von der Erzeugung des Kalibrierungsboards bis hin zur Speicherung der berechneten Parameter – abbilden lässt. Die Klasse wurde im Rahmen des Projekts von Prof. Dr. Marc Hensel entwickelt und für die vorliegende Arbeit eingesetzt[19].
Zu Beginn wird mit Hilfe dieser Implementierung ein digitales ChArUco-Board erzeugt, dessen Parameter sowie die Anzahl der Felder, Feldgrößen und Markergrößen individuell eingestellt werden können. Das generierte Bild kann anschließend ausgedruckt und als Referenzobjekt für die Kalibrierung genutzt werden. Auf diese Weise wird sichergestellt, dass eine konsistente geometrische Struktur vorliegt, die in den Aufnahmen der Kamera eindeutig erkannt werden kann.
Die eigentliche Kalibrierung erfolgt interaktiv und unabhängig vom Flippersystem. Nach dem Start des Prozesses wird das Live-Bild der Kamera angezeigt und der Benutzer kann durch Tasteneingaben gezielt Aufnahmen des Boards hinzufügen. Abbildung 6.3 zeigt die Kalibrierung mittels der ChArUco-Marker. Jede Aufnahme wird direkt analysiert: Die Marker werden erkannt, die Lage des Boards im Bild bestimmt und die Schachbrettecken interpoliert. Wenn ausreichend viele Eckpunkte identifiziert werden, wird die Aufnahme in den Kalibrierungsdatensatz übernommen. Auf diese Weise entstehen sukzessive mehrere

[18] `https://docs.opencv.org/3.4/df/d4a/tutorial_charuco_detection.html`, abgerufen am 21. September 2025.
[19] `https://github.com/MarcOnTheMoon`, abgerufen am 21. September 2025.

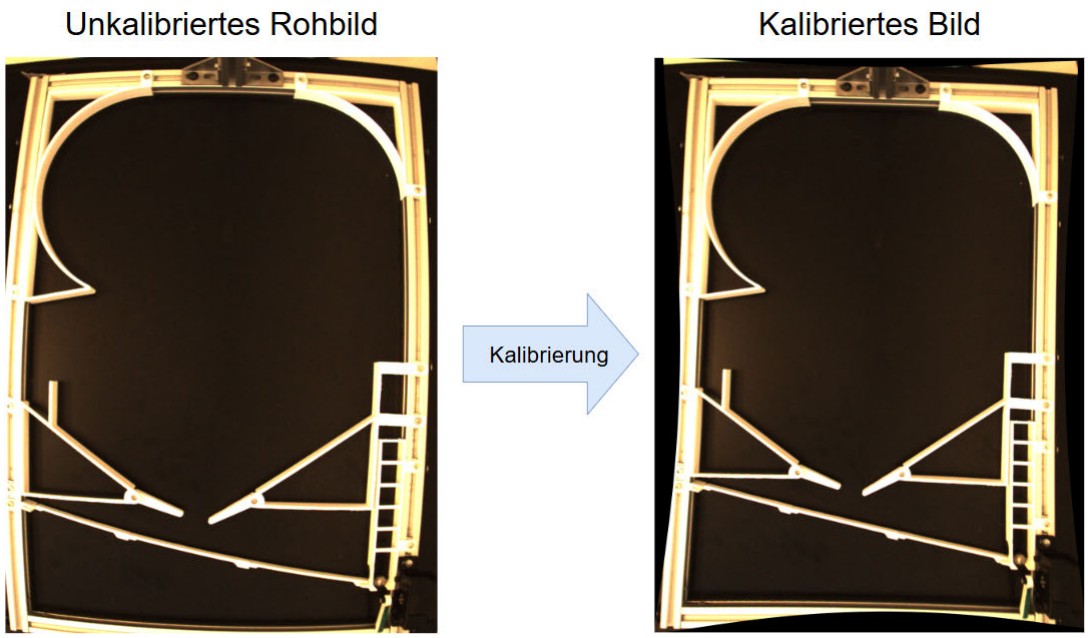


Abbildung 6.4: Kalibrierung der Rohbilder: Rohbild (links) und entzerrtes Bild (rechts)

Ansichten des Boards aus unterschiedlichen Perspektiven. Hierdurch wird die Genauigkeit der späteren Parameterberechnung verbessert.
Sobald genügend Ansichten gesammelt wurden, werden die eigentlichen Kalibrierungsparameter berechnet. Die Software ermittelt hierbei die intrinsische Kameramatrix sowie die Verzerrungskoeffizienten, welche die optischen Abbildungseigenschaften und Linsenfehler beschreiben. Die Ergebnisse werden in einer JSON-Datei gespeichert, sodass sie später wieder eingelesen und zur Entzerrung von Bildmaterial genutzt werden können. Ein optionaler Demonstrationsmodus erlaubt es darüber hinaus die Wirkung der Kalibrierung unmittelbar zu überprüfen, indem originale und entzerrte Bilder direkt verglichen werden.
Damit steht ein flexibles Werkzeug zur Verfügung, welches es ermöglicht eine Kamera systematisch, wie in Abbildung 6.4, zu kalibrieren und die gewonnenen Parameter in nachgelagerten Bildverarbeitungsprozessen einzusetzen.

6.2.2 Auswahl des Verfahrens für die Bildverarbeitung

Die Bildverarbeitung bildet das zentrale Bindeglied zwischen der physischen Spielumgebung und der künstlichen Intelligenz des Flippers. Ihre Aufgabe ist es relevante Informationen aus den Kamerabildern in Echtzeit zu extrahieren und aufzubereiten, sodass die KI fundierte Entscheidungen über das Auslösen der Flipper treffen kann.
Ein wesentlicher Bestandteil der geplanten Bildverarbeitung wird die Erfassung und Vorverarbeitung der Kamerabilder sein. Dazu zählen die Entzerrung der Bilder, die Rotation und die Fokussierung auf die relevanten Spielfeldbereiche. Durch diese Schritte sollen Verzerrungen durch die Kameralinse minimiert und nur jene Bildinformationen berück-

sichtigt werden, die für die Erkennung der Kugel und der Flipper relevant sind. Auf Basis dieser vorbereiteten Bilder wird die Kalibrierung der Flipperbereiche erfolgen. Ziel ist es die exakte Lage und Bewegungswinkel der Flipper zu erfassen, um später deren aktuelle Stellung zuverlässig bestimmen zu können.
Ein zentrales Element wird die Kugeldetektion und -verfolgung sein. Die Bildverarbeitung soll in der Lage sein die Kugel von den Flipperarmen und anderen Objekten auf dem Spielfeld zu unterscheiden. Hierzu sind Verfahren wie Bewegungsdetektion, Filterung zur Rauschunterdrückung und Konturenanalyse vorgesehen. Auf Basis der erkannten Kugelpositionen sollen Geschwindigkeit und Bewegungsrichtung berechnet werden, die als Eingabedaten für die KI dienen. Ergänzend soll die Bildverarbeitung bestimmte Spielereignisse erkennen, wie beispielsweise das Auftreten farblich markierter Bereiche, die als Signale für Aktionen im Spiel interpretiert werden können.
Schließlich müssen alle erfassten Informationen – Kugelposition, Flipperwinkel und erkannte Spielereignisse – in einer standardisierten Form aufbereitet werden, sodass sie der KI in Form des Zustandsraums zur Verfügung gestellt werden können. Die geplante modulare Umsetzung soll gewährleisten, dass die Verarbeitungsschritte in Echtzeit zusammenarbeiten und eine kontinuierliche Rückmeldung über den aktuellen Spielzustand liefern.

Nutzwertanalyse des Verfahrens

Um die aus den Bilddaten gewonnenen Informationen möglichst präzise und effizient zu extrahieren, ist die Wahl eines geeigneten Bildverarbeitungsverfahrens von zentraler Bedeutung. Unterschiedliche Ansätze bieten spezifische Vor- und Nachteile hinsichtlich der Erkennungsgenauigkeit, Laufzeit, Robustheit und des Implementierungsaufwands.
In der Literatur und Praxis sind hierbei im Wesentlichen drei Methoden zu unterscheiden: klassische (rein algorithmische) Verfahren, moderne KI-basierte Methoden und hybride Ansätze beider Bereiche.

Klassisches Verfahren

Die klassische Bildverarbeitung wie Kantendetektion (z. B. Canny-Algorithmus), Hough - Transformation zur Linien- bzw. Kreiserkennung oder adaptive Schwellenwertbildung wurden in zahlreichen Anwendungen eingesetzt, zum Beispiel zur Positionsbestimmung von Bauteilen in der industriellen Qualitätskontrolle oder zur einfachen Objektverfolgung auf kontrastreichen Hintergründen. Ihr Vorteil liegt in der oft hohen Ausführungsgeschwindigkeit und Nachvollziehbarkeit. Von Nachteil ist es jedoch, dass sie häufig störanfällig bei wechselnden Lichtbedingungen oder unregelmäßigen Hintergründen sind [MDP21].

KI-basierte Methoden

Solche Methoden, darunter insbesondere Deep-Learning-Ansätze wie Convolutional Neural Networks (CNN) oder die Architektur YOLO, sind mittlerweile Stand der Technik in

Kriterium	**Beschreibung**	**Gewichtung [%]**
Erkennungsgenauigkeit	Präzision bei der Bestimmung	30
Verarbeitungsgeschwindigkeit	Zeitbedarf, Bildanalyse	20
Robustheit gegenüber Störungen	Toleranz bei Lichtbedingungen	15
Implementierungsaufwand	Zeit für die Entwicklung	25
Skalierbarkeit	Eignung für weitere Projekte	10

Tabelle 6.1: Bewertungskriterien und Gewichtung

der Objekterkennung auch unter komplexen Bedingungen. Sie werden in aktuellen Forschungsvorhaben etwa zur Erkennung von Fahrbahnmarkierungen, in der medizinischen Bildanalyse oder bei der Ballverfolgung im Sport eingesetzt. Solche Verfahren zeichnen sich durch ihre hohe Flexibilität, Robustheit und Anpassungsfähigkeit an neue Szenarien aus, erfordern jedoch große Trainingsdatensätze und erhebliche Rechenressourcen[20].

Hybride Ansätze

Sie vereinen die Stärken beider Bereiche, indem beispielsweise ein neuronales Netz für eine grobe Lokalisierung der Objekte sorgt, ehe ein klassischer Algorithmus für die präzise Winkel- oder Schwerpunktberechnung verwendet wird. Dies bietet Vorteile in Bezug auf Genauigkeit und Geschwindigkeit und wird zunehmend in anspruchsvollen industriellen Systemen genutzt, etwa in der automatisierten Montagekontrolle, wo Präzision gefordert ist[21].

Für eine fundierte Entscheidung, welches Verfahren für die vorliegende Anwendung – das automatische Auslesen der Flipperarmwinkel und der Kugelposition in einem Flipper-Setup – am Besten geeignet ist, wird im Folgenden eine Nutzwertanalyse durchgeführt. Berücksichtigt werden dabei zentrale Kriterien wie Erkennungsgenauigkeit, Verarbeitungsgeschwindigkeit, Robustheit gegenüber Störungen, Implementierungsaufwand und Skalierbarkeit.

Zur Beurteilung der verschiedenen Ansätze für die Auswertung der Bilddaten bezüglich der Bestimmung der Flipperarmwinkel und der Kugelposition wurde eine Nutzwertanalyse durchgeführt.

Kriterien und Gewichtung

[20] `https://www.sciencedirect.com/science/article/abs/pii/S092523121830924X`, abgerufen am 30. September 2025.

[21] `https://arxiv.org/abs/1812.03506`, abgerufen am 30. September 2025.

Kriterium	**Klassisch**	**KI-basiert**	**Hybrid**
Erkennungsgenauigkeit	3	4	5
Verarbeitungsgeschwindigkeit	5	2	4
Robustheit ggü. Störungen	2	5	4
Implementierungsaufwand	4	1	1
Skalierbarkeit	4	5	4

Tabelle 6.2: Bewertung der Methoden (1: schlecht bis 5: exzellent)

Die Gewichtung der Kriterien orientiert sich an den spezifischen Anforderungen der Anwendung, nämlich der zuverlässigen Bestimmung der Flipperarmwinkel und der Kugelposition in einem Flipper-Setup.
Die **Erkennungsgenauigkeit** wurde mit 30 % am höchsten gewichtet, da die korrekte Bestimmung der Kugelposition und Flipperstellung die Grundlage für eine weiterführende Auswertung und Steuerung darstellt. Eine unzureichende Präzision würde die gesamte Analyse entwerten.
Mit 25 % wurde der **Implementierungsaufwand** ebenfalls hoch angesetzt. Da es sich um ein Forschungs- und Entwicklungsprojekt handelt, spielt die verfügbare Zeit zur Erstellung einer funktionierenden Lösung eine entscheidende Rolle. Methoden, die zwar leistungsfähig, aber nur mit erheblichem Programmier- und Trainingsaufwand umsetzbar sind, wurden daher kritisch bewertet.
Die **Verarbeitungsgeschwindigkeit** erhielt eine Gewichtung von 20 %. Gerade im Kontext eines Spiels ist eine möglichst verzögerungsfreie Bildauswertung notwendig, um Bewegungen in Echtzeit erfassen zu können.
Die **Robustheit gegenüber Störungen**, etwa wechselnden Lichtverhältnissen oder unruhigen Hintergründen, wurde mit 15 % berücksichtigt. Zwar ist dies wichtig, doch stehen für das geplante Setup kontrollierte Bedingungen zur Verfügung, weshalb Robustheit eine geringere Priorität als Genauigkeit und Geschwindigkeit einnimmt.
Schließlich wurde die **Skalierbarkeit** mit 10 % am niedrigsten gewichtet. Die Erweiterbarkeit auf andere Aufgabenfelder ist zwar wünschenswert, jedoch nicht unmittelbar für die Zielstellung entscheidend. Dieses Kriterium dient daher eher als sekundärer Faktor zur Differenzierung der Methoden.

Interpretation

Die Analyse zeigt, dass der Implementierungsaufwand bei der KI-basierten und insbesondere bei der hybriden Methode im Verhältnis zum erreichbaren Zusatznutzen sehr hoch ist. Unter Berücksichtigung aller Kriterien schneidet daher die klassische Bildverarbeitung am Besten ab. Sie bietet eine angemessene Genauigkeit und Geschwindigkeit bei deutlich geringerem Aufwand hinsichtlich Entwicklung und Ressourcen. Damit erscheint

Kriterium	**Klassisch**	**KI-basiert**	**Hybrid**
Erkennungsgenauigkeit (30%)	3	4	5
Verarbeitungsgeschwindigkeit (20%)	5	3	4
Robustheit (15%)	2	5	4
Implementierungsaufwand (25%)	5	1	1
Skalierbarkeit (10%)	4	5	4
Summe	**3,85**	**3,1**	**3,55**

Tabelle 6.3: Nutzwertanalyse: klassische, KI-basierte und hybride Bildverarbeitung

der klassische Ansatz für die vorliegende Aufgabenstellung als die wirtschaftlichste und praktikabelste Lösung.

6.2.3 Parameter der Kamera

Ein wesentlicher Parameter bei der Konfiguration der Kamera ist die Belichtungszeit (Exposure Time). Diese bestimmt, wie lange der Bildsensor Licht aufnimmt und hat damit direkten Einfluss auf die Bildqualität. Im vorliegenden Szenario wird eine Mindestbelichtungszeit von 10.000 μs (10 ms) gewählt. Der Grund hierfür liegt in der Netzfrequenz des deutschen Stromnetzes von 50 Hz, welche in Kombination mit LED-Beleuchtung zu sichtbarem Flackern führen kann.

6.2.4 Begründung der Belichtungszeit

Das heimische Stromnetz in Deutschland hat eine Netzfrequenz von

$$f_{Netz} = 50\,\text{Hz}, \tag{6.1}$$

$$T_{50} = \frac{1}{f_{Netz}} = \frac{1}{50} = 0{,}02\,\text{s} = 20\,\text{ms}. \tag{6.2}$$

Viele LED-Leuchten verwenden eine Gleichrichtung mit anschließender Vollwellenrectifikation, wodurch sich die Frequenz des Lichtflackerns verdoppelt:

$$f_{LED} = 2 \cdot f_{Netz} = 100\,\text{Hz}, \tag{6.3}$$

$$T_{100} = \frac{1}{100} = 0{,}01\,\text{s} = 10\,\text{ms} = 10.000\,\mu\text{s}. \tag{6.4}$$

Damit die Kamera eine vollständige Periode des LED-Flackerns erfasst und die Helligkeitsschwankungen durch Mittelung geglättet werden, ist eine Belichtungszeit von mindestens 10 ms notwendig. Kürzere Belichtungszeiten würden nur einen Ausschnitt der Periode abdecken und somit sichtbares Flackern in den Aufnahmen erzeugen.

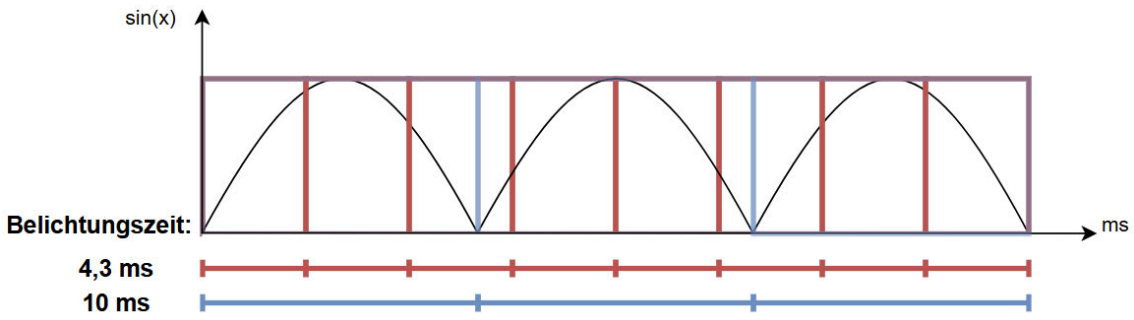


Abbildung 6.5: Schematische Darstellung des LED-Flackerns als gleichgerichteter Sinusfunktion und eines Belichtungsfensters von 10 ms und 4,3 ms.

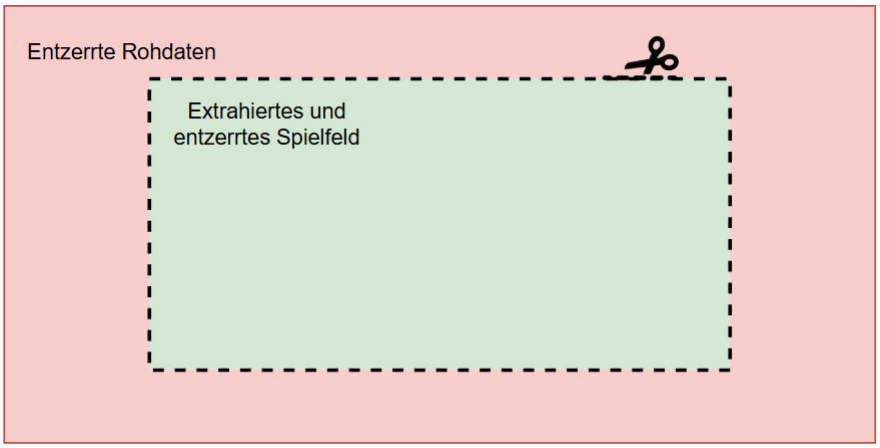


Abbildung 6.6: Extrahierung des Spielfelds

Die Abbildung 6.5 zeigt schematisch das periodische Signal der LED-Helligkeit als gleichegerichtete Sinuswelle und ein Belichtungsfenster von 10 ms und 4,3 ms.

6.2.5 Extraktion des Spielfelds

Um den Agenten mit dem physischen Demonstrator zu trainieren, muss gemäß der Anforderung **BV-F3** eine störungs- und verzerrungsfreie Umgebung herrschen. Wenn sich die physische Umgebung mit der virtuellen Umgebung in allen Aspekten ähnelt, ist das Training effizienter und weniger störanfällig als mit den von der Kamera gegebenen und verzerrten Rohdaten. Abbildung 6.6 zeigt das Vorgehen, welches mit dem erhaltenen Rohbild durchgeführt werden soll. Dies führt dazu, dass sich das Datenvolumen reduziert und somit die Geschwindigkeit der Berechnungen leicht angehoben wird. Zudem entfernt die Extrahierung überflüssige und störende Strukturen, welche bei der Bildverarbeitung Probleme erzeugen können.

Um das Bild entsprechend zu verarbeiten, wurde eine Klasse entwickelt, welche im Kalibrierungsprozess eingesetzt wird, das Spielfeld ausschneidet und der folgenden Verarbei-

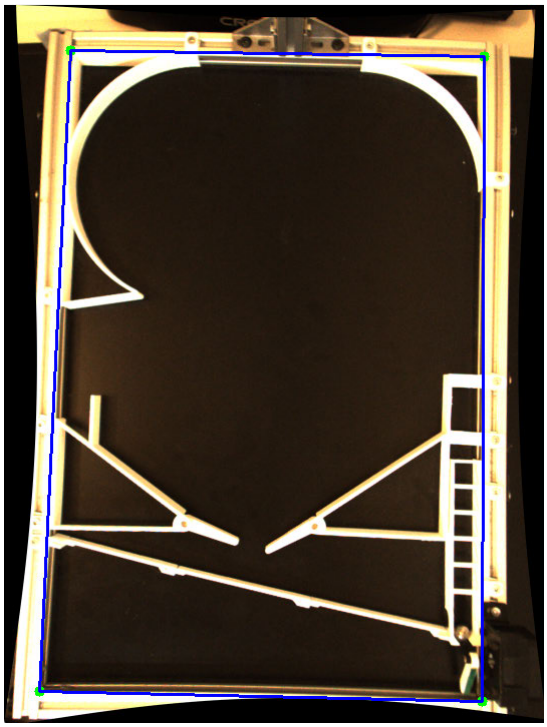


Abbildung 6.7: Interaktive Auswahl der Spielfeld-Ecken mithilfe des FieldCroppers

tung übergibt.

Die Klasse `FieldCropper` erlaubt es, den Ausschnitt des Spielfelds interaktiv zu definieren und für zukünftige Bildverarbeitungsprozesse bereitzustellen.

Der Ansatz basiert auf einer benutzergeführten Initialisierung. Zunächst wird ein Eingabebild geladen und auf eine handlichere Größe skaliert, um die nachfolgende Interaktion zu erleichtern. Anschließend können die vier Eckpunkte des Spielfelds durch Mausklicks auf das angezeigte Bild verschoben werden. Dabei werden die Koordinaten in der Originalauflösung des Bildes gespeichert, sodass keine Informationsverluste durch die Skalierung entstehen. Standardmäßig ist ein rechteckiger Bereich vordefiniert, der als Ausgangspunkt dient und vom Benutzer nachjustiert werden kann.

Die Benutzerinteraktion wird über eine Callback-Funktion realisiert, die Mausklicks registriert und die Position des nächstgelegenen Punkts dynamisch anpasst. Auf diese Weise lassen sich kleine Korrekturen schnell und intuitiv vornehmen. Nach Abschluss der Auswahl wird die Eingabe durch Drücken einer Taste bestätigt. Der gewählte Bildausschnitt wird in einer JSON-Datei gespeichert, sodass er beim nächsten Start des Programms automatisch geladen und erneut verwendet werden kann. Dies ermöglicht eine einmalige Kalibrierung des Spielfelds, ohne dass die Ecken bei jeder Ausführung neu gesetzt werden müssen.

Die Speicherung der Koordinaten erfolgt im JSON-Format, sodass eine einfache Weitergabe und Wiederverwendung gewährleistet wird. Sollte die Datei beschädigt oder nicht vorhanden sein, wird der Nutzer aufgefordert, die Punkte erneut zu setzen. Damit ist eine

robuste Handhabung verschiedener Anwendungsfälle sichergestellt.

6.2.6 Perspektivische Transformation des Spielfeldes

Nachdem die vier Spielfeldecken im Originalbild durch den Nutzer definiert wurden, erfolgt eine perspektivische Transformation, um den Bereich in eine normierte rechteckige Form zu überführen. Dies wird durch die Funktion `field_transform()` umgesetzt.
Die Funktion erhält als Eingabe das Kamerabild, die ausgewählten Punkte sowie die gewünschte Zielbreite und Zielhöhe des transformierten Bildes. Zunächst werden die gewählten Eckpunkte $\mathbf{p}_i = (x_i, y_i)$ als Quellpunkte (*src_pts*) definiert. Dies sind die Koordinaten der Spielfeldecken im verzerrten Originalbild. Anschließend werden die Zielpunkte (*dst_pts*) festgelegt, die ein Rechteck mit den Koordinaten

$$\mathbf{p}'_1 = (0,0), \quad \mathbf{p}'_2 = (w-1,0), \quad \mathbf{p}'_3 = (w-1,h-1), \quad \mathbf{p}'_4 = (0,h-1) \tag{6.5}$$

beschreiben. Hierbei definiert w die Breite und h die Höhe des normierten Ausgabebildes. Mit diesen Punktpaaren wird eine Matrix H der Größe 3×3 berechnet, die die projektive Abbildung beschreibt:

$$\lambda \begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = H \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}, \tag{6.6}$$

wobei (x, y) die Originalkoordinaten und (x', y') die transformierten Koordinaten sind. Der Faktor λ sorgt für die Normierung in homogenen Koordinaten.
Die Berechnung der Matrix H erfolgt in OpenCV durch den Aufruf

$$H = \texttt{cv2.getPerspectiveTransform(src\_pts, dst\_pts)}.$$

Anschließend wird das gesamte Bild mit der nachfolgenden Formel transformiert:

$$I'(x',y') = I(H^{-1}(x',y')). \tag{6.7}$$

In OpenCV übernimmt dies die Funktion `cv2.warpPerspective()`, die das Bild I mit der Matrix H in das Ausgabebild I' überführt. Das Ergebnis ist ein entzerrtes, rechteckiges Spielfeld, das unabhängig von Kamerawinkel oder Verzerrung konsistent weiterverarbeitet werden kann.
Wie in Abbildung 6.8 zu sehen, kann somit nach erfolgreicher Transformation ein Koordinatensystem geschaffen werden, welches Anwendung sowohl in der physischen als auch in der virtuellen Umgebung findet.

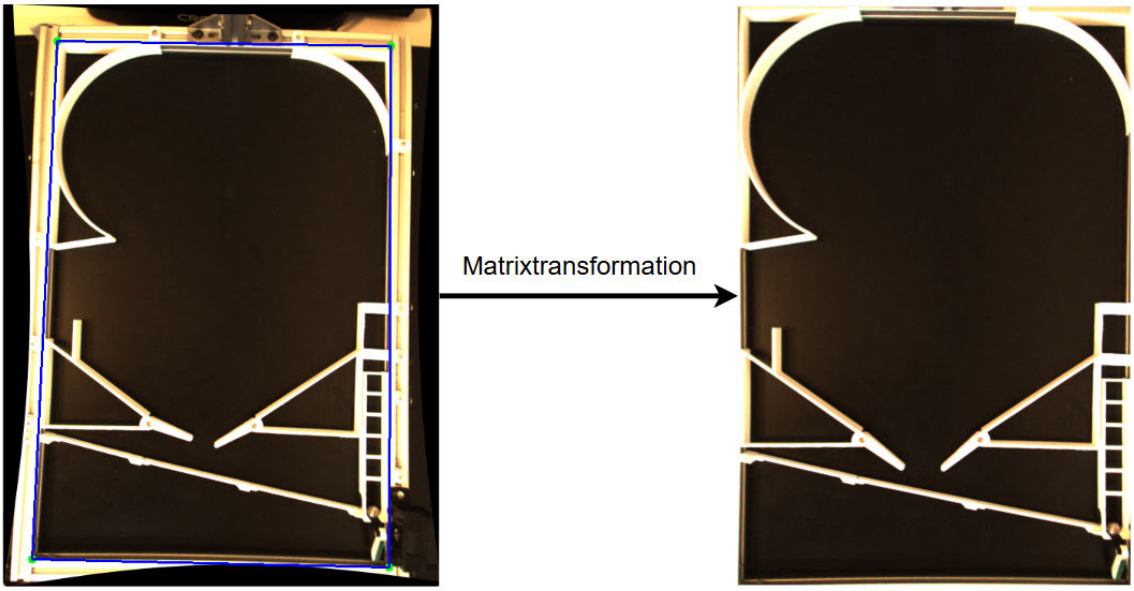


Abbildung 6.8: Umwandlung des Bildes

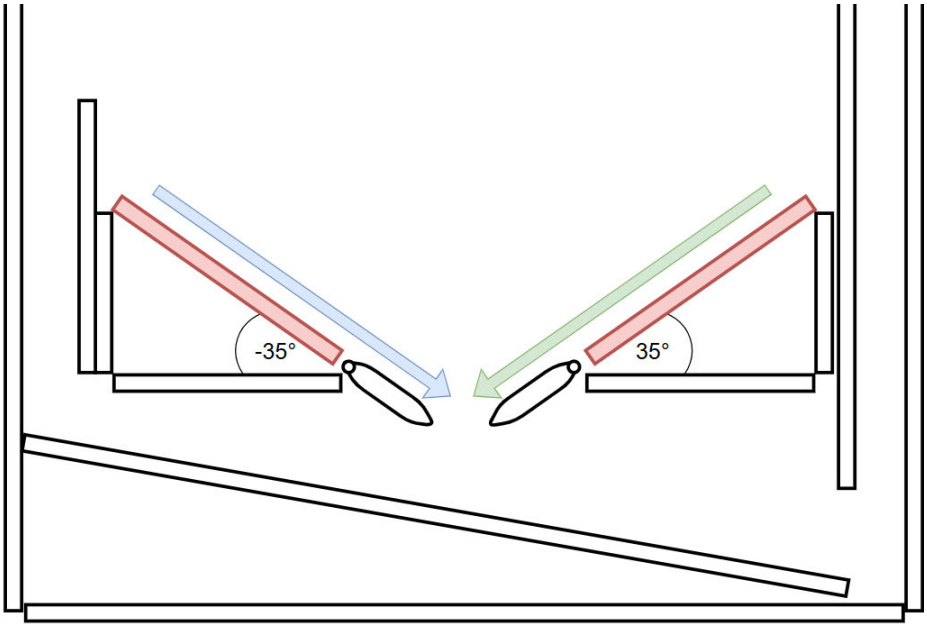


Abbildung 6.9: Skizze des unteren Flipperspielfelds

6.2.7 Erfassung der Winkel der Flipperarme für Kalibrierungsmaßnahmen

Entsprechend der Beschreibung in der Anforderung **BV-F4** ist es relevant, dass die Ruhelage der Flipperarme jederzeit korrigiert werden kann. Hierzu müssen die Winkel erkannt und in einem zeitlichen Rahmen regelmäßig korrigiert werden. Gemäß der Anforderung PD-NF1 müssen die Flipperarme in einer Flucht zu den Inlanes stehen.

Abbildung 6.9 zeigt die beschriebene Flucht an den rot markierten Inlanes. Der grüne Pfeil soll den rechten Flipperarme mit 35° und der blaue Pfeil den linken Flipperarme mit einem Lagewinkel von -35° relativ zur Waagerechten beschreiben.

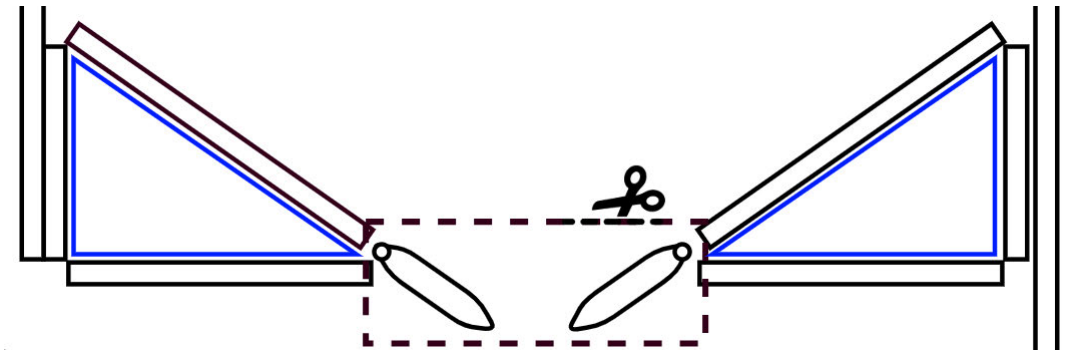


Abbildung 6.10: Erkennung des Bereichs der Flipperarme

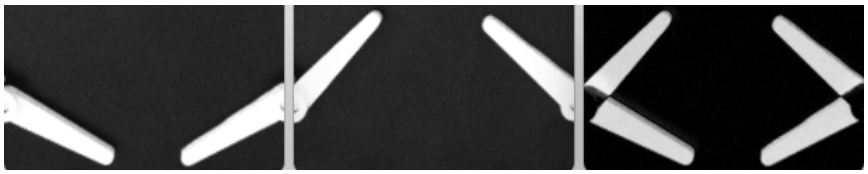


Abbildung 6.11: Differenzbild: Flipper inaktiv (links), Flipper aktiv (mittig) und Differenzbild (rechts)

Extraktion des Flipperbereichs

Um grundlegend die Flipperarme mittels Bildverarbeitung zu finden und die Winkel herauszufinden, muss die Position dieser bestimmt werden. Da die Flipperarme statisch in ihrem Drehpunkt sind, ist es einfacher den Bereich zu extrahieren.

Die Methode `calibrate_flipper_area()` führt diesen Kalibrierungsprozess durch. Zunächst wird das Spielfeld mit Hilfe der zuvor beschriebenen perspektivischen Transformation in eine normierte Darstellung überführt, sodass die Flipper unabhängig vom Kamerawinkel immer im gleichen Bildausschnitt erscheinen.
Im nächsten Schritt werden zwei Bilder aufgenommen: eines mit abgesenkten Flippern I_{down} und eines mit angehobenen Flippern I_{up}. Beide Aufnahmen werden in Graustufenbilder G_{down} und G_{up} konvertiert:

$$G_{\text{down}} = \texttt{cv2.cvtColor}(I_{\text{down}}), \quad G_{\text{up}} = \texttt{cv2.cvtColor}(I_{\text{up}}).$$

Die Differenz dieser beiden Graustufenbilder liefert, wie in Abbildung 6.10 gezeigt, jene Bildbereiche, die sich durch die Flipperbewegung verändert haben.

$$D(x,y) = | G_{\text{down}}(x,y) - G_{\text{up}}(x,y) | \tag{6.8}$$

Um diese Bereiche klarer zu isolieren, wird ein Schwellwertverfahren angewandt:

$$T(x,y) = \begin{cases} 255, & \text{falls } D(x,y) > \tau \\ 0, & \text{sonst} \end{cases} \tag{6.9}$$

Der Schwellenwert $\tau = 150$ hat sich als robust bewiesen hat. Dadurch entsteht ein Binärbild, in dem die Flipperbewegung als weiße Regionen hervorgehoben ist.
Anschließend werden mithilfe der Funktion `cv2.findContours` die zusammenhängenden Flächen (Konturen) der Differenzregion bestimmt. Über alle gefundenen Konturen wird ein minimales Begrenzungsrechteck berechnet, das den gesamten Bewegungsbereich der Flipper abdeckt:

$$x_{\min} = \min_i(x_i), \quad y_{\min} = \min_i(y_i), \quad x_{\max} = \max_i(x_i + w_i), \quad y_{\max} = \max_i(y_i + h_i). \tag{6.10}$$

Um sicherzustellen, dass der Bereich vollständig erfasst wird, wird zusätzlich eine Toleranz t addiert, sodass der finale Flipperbereich durch die Koordinaten beschrieben ist:

$$(x_{\min}, x_{\max}, y_{\min} - t, y_{\max} + t) \tag{6.11}$$

Das Ergebnis der Extraktion des Flipperarmbereichs ist somit ein rechteckiger Bildausschnitt, der den gesamten Bewegungsraum der Flipper enthält und für die nachfolgenden Bildverarbeitungsschritte, wie die Erkennung von Kugel-Flipper-Interaktionen, genutzt werden kann.

Erfassung der Flipperwinkel

Für die zuverlässige Erfassung der Winkel der Flipperarme wurde ein Algorithmus zur automatisierten Winkelerkennung implementiert. Ziel dieser Entwicklung war es die Neigung der beiden Flipperarme aus Bilddaten zu bestimmen.
Die Implementierung basiert auf einer Kombination von Bildvorverarbeitung, Kantendetektion, Konturanalyse und geometrischer Approximation. Nachfolgend werden die einzelnen Schritte der Funktion `detect_flipper_arms` im Detail erläutert.

Perspektivkorrektur und Bildvorbereitung

Wie in Abbildung 6.12 gezeigt, wird der extrahierte Flipperbereich, in dem sich die beiden Flipperarme bewegen, zur weiteren Verarbeitung genutzt. Um die Analyse robuster zu gestalten, wird das Bild entlang seiner vertikalen Mittelachse in eine linke und eine rech-

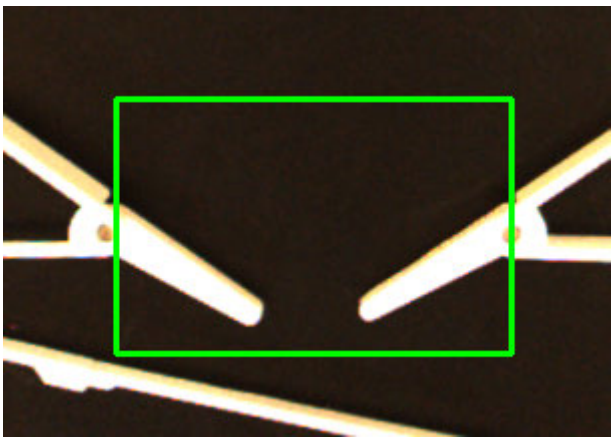


Abbildung 6.12: Aufnahme des ermittelten Flipperbereichs

te Hälfte aufgeteilt. So kann jeder Flipperarm mit der Funktion `process_half` separat verarbeitet und analysiert werden ohne dass es zu Komplikationen kommt.

Rauschreduktion

Jede Bildhälfte wird zunächst mit einem Gaußschen Weichzeichner geglättet:

```
cv2.GaussianBlur(image_part, (5, 5), 0)
```

Dieser Schritt dient dazu, hochfrequentes Bildrauschen und kleine Störpixel zu unterdrücken. Ohne diese Glättung könnte die nachfolgende Kantendetektion fälschlicherweise viele irrelevante Kanten erkennen, was die Konturerkennung erschweren würde. Konturen die auch nur minimal abweichende Werte aufweisen, können zu großen Winkeländerungen führen. Der Kernel von (5×5) hat sich dabei als guter Kompromiss zwischen Glättung und Erhalt von relevanten Details erwiesen.

Kantendetektion

Nach der Glättung wird mit dem Canny-Algorithmus nach Kanten gesucht:

```
cv2.Canny(blurred, 50, 170, apertureSize=3)
```

Der Canny-Operator hebt Stellen mit starken Intensitätsunterschieden im Bild hervor und markiert diese als Kanten. Die Schwellwerte 50 und 170 haben sich durch mehrfaches Testen als beste Einstellung erwiesen.

Konturenerkennung

Die im Bild vorhandenen Kanten werden anschließend zu Konturen zusammengefasst:

```
cv2.findContours(part_edges,cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
```

Mit dem Parameter `cv2.RETR_EXTERNAL` wird dabei festgelegt, dass ausschließlich die äußersten Konturen erfasst werden und innere Strukturen oder Löcher unberücksichtigt

bleiben. Durch `cv2.CHAIN_APPROX_SIMPLE` werden die Konturen schließlich in komprimierter Form gespeichert, sodass nur die wesentlichen Punkte wie Eckpunkte erhalten bleiben und redundante Zwischenpunkte weggelassen werden. Da der Flipperarm in jeder Bildhälfte typischerweise die größte zusammenhängende Struktur ist, wird die Kontur mit der größten Fläche als Kandidat ausgewählt:

```
largest_contour = max(contours, key=cv2.contourArea)
```

Auf diese Weise werden kleinere, unerwünschte Objekte oder zufällige Kanten ausgeschlossen.

Polygonapproximation

Die größte Kontur ist oft unregelmäßig und enthält viele Punkte, die für die Berechnung der Orientierung nicht notwendig sind. Daher wird sie durch folgende Funktion vereinfacht:

```
cv2.approxPolyDP(largest_contour, epsilon, True)
```

Diese Funktion reduziert die Anzahl der Stützpunkte der Kontur, ohne ihre Form wesentlich zu verändern. Dadurch entsteht ein stabiles Polygon, welches weniger empfindlich auf Bildrauschen oder kleine Unregelmäßigkeiten reagiert. Der Parameter `epsilon` legt die Toleranz der Abweichung fest und ist hier proportional zur Länge der Kontur gewählt.

Geometrische Approximation mit Minimalrechteck

Um die Orientierung des Flipperarms zu bestimmen, wird um das approximierte Polygon ein Minimalrechteck gelegt:

```
rect = cv2.minAreaRect(approx)
box = cv2.boxPoints(rect)
```

Das Rechteck beschreibt die kleinste Fläche, die die gesamte Kontur umschließt. Aus der Rückgabe der Funktion erhält man neben der Position und den Kantenlängen auch einen Rotationswinkel (vgl. Abbildung 6.13).

Visualisierung

Zur Kontrolle während der Entwicklung werden sowohl die gefundene Kontur als auch das Minimalrechteck in die jeweilige Bildhälfte eingezeichnet:

Ausgabe der Ergebnisse

Zum Abschluss gibt die Funktion die berechneten Winkel der linken und rechten Flipperarme zurück. Diese Werte können anschließend zur Kalibrierung gegen Schrittverluste genutzt werden.

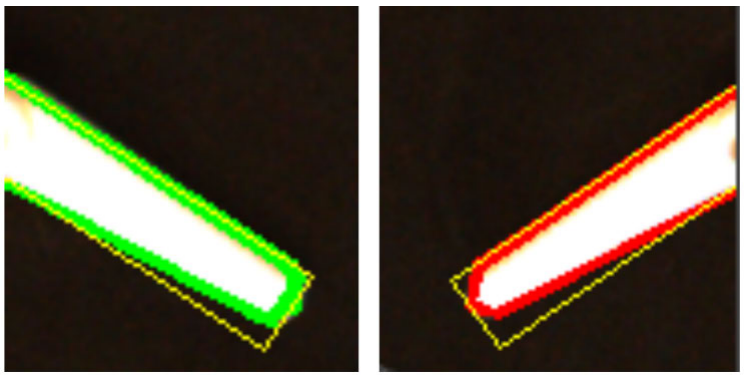


Abbildung 6.13: Ausgabe wie die Winkel der Flipperarme erkannt werden

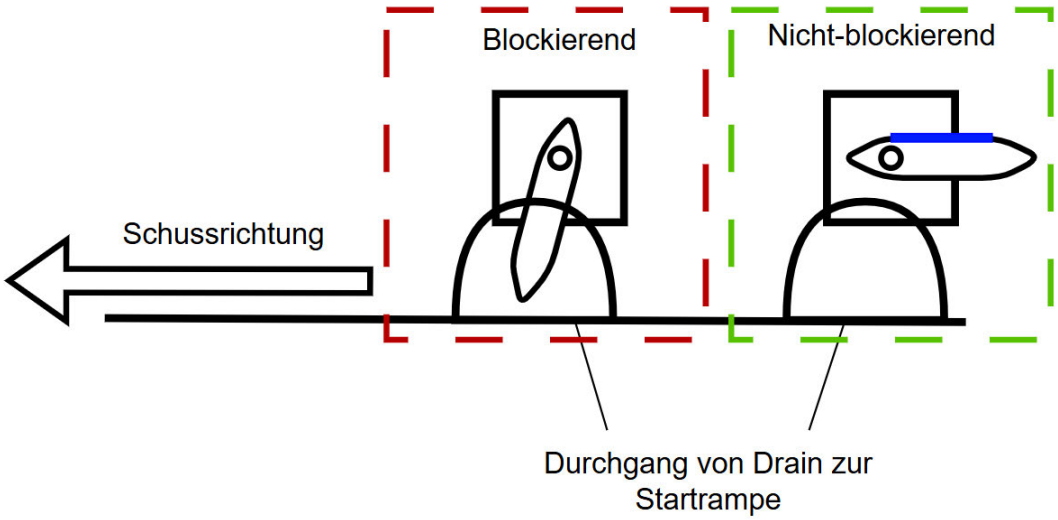


Abbildung 6.14: Seitenansicht der beiden möglichen Szenarien des Startarms

6.2.8 Korrektur des Startarms

Um die Anforderung **BV-F2** erfüllen zu können muss erkannt werden, wenn der Startarm nicht an einer gewünschten Position steht. Wie bereits in BV-F2 beschrieben, kann der Startarm die erfolgreiche Zurückführung der Kugel in die Startrampe verhindern, indem er den Durchgang vom Drain zur Startrampe versperrt. Dies kann nur der Fall sein, wenn der Startarm eine nach unten gerichtete Neigung aufweist.

Abbildung 6.14 zeigt die beiden Szenarien, die auftreten können. Links in rot markiert zeigt einen Fehlerfall, der gegebenenfalls zur Störung des Systems führen kann. Rechts in grün ist in der Abbildung der erstrebenswerte Fall, bei dem der Durchgang vom Drain zur Startrampe nicht versperrt ist, dargestellt.

Um dem blockierenden Zustand entgegen zu wirken muss der Startarm solange in kleinen Schritten gedreht werden bis eine akzeptable Position erreicht wird. Dies soll mit einer Farberkennung geschehen. Auf die gegen den Uhrzeigersinn gewandten Seite des Start-

arms wird eine farbliche Markierung gesetzt. Diese Markierung ist für die Kamera nur sichtbar, wenn der Startarm sich in einer nicht-blockierenden Position befindet. In Abbildung 6.14 wird die Markierung mit blauer Farbe repräsentiert.

Um dies automatisiert zu überprüfen, wird die Funktion `detect_green_in_bottom_right()` eingesetzt. Ziel dieser Funktion ist es den Startarm so lange zu drehen bis er sich in einer Position befindet, in der die Rampe frei ist. Erreicht wird dies durch die Erkennung einer farblich markierten Stelle am Startarm, die als visuelles Referenzsignal dient.
Der Prozess verläuft in mehreren Schritten:

1. Normierung und Unterteilung des Spielfeldes
Das aktuelle Kamerabild wird zunächst mit der perspektivischen Transformation in ein normiertes Koordinatensystem überführt. Anschließend wird das Bild in $r = 5$ Reihen und $c = 10$ Spalten unterteilt. Hierbei können überflüssige Bereiche, die nicht im Interesse der Detektion liegen, entfernt werden. Solange der Demonstrator in seinen Maßen nicht umgebaut wird, ändert sich an der Dimensionierung des Bereichs des Startarms ebenfalls nichts. Jeder Bildbereich hat also die Dimensionen

$$h_{\text{region}} = \frac{H}{r}, \quad w_{\text{region}} = \frac{W}{c}, \tag{6.12}$$

wobei H die Höhe und W die Breite des transformierten Bildes sind. Der relevante Bereich für den Startarm befindet sich in der rechten unteren Ecke des Spielfeldes.

2. Farbkonvertierung und Segmentierung
Der ausgewählte Bildbereich wird in den HSV-Farbraum überführt, um die Farberkennung robuster gegenüber Helligkeitsschwankungen zu machen. Anschließend wird eine Maske gebildet, die Pixel im Grünbereich isoliert:

3. Detektion und Rückmeldung
Befinden sich Pixel im Maskenbild ($M(x,y) = 1$), gilt die grüne Markierung als gefunden. In diesem Fall wird der erkannte Bereich im Originalbild durch ein Rechteck markiert und der Vorgang beendet. Falls keine grünen Pixel gefunden werden, wird der Startarm mithilfe der Steuerung um 10 Schritte mit je 1,8° weiterbewegt und der Vorgang wiederholt sich, bis die Markierung sichtbar ist.

4. Ergebnis
Die Funktion durchläuft so lange eine Schleife bis die grüne Markierung, wie in Abbildung 6.15 zu sehen, gefunden wurde. Dadurch steht der Bewegungszustand des Startarms automatisch zur Verfügung und kann für die fortlaufende Spielinitialisierung genutzt werden.

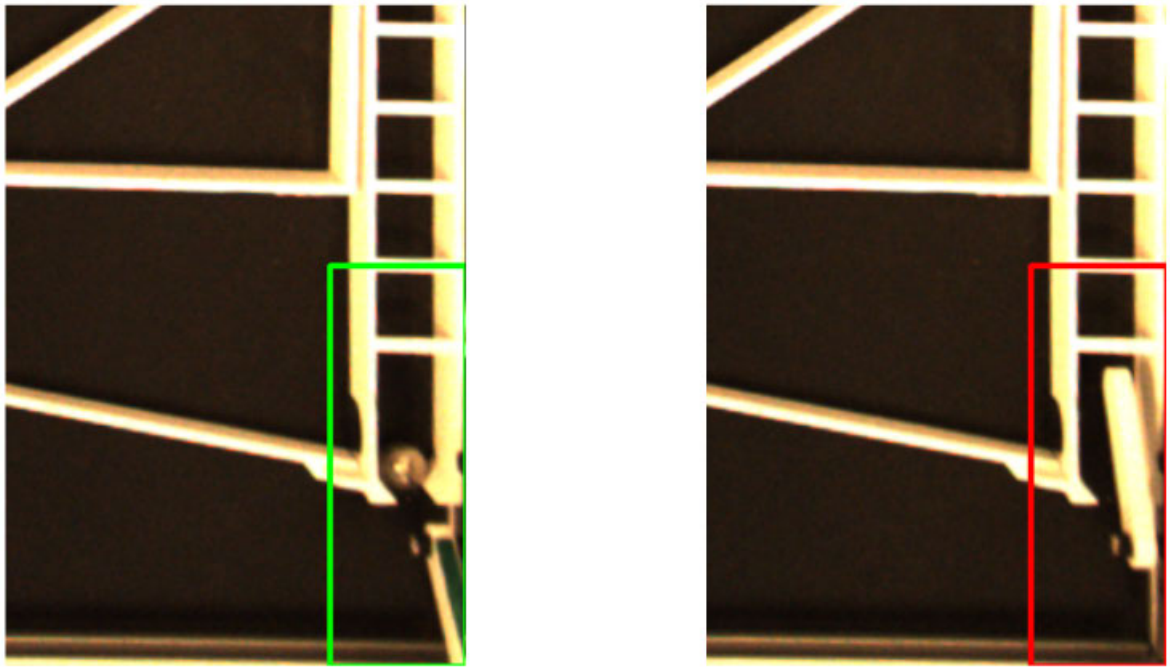


Abbildung 6.15: Erkennung der grünen Markierung am Startarm

$$M(x,y) = \begin{cases} 1, & \text{falls } (H,S,V) \in [45,75,75] \text{ bis } [85,255,115] \\ 0, & \text{sonst} \end{cases} \tag{6.13}$$

Die Ablauf der Kalibrierung des Startarms wird in der Abbildung 6.16 in einer vereinfachten Form beschrieben.

6.2.9 Detektion der Flipperkugel

Ein zentraler Bestandteil der Bildverarbeitung besteht in der zuverlässigen Detektion der Kugel auf dem Spielfeld. Diese Aufgabe ist besonders herausfordernd, da die Kugel sehr schnell über das Spielfeld rollt, gleichzeitig jedoch auch andere bewegliche Objekte wie die Flipperarme vorhanden sind. Um eine stabile Erkennung zu gewährleisten wird ein Verfahren eingesetzt das auf einer Differenzbildanalyse mit Hintergrundmodellierung basiert. Die hierfür entwickelte Methode `t_detect_flipper_ball()` kombiniert eine zeitliche Glättung des Hintergrunds mit der Konturanalyse von Bewegungsbereichen. Diese Funktion wird als Thread gestartet und sorgt so dafür, dass die Position der Kugel schnell genug zur Verfügung stehen kann.

Vorbereitung und Normierung

Das aktuelle Kamerabild wird zunächst durch eine perspektivische Transformation in ein normiertes Koordinatensystem überführt, sodass unabhängig von Kameraperspektive und

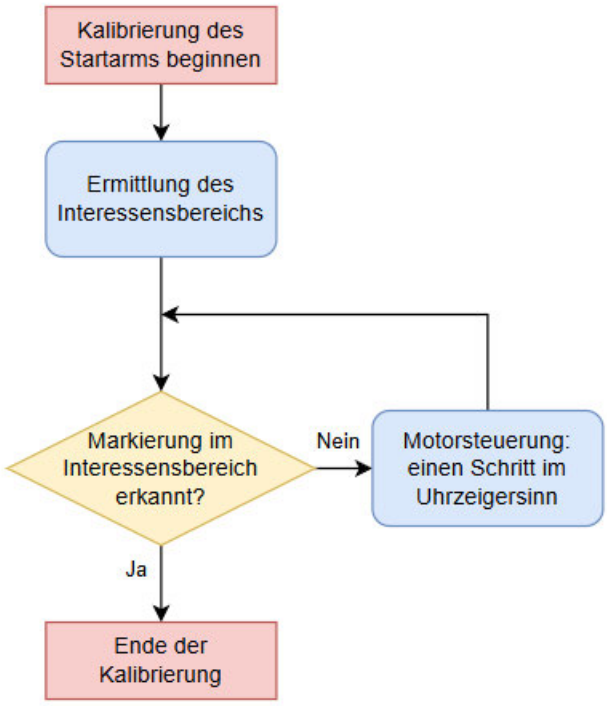


Abbildung 6.16: Vereinfachtes Ablaufdiagramm der Startarmkalibrierung

Verzerrung immer dieselben Spielfeldkoordinaten betrachtet werden. Anschließend wird das Bild in Graustufen konvertiert und mit einem Gauß-Filter geglättet, um lokale Intensitätsschwankungen und Rauschen zu reduzieren.

Bewegungserkennung durch ein gleitendes Hintergrundmodell

Um bewegte Objekte vom statischen Hintergrund zu trennen, wird ein adaptives Hintergrundmodell $B(x,y)$ geführt. Dieses Modell wird kontinuierlich mit einem gewichteten Durchschnitt aus den eingehenden Bildern aktualisiert:

$$B_t(x,y) = (1-\alpha) \cdot B_{t-1}(x,y) + \alpha \cdot I_t(x,y) \tag{6.14}$$

Hierbei ist $I_t(x,y)$ das aktuelle Graustufenbild und α ist ein Glättungsfaktor der mit dem Wert 0,5 ausgewählt wurde. $\alpha = 0{,}5$ hat sich bei mehreren Versuchen für die Stabilisierung als robust und wirkungsvoll bewiesen.

Das Differenzbild wird anschließend mit folgender Formel berechnet:

$$D(x,y) = | I_t(x,y) - B_t(x,y) | \tag{6.15}$$

Hierdurch werden ausschließlich die Bildbereiche, die sich im Vergleich zum stabilisierten Hintergrund verändern, hervorgehoben.

Schwellenwert und Rauschunterdrückung

Um die Kugelbewegung klar vom Hintergrund abzugrenzen, wird das Differenzbild durch einen Schwellwert in ein Binärbild überführt. Dabei gilt:

$$M(x,y) = \begin{cases} 1, & D(x,y) > \tau, \\ 0, & \text{sonst} \end{cases} \tag{6.16}$$

$\tau = 25$ wurde als Grenzwert gewählt. Durch anschließende morphologische Operationen (Öffnen und Schließen mit einem elliptischen Kern) werden kleine Rauschpixel entfernt und die Konturen der Kugel geglättet.

Konturerkennung und Schwerpunktberechnung

Die resultierende Binärmaske wird mit `cv2.findContours` verarbeitet, sodass die Konturen der bewegten Objekte im Bild verfügbar sind. Um Bildartefakte oder sehr kleine Bewegungen auszuschließen, wird nur mit Konturen gearbeitet, deren Fläche A im Bereich $10 < A < 1000$ Pixel liegt.

Für jede gültige Kontur werden die geometrischen Momente M_{pq} berechnet:

$$M_{pq} = \sum_x \sum_y x^p y^q f(x,y). \tag{6.17}$$

Daraus lässt sich der Schwerpunkt (Zentrum der Kugel) bestimmen:

$$c_x = \frac{M_{10}}{M_{00}}, \quad c_y = \frac{M_{01}}{M_{00}}.$$

Ausschlussbereiche zur Vermeidung von Fehlmessungen

Ein wesentliches Problem bei der Bewegungserkennung besteht in den Flipperarmen. Diese bewegen sich sehr schnell und erzeugen dadurch starke Veränderungen im Differenzbild. Ohne weitere Maßnahmen würden diese als Kugel interpretiert werden und die Erkennung massiv stören.

Um dies zu verhindern wird ein vordefinierter Bildbereich $R = [x_{\min}, x_{\max}] \times [y_{\min}, y_{\max}]$, der die Flipperarme umfasst, von der Analyse ausgeschlossen. Konkret bedeutet dies, dass Konturen, deren Schwerpunkt innerhalb dieses Ausschlussrechtecks liegt, ignoriert werden. Dadurch wird sichergestellt, dass ausschließlich die tatsächliche Kugelbewegung erkannt wird, selbst wenn gleichzeitig die Flipperarme bewegt werden.

Visualisierung

Für Debugging und Analyse werden die erkannten Konturen grün und die berechnete Kugelposition durch einen roten Punkt im Bild hervorgehoben. Somit ist visuell nachvollzieh-

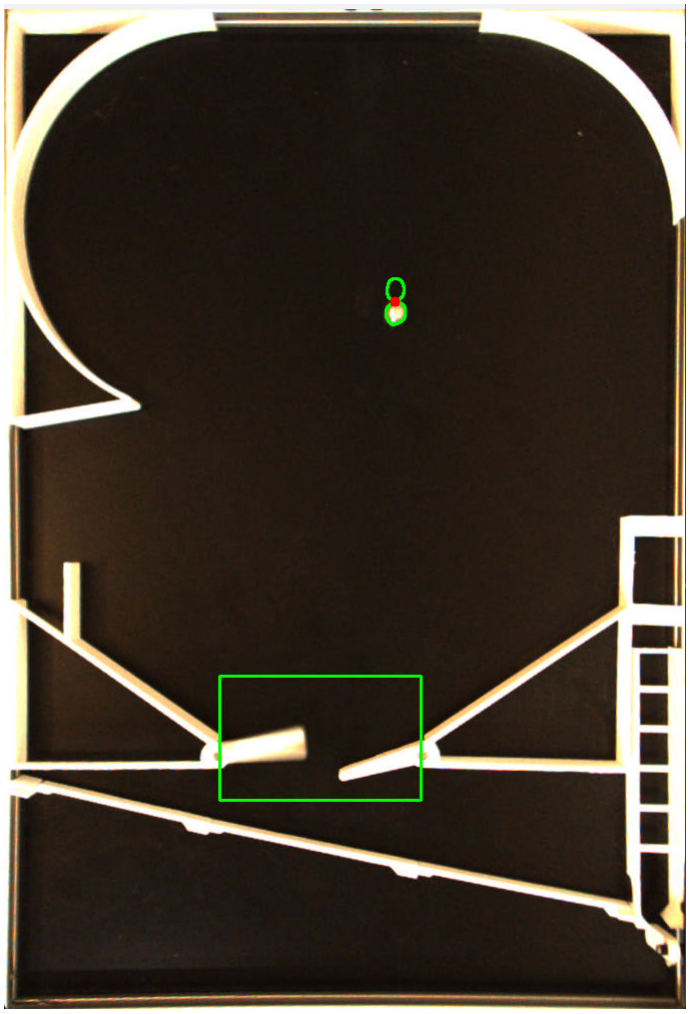


Abbildung 6.17: Detektion der Kugel mit markiertem Bereich der Flipperarme

bar, ob die Ausschlussbereiche korrekt greifen und die Kugelerkennung stabil funktioniert. Abbildung 6.17 zeigt die Markierungen der Kugel.
Durch die Kombination von Hintergrundmodellierung, differenzbildbasierter Bewegungserkennung, Schwerpunktberechnung mittels Momenten sowie gezielten Ausschlussbereichen konnte eine robuste Kugelerkennung realisiert werden, die auch bei parallelen Flipperbewegungen zuverlässig arbeitet.

6.2.10 Zeitvorgaben der physischen Umgebung

Aufgrund der technischen Beschränkungen ist es nicht möglich dem Agenten nach Konvertierung auf die physische Umgebung die höchste Bildrate zu übergebe, welche die Kamera liefern kann. Die Berechnungszeit der Ermittlung des Zustandsraums bedarf einer gewissen Zeit, die während der Entwicklung dokumentiert wurde. Wichtig ist hierbei ein stabiles Zeitmanagement vorweisen zu können um dem Agenten in der virtuellen Umgebung eine feste Zustandsperiode vorgeben zu können. Ziel ist es im Rahmen der zeitlichen

Bildbeschaffung - get_image()	16,895	ms
Kugelposition - t_detect_flipper_ball()	8,996	ms
Aktionszeit des Agenten	55,103	ms

Tabelle 6.4: Gemessene Zeiten der wichtigsten Sequenzen

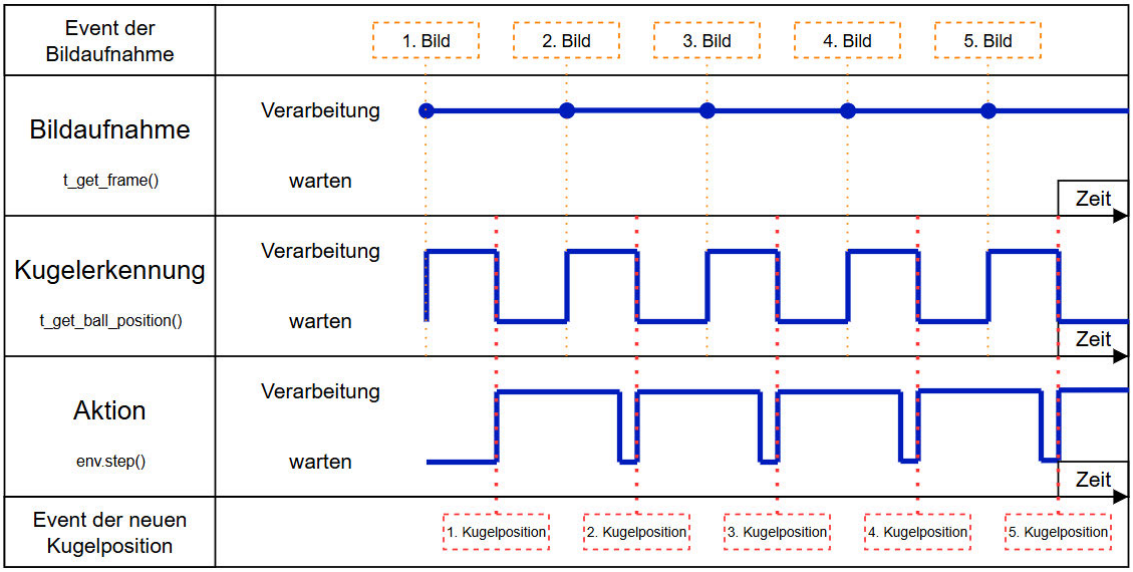


Abbildung 6.18: Vereinfachter zeitlicher Ablauf einer Zustandsverarbeitung

Planung die bestmögliche Berechnungszeit, welche die stabilste und höchste Bildrate liefert, zu ermitteln.
In Tabelle 6.4 werden die gemessenen Berechnungszeiten der einzelnen Vorgänge gezeigt. Die Kamera liefert insgesamt eine Bildrate von 61 fps. Der gesamte Prozess der Bildaufnahme weißt eine Bildrate von ca. 59 FPS auf.
Der Prozess der Ermittlung der Kugelposition bedarf einer Berechnungszeit von ca. 9 ms. Die Zeit, welche der Agent benötigt um eine Aktion auszuführen, liegt bei ca. 31 ms und lässt sich auf die Kommunikation zwischen Soft- und Hardware zurückführen.

Wenn eine Aktion ausgeführt werden soll, wird ein Befehl über die Arduinoschnittstelle gesendet. Dieser Befehl löst bei der Motorsteuerung die Ansteuersignale für die jeweiligen Aktoren aus. Zur Überprüfung, ob das Signal von der physischen Umgebung zur Motorsteuerung erfolgreich war, gibt es ein `acknowledge`-Signal, welches an die Umgebung zurückgesendet wird. Dieses Signal beinhaltet die Nachricht „Ok“ und wird mittels des Befehls `Serial.println(,,ok‘‘)` gesendet. Die Messung hat ergeben, dass das Warten auf die Bestätigung einen hohen Zeitaufwand bedeutet und somit fast die Gesamte Aktionszeit einnimmt.

In Abbildung 6.18 wird in einem vereinfachtem Schema das zeitliche Verhalten des Systems gezeigt. Zu Beginn wird immer ein Bild aufgenommen und dem System zur Verfü-

gung gestellt. Mittels der Klasse `threading.Event()` ein Event erzeugt, welches ermöglicht die verschiedenen Threads auf einander abzustimmen. Die Methode `event.wait()` ermöglicht es auf das Eintreten eines Events zu warten. Diese Funktion findet hier Anwendung. Die Funktion der Kugelerkennung wartet solange bis ein neues Bild vorhanden ist. Im Anschluss startet der Prozess der Positionsermittelung. Währenddessen beginnt die Bildaufnahme erneut, da sie, wie in Tabelle 6.4 zu sehen, mehr Zeit in Anspruch nimmt. Sobald die Position der Kugel erkannt wurde, wird das Event gesetzt, was dazu führt, dass der wartende Agent nun die Aktion ausführen kann.

6.2.11 Sequenzdiagramm der Unterklasse `PinballCamera`

In der folgenden Abbildung 6.19 wird eine vereinfachte Darstellung der in der Klasse `PinballCamera` auftretenden Sequenzen gezeigt.
Nachdem die Kamera initialisiert wurde, startet der Thread, welcher die Bilder von der Kamera entnimmt. Danach werden die Eckpunkte des Spielfelds kalibriert. Nachdem die Klasse `FieldCropper` diese wiedergegeben hat, wird der Bereich der Flipperarme durch entsprechende Ansteuerungen an den Motor und Aufnahme des Differenzbildes ermittelt. Im Anschluss wird der Thread, welcher für die Positionsbestimmung zuständig wird, gestartet. Mit den entsprechenden Zeitbedingungen entsteht hiermit der Prozess, welcher Informationen über den aktuellen Zustandsraum ausgeben kann.

Die Methode `reset_calibration()` dient dazu, das System nach einer gescheiterten Runde, beispielsweise wenn die Kugel in den Drain gelangt, in einen definierten Ausgangszustand zu versetzen. Zu Beginn wird die interne Variable `calibration_running` aktiviert, wodurch parallele Bildverarbeitungsprozesse während der Kalibrierung unterdrückt und somit anschließend eine erneute Kalibrierung der Flipperarme durchgeführt wird. Dies stellt sicher, dass deren Bewegungsbereich für die nachfolgenden Bildverarbeitungsaufgaben korrekt berücksichtigt wird. Parallel dazu überprüft die Funktion `detect_green_in_bottom_right()`, ob der Startarm in seiner korrekten Position steht und die Startrampe freigibt. Nach einer kurzen Wartezeit wird über die Steuerung der Flippermaschine eine neue Kugel in die Startposition gebracht und die Startmechanik aktiviert. Eine kleine Verzögerung verhindert hierbei, dass der Startarm fälschlicherweise als Bewegung erkannt wird. Schließlich wird die Variable `calibration_running()` wieder deaktiviert und das System ist für die nächste Spielrunde vorbereitet.

Abbildung 6.20 zeigt ein vereinfachtes Klassendiagramm der physischen Umgebung.

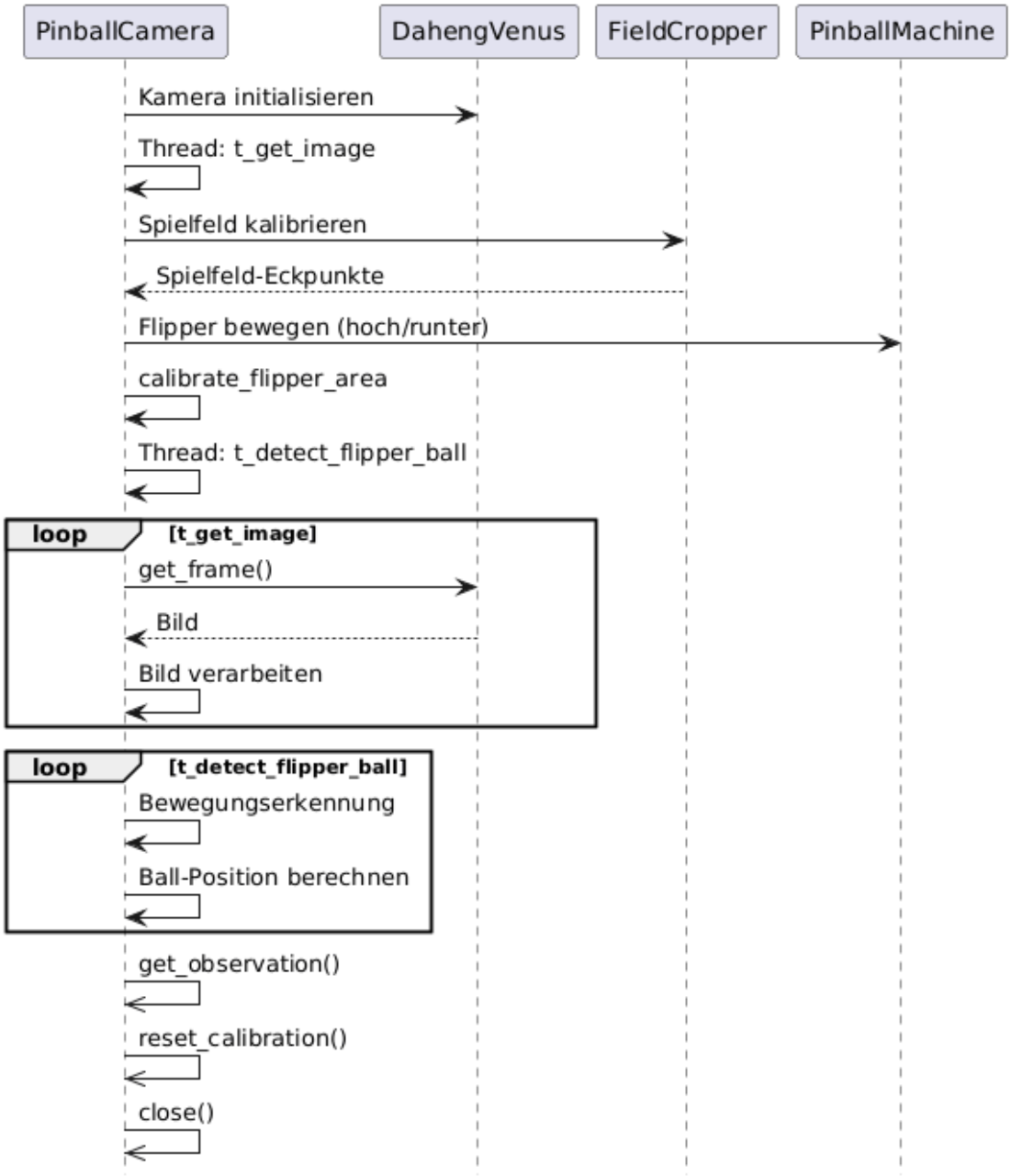


Abbildung 6.19: Vereinfachtes Sequenzdiagramm der Klasse `PinballCamera`

6.3 Entwicklung der virtuellen Umgebung

Um einen KI-gesteuerten Flipperautomaten zu entwickeln, der Zustandsinformationen aus Kamerabildern verarbeitet, ist eine realitätsnahe virtuelle Umgebung unerlässlich. Diese dient als Trainingsumgebung für den KI-Agenten und ermöglicht es, große Mengen an Spielinteraktionen zu simulieren, ohne dass dabei mechanischer Verschleiß oder zeitliche Einschränkungen durch reale Hardware entstehen.

Die virtuelle Umgebung bildet die physikalischen Eigenschaften des Flippers realitätsgetreu nach. Dazu gehören Kollisionsverhalten, Gravitation, Reibung und die Dynamik der Kugel. Statische Spielfeldobjekte wie Außenwände und Hindernisse werden über eine

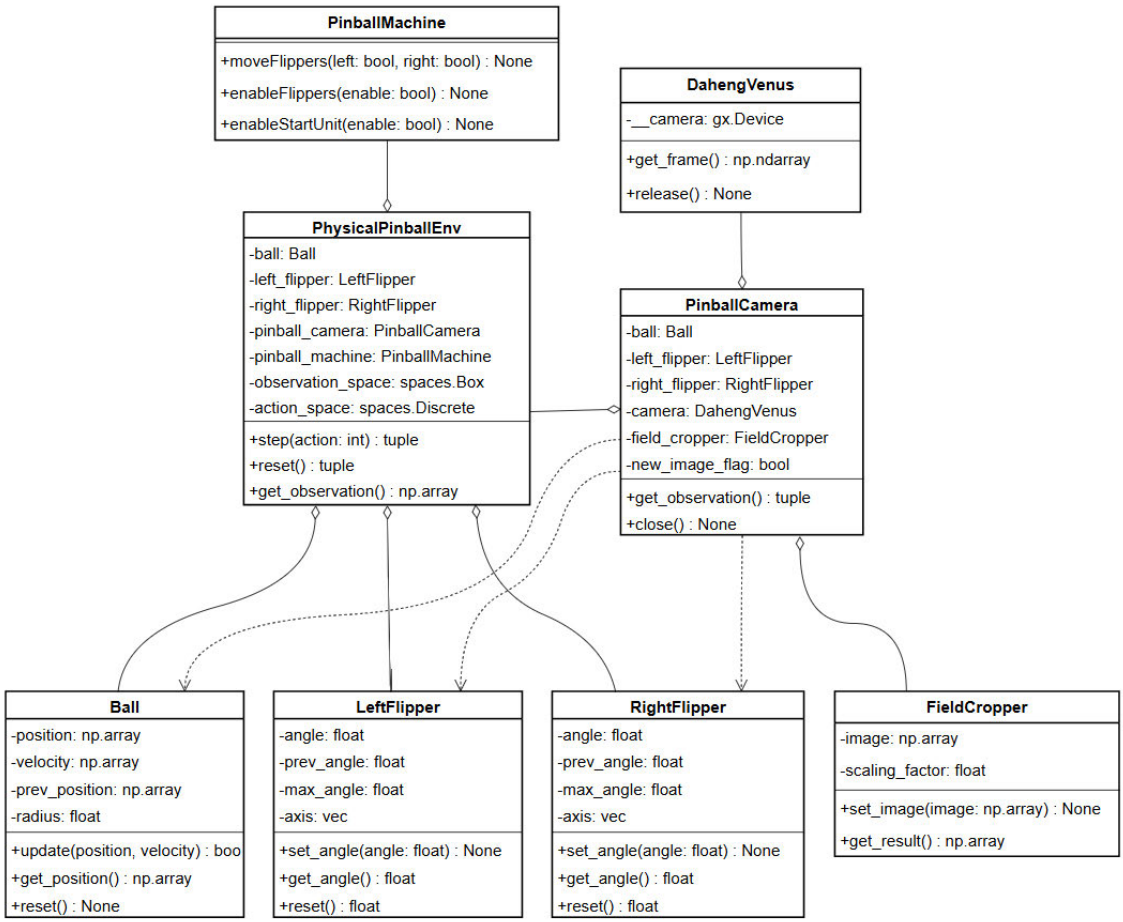


Abbildung 6.20: Vereinfachtes Klassendiagramm der physischen Umgebung

definierte Hinderniskarte modelliert, welche die Struktur des Flipperautomaten abbildet. Diese Karte wird für eine schnelle Kollisionsberechnung eingesetzt.
Die Kollisionsberechnung erlaubt es, das Verhalten der Kugel beim Aufprall auf Hindernisse sowie auf die dynamisch gesteuerten Flipperarme korrekt zu simulieren. Letztere unterliegen einer gesonderten physikalischen Modellierung, da sie durch den Agenten aktiv bewegt werden und unmittelbaren Einfluss auf das Spielgeschehen haben.

6.3.1 3D-Modell für den Rendermodus

Für die Visualisierung der Kugel wurde eine spezielle Render-Klasse `RenderScene` implementiert, welche die gesamte 3D-Modellierung und das Rendering übernimmt. Diese Klasse ist dafür verantwortlich die physikalischen Objekte (z. B. Spielfeld, Flipper, Kugel, Hindernisse) in einer `VPython`-Szene darzustellen. Das Rendering wird nur dann ausgeführt wenn es während des Trainings des Agenten aktiviert ist. Es wurde von Prof. Dr. Marc Hensel eine Grundstruktur des 3D-Modells zur Verfügung gestellt. Diese musste jedoch um notwendige Objekte ergänzt werden. Im Folgenden wird der Aufbau und die Funktionsweise dieser Klasse beschrieben.

Initialisierung der Render-Szene

Beim Erstellen eines Objekts der Klasse `RenderScene` werden zunächst die Hauptkomponenten der Szene initialisiert:

- Übergabeparameter sind die Kugel, der linke und rechte Flipper und optionale Parameter für die Größe und Skalierung der Szene.
- Es werden dynamische Objekte (Kugel und Flipper) und statische Objekte (Spielfeldrahmen, Lanes, Hindernisse) vorbereitet.
- Die Szene (`scene`) wird mit Breite, Höhe, Reichweite, Hintergrundfarbe und weiteren Visualisierungsparametern konfiguriert.
- Zusätzlich wird die Kollisionskarte (Collision Map) skaliert, um die Simulation konsistent mit den physikalischen Größen auszuführen.

Rendering statischer Objekte

Die Methode `render_scene()` ruft nacheinander verschiedene Unterfunktionen auf, welche die statischen Elemente des Spielfelds rendern:

- `render_frame()`: Darstellung des Spielfeldrahmens und der Begrenzungen
- `render_launcher_lane()`: Modellierung der Abschussbande
- `render_arc_lane()`: Konstruktion der Bogendiagonale
- `render_top_corners()`: Modellierung der oberen Bogenstücke zur Spielfeldbegrenzung
- `render_flipper_inlanes()`: Darstellung der inneren Spielfeldbereiche in Nähe der Flipper

Alle Objekte werden mit Hilfe von `box`- und `extrusion`-Primitiven erzeugt und in Listen (`box_shapes`, `arc_shapes`) gespeichert. Dies ermöglicht eine spätere Rotation oder Anpassung der Geometrien.

Rendering dynamischer Objekte

Die dynamischen Objekte sind die Flipper und die Kugel:

- `render_flipper()`: Stellt die Flipper als Quader dar, deren Achsen an den physikalischen Drehpunkten ausgerichtet sind.
- `render_ball()`: Modelliert die Kugel als Spähre, deren Position zu Beginn durch die Startkoordinaten definiert ist.

Die Flipper werden hierbei initial um ihren Neigungswinkel rotiert, sodass sie die korrekte Ausgangsposition einnehmen.

Aktualisierung während der Simulation
Die Methode `update()` aktualisiert den Zustand der dynamischen Objekte während der Simulation:

- Die Kugelposition wird durch die aktuelle physikalische Position (`ball.get_position()`) bestimmt und in der Szene aktualisiert.
- Die Flipperrotationen werden durch den Unterschied zwischen dem alten und neuen Winkel berechnet. Anschließend wird das zugehörige Renderobjekt im entsprechenden Winkel um die jeweilige Drehachse rotiert.

Damit folgt die 3D-Darstellung kontinuierlich dem physikalischen Zustand der Simulation.

Die Klasse `RenderScene` übernimmt die vollständige 3D-Modellierung und Darstellung der Flipper-Umgebung. Sie trennt dabei klar zwischen:

- **statischen Objekten**, welche das Spielfeld und die Hindernisse darstellen, und
- **dynamischen Objekten**, die während der Simulation kontinuierlich aktualisiert werden.

Durch die in Abbildung 6.21 gezeigte Struktur ist es möglich während des Trainings wahlweise eine visuelle Repräsentation der Umgebung zu aktivieren, ohne die physikalische Logik des Agenten zu beeinträchtigen.
Abbildung 6.22 zeigt das finale 3D-Modell, welches für das Training mit eingeschaltetem Rendering angewandt wird. Die Achsenmarkierung zeigt an wie das Koordinatensystem im Modell aufgebaut ist. Der Ursprung des Modells, also (0, 0, 0) wird durch die grüne Kugel in der Mitte markiert. Die rote Markierung steht für die X-Achse. Die grüne Markierung zeigt die Y-Achse. Die Tiefe wird durch die blaue Markierung in Form der Z-Achse gezeigt. Dieses Koordinatensystem dient in der virtuellen Umgebung als Grundlage und wird ebenfalls für die folgenden physikalischen Berechnungen verwendet.

6.3.2 Physikalische Eigenschaften

In diesem Kapitel werden die physikalischen Modelle und Implementierungsentscheidungen beschrieben, die der `Physics`-Klasse zugrunde liegen. Ziel der Implementierung ist es, das Verhalten der Kugel auf dem Flipperfeld realistisch abzubilden. Die physikalische Modellierung umfasst die zeitdiskrete Integration der Kugeldynamik, die Behandlung von Reibungs- und Hangkräften, die Erkennung und Verarbeitung von Kollisionen mit statischen Hindernissen und die spezielle Modellierung der Wechselwirkung mit den beweglichen Flippern.

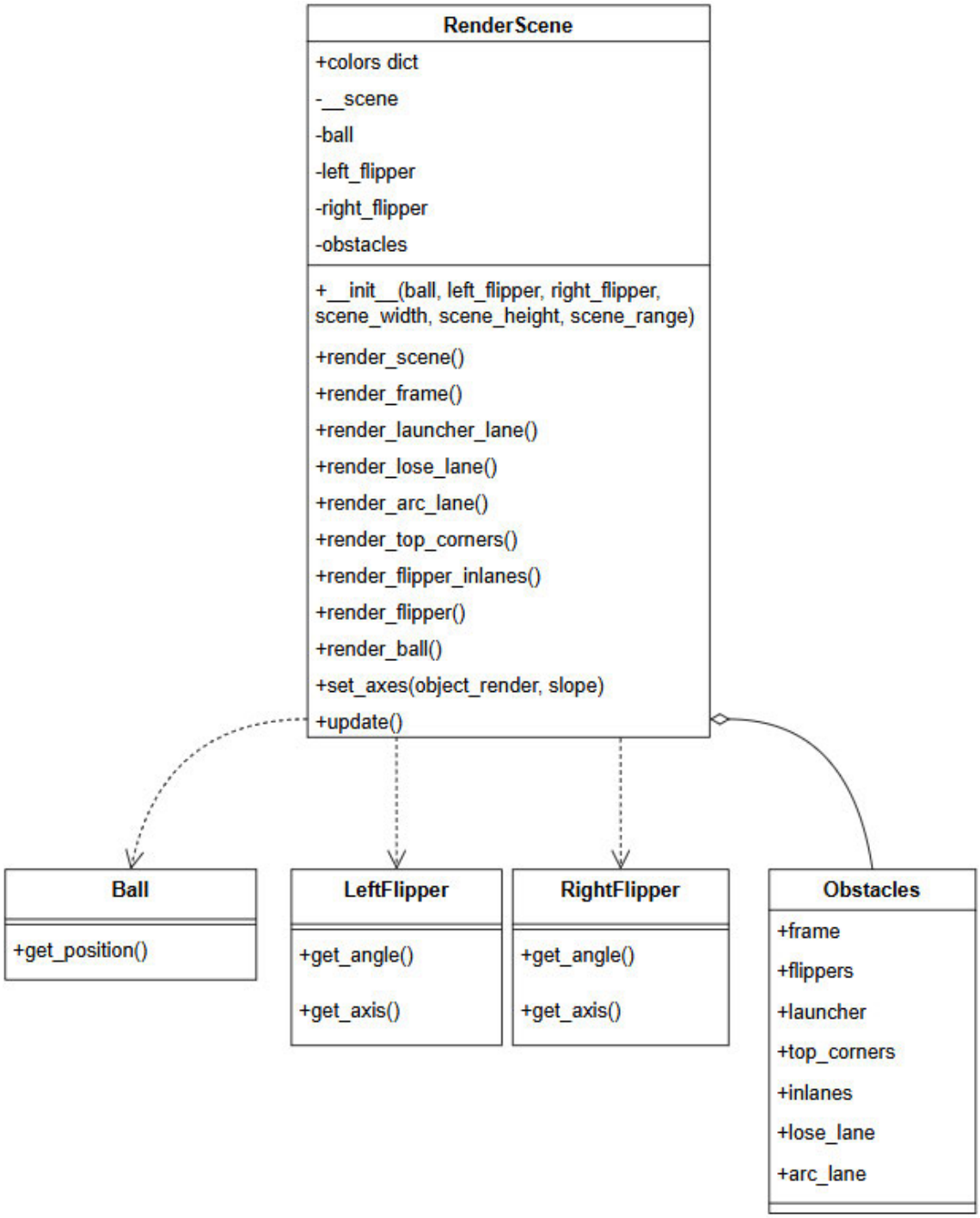


Abbildung 6.21: Klassendiagramm der 3D-Modellierung

Die Kinematik der Kugel wird in festen Zeitschritten (`dt`) numerisch integriert. Hierzu werden die Komponenten der Beschleunigung aus Hangabtriebskräften und Oberflächenreibung hergeleitet und auf die jeweilige Geschwindigkeitskomponente angewendet. Die Position wird unter Verwendung des aktuellen Geschwindigkeitswertes und der errechneten Beschleunigung aktualisiert. Diese einfache, explizit-diskrete Integration ist robust

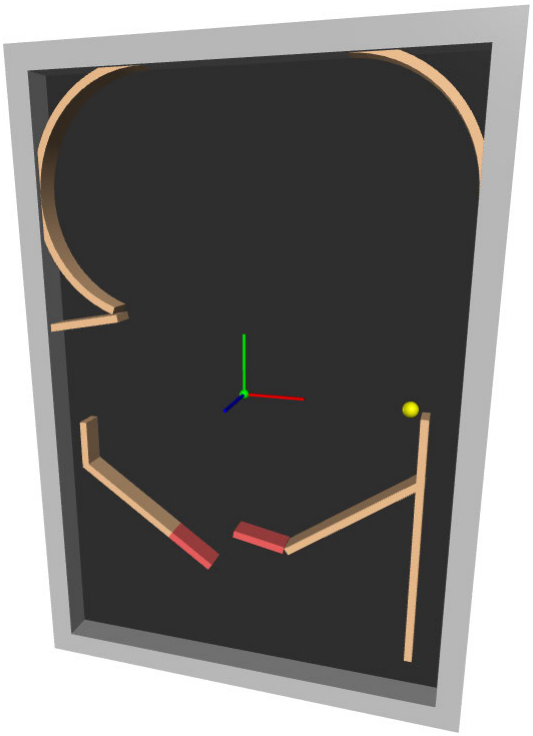


Abbildung 6.22: Laufende Simulation mit allen Objekten inkl. Achsenmarkierung

und hinreichend genau für die in der Umgebung gewählten Zeitschritte, erfordert jedoch geeignete Wahl von `dt` und Reibungsparametern zur numerischen Stabilität.

Die Kollisionsdetektion verfolgt einen hybriden Ansatz: Zum Einen wird eine Lookup-Map (LUT) verwendet, die für Rasterzellen Informationen über Kollisionszustand und lokale Oberflächenneigung enthält. Trifft dies zu, werden die betreffenden Rasterkoordinaten als Kollisionsliste zurückgegeben. Zum Anderen erfolgen für die Flipper präzisere geometrische Tests: Die Flipper werden als orientierte Rechtecke (OBB) mit vier Eckpunkten modelliert; für jede Kantenlinie wird eine Kreis–Linien-Distanzprüfung durchgeführt und zusätzlich wird mittels `shapely.LineString` geprüft, ob die Kugelbahn eine Kante geschnitten hat. Hierdurch werden sowohl Kontakt- als auch Durchquerungsfälle erkannt.

Die Kollisionsantwort ist modular implementiert und in der Methode `process_collision` zusammengefasst. Aus der lokalen Neigung des Hindernisses wird ein Normalenvektor bestimmt. Die Kugelgeschwindigkeit wird in normale und tangentiale Komponenten zerlegt, welche normale Komponente invertiert und mit einem Dämpfungs- und Reibungsfaktor skaliert und schließlich wieder mit der tangentialen Komponente vereinigt. Zusätzlich können durch die Flipper induzierte Geschwindigkeitsanteile, als Tangentialgeschwindigkeit am Treffpunkt, addiert werden. Vor der Kollisionsverarbeitung wird die Kugelposition bei

erkannten Überschneidungen auf einen zuvor gespeicherten Zustand zurückgesetzt und, falls nötig, entlang der Verbindung zum Kollisionspunkt so korrigiert, dass Penetrationen aufgehoben werden.

Die Flipperdynamik ist als kinematisches Modell implementiert: Winkeländerungen werden schrittweise durch `set_flipper_angles` realisiert und durch Grenzwerte (maximale/minimale Winkel) begrenzt. Bei einer Kollision mit einem Flipperarm wird die am Treffpunkt durch die Flipperrotation erzeugte Tangentialgeschwindigkeit berechnet (Produkt aus Winkelgeschwindigkeit und Radius), in Richtung tangential zur Radiuslinie projiziert und als zusätzlicher Geschwindigkeitsanteil in die Kollisionsantwort eingebracht. Diese Behandlung erlaubt es, typische Flipper-Effekte wie Beschleunigungsstöße und Richtungsänderungen realistisch abzubilden.

Zur Robustheit der Simulation wurden mehrere praktische Mechanismen vorgesehen: Skalierungsfaktoren (`SCALING`) wandeln zwischen Raster- und Simulationskoordinaten um und einfache Absicherungen setzen die Kugel bei zu großen Schritten über Objekte oder bei Kollisionen auf einen vorherigen Zustand zurück.
Die folgenden Unterkapitel erläutern die einzelnen Bestandteile dieses Systems im Detail: Zuerst die Modellierung der Kugeldynamik und der Reibungs-/Hangkräfte, danach die Konstruktion und Nutzung der Kollisions-LUT und die geometrischen Kollisionsprüfungen, anschließend die Kollisionsantwort und abschließend die Modellierung und Interaktion der Flipperarme inklusive der numerischen Parameter und Grenzfälle. Dieses Zusammenspiel aus einfachen physikalischen Annahmen, pragmatischer numerischer Integration und gezielten geometrischen Prüfungen bildet die Grundlage für eine realistische und effizient ausführbare Simulation des Flipperfelds.

6.3.3 Rollverhalten der Kugel mit der Funktion `step`

Die Kugeldynamik auf dem geneigten Spielfeld wird durch die Überlagerung zweier Kräfte bestimmt: der hangabwärts gerichteten Komponente der Gewichtskraft und der entgegenwirkenden Reibungskraft.
Die Beschleunigungskomponente entlang einer Achse $i \in \{x,y\}$ ergibt sich aus der Differenz zwischen Hangabtriebskraft und Reibungskraft:

$$a_{\text{Kugel}} = \frac{5}{7}\, g \left(\sin\theta - c_r \cos\theta\right) \tag{6.18}$$

Hierbei beschreibt a_g die resultierende Beschleunigung durch die Schwerkraft, θ_i den Neigungswinkel der Ebene in Bewegungsrichtung, μ den Reibungskoeffizienten und v_i die

aktuelle Geschwindigkeit der Kugel in Richtung i. Der erste Term modelliert die hangabwärts gerichtete Komponente und der zweite Term die Reibung, die stets entgegen der Bewegungsrichtung wirkt.

Ermittelung der Rollreibungszahl

Mit mehreren Videoaufnahmen wurde die Zeit von der höchstmöglichen Position des Spielfeld bis zum Ende das Rollverhalten der Kugel gemessen. Insgesamt wurde eine durchschnittliche Zeit von 1,038 Sekunden errechnet.

- Strecke der Ebene: $s = 0{,}49365$ m
- Neigungswinkel: $\theta = 7°$
- gemessene Zeit: $t = 1{,}038$ s
- Kugel: massiver Körper, Trägheitsmoment $I = \frac{2}{5}mR^2$
- Erdbeschleunigung: $g = 9{,}81\ \mathrm{m/s^2}$

Die Kugel bewegt sich gleichmäßig beschleunigt aus dem Ruhezustand. Nach den Grundgesetzen der Kinematik gilt:

$$s = \frac{1}{2}at^2 \tag{6.19}$$

Hierbei bezeichnet a die effektive Beschleunigung entlang der Ebene. Durch Umstellen ergibt sich:

$$a = \frac{2s}{t^2} \tag{6.20}$$

Mit dem Einsetzen der bekannten Werte ergibt sich:

$$a = \frac{2 \cdot 0{,}49365}{1{,}038^2} \tag{6.21}$$

$$a \approx 0{,}9167\ \mathrm{m/s^2} \tag{6.22}$$

Dies entspricht der mittleren Beschleunigung der Kugel entlang der schiefen Ebene unter Berücksichtigung der Reibungsverluste.

Die Dynamik einer Kugel auf einer geneigten Ebene wird durch das Zusammenspiel von Hangabtriebskraft, Trägheit der Kugel und Rollreibung bestimmt. Für eine massive Kugel

lässt sich die Beschleunigung entlang der Ebene gemäß den Gleichungen der Rollbewegung ausdrücken:

$$a = \frac{5}{7} g \left(\sin\theta - c_r \cos\theta \right) \tag{6.23}$$

Der Faktor $\frac{5}{7}$ berücksichtigt das Trägheitsmoment der Kugel ($I = \frac{2}{5} m R^2$) und beschreibt die Reduktion der linearen Beschleunigung aufgrund der Rotationsenergie. Die Rollreibungszahl c_r beschreibt den proportionalen Anteil der Rollreibung zur Normalkraft. Durch Umstellen nach c_r ergibt sich:

$$c_r = \frac{\sin\theta - \frac{7}{5}\frac{a}{g}}{\cos\theta} \tag{6.24}$$

Die erforderlichen trigonometrischen und normierten Größen werden berechnet als:

$$\sin\theta \approx \sin 7^\circ \approx 0{,}121869 \tag{6.25}$$

$$\cos\theta \approx \cos 7^\circ \approx 0{,}992546 \tag{6.26}$$

$$\frac{a}{g} = \frac{0{,}9167}{9{,}81} \approx 0{,}09345 \tag{6.27}$$

$$\frac{7}{5}\frac{a}{g} \approx 1{,}4 \cdot 0{,}09345 \approx 0{,}1308 \tag{6.28}$$

Damit ergibt sich die Rollreibungszahl:

$$c_r = \frac{0{,}1218 - 0{,}1308}{0{,}9925} \tag{6.29}$$

$$c_r \approx -0{,}0090 \tag{6.30}$$

Die Wiedergabegeschwindigkeit der Messungen, wurden auf die niedrigste Stufe von 10% verlangsamt. Aufgrund mangelnder technischer Gegebenheiten, war es nicht möglich eine feinere Messung zu vollziehen. Hierdurch ist eine nicht real vertretbare Rollreibungszahl gemessen und berechnet worden. Siehe Ergebnis der Berechnung 6.30. Demnach wurde in nachfolgenden Berechnungen mit einer Rollreibungszahl von $c_r = 0{,}0$ gerechnet.

Schrittfunktion der Flippersimulation

Die `step()`-Funktion bildet einen Zeitschritt der physikalischen Simulation des Flipperautomaten ab. Sie berechnet die Bewegung der Kugel unter Einfluss von Neigung, Rollreibung, Geschwindigkeit und Kollisionen und die Wirkung der Flipper.

1. Definition der Parameter

Für den aktuellen Zeitschritt Δt werden zunächst die Neigungswinkel des Spielfeldes in den beiden Achsen festgelegt:

$$\theta_x = 0°, \tag{6.31}$$

$$\theta_y = 7 \tag{6.32}$$

Die Sinuswerte der Neigung und der Rollreibungskoeffizient μ_r werden für die Berechnung der Beschleunigung benötigt:

$$a_{\text{fric},i} = \mu_r \, g \cos(\theta_i), \qquad i \in \{x,y\}. \tag{6.33}$$

2. Berechnung der Beschleunigung

Die Beschleunigung der Kugel entlang der Y-Achse setzt sich zusammen aus der Hangabtriebskraft a_{tilt} und der Rollreibung a_{roll}. Es gilt:

$$a_y = \begin{cases} -\frac{5}{7} g \left(\sin\theta - c_r \cos\theta \right) \cdot (\sin(\theta_y) - a_{\text{fric},y}), & v_y \leq 0, \\ -\frac{5}{7} g \left(\sin\theta - c_r \cos\theta \right) \cdot (\sin(\theta_y) + a_{\text{fric},y}), & v_y > 0. \end{cases} \tag{6.34}$$

Die Beschleunigung hängt von der Vorzeichenrichtung der Geschwindigkeit ab, da die Rollreibung immer der Bewegung entgegen wirkt.

3. Überprüfung auf Stillstand

Wenn die Hangabtriebskraft kleiner als die Rollreibung ist, kann die Kugel auf der Achse zum Stillstand kommen:

$$|\sin(\theta_i)| < a_{\text{fric},i} \quad \Rightarrow \quad v_i = 0, \quad a_i = 0. \tag{6.35}$$

4. Update der Geschwindigkeit

Die Geschwindigkeit wird über den Zeitschritt Δt aktualisiert:

$$v_i(t + \Delta t) = v_i(t) + a_i \, \Delta t, \quad i \in \{x,y\}. \tag{6.36}$$

5. Update der Position

Die Position der Kugel wird unter Berücksichtigung der aktuellen Geschwindigkeit und

der Beschleunigung berechnet:

$$p_i(t + \Delta t) = p_i(t) + v_i(t)\,\Delta t + \frac{1}{2} a_i\,(\Delta t)^2, \quad i \in \{x,y\}. \tag{6.37}$$

6. Überprüfung auf entstandene Kollisionen
Im letzten Schritt wird überprüft ob durch ermittelte Positionsänderung eine Kollision mit einem der Flipperarmen oder eines statischen Objekt entstanden ist.

Zusammenfassung der Schrittfolge
Die Funktion `step()` implementiert damit die vollständige und zeitdiskrete Bewegung der Kugel:

1. Berechnung der Beschleunigung aus Hangabtrieb und Rollreibung,
2. Diskrete Integration der Geschwindigkeit,
3. Aktualisierung der Position und
4. Berechnung möglicher Kollisionen.

6.3.4 Kollisionskarte für statische Objekte

Die Kollisionserkennung wird in zwei verschiedene Arten aufgeteilt. Die Kollision mit den Flipperarmen wird sofort berechnet werden. Dies bedeutet, dass mit jedem Zeitschritt innerhalb der Simulation, die potentielle Kollision mit den beiden Flipperarmen abgefragt wird. Dies hat den Hintergrund, dass die Flipperarme neben der Kugel die einzigen dynamischen Objekt sind.

Darüber hinaus wird eine Kollisionskarte generiert, welche in den einzelnen Verhältnissen und Anordnungen der gerenderten Objekten gleich ist. Die Kollisionskarte ist in einzelne Zellen unterteilt. Bei jedem Zeitschritt ist es so möglich zu, überprüfen ob sich die Kugel in einer Zelle befindet in der bereits ein Objekt vorhanden ist. Wenn die Kugel eine Koordinate der Kollisionskarte betritt, in der ein Objekt eingetragen wurde, muss eine Kollision berechnet werden.
Abbildung 6.23 zeigt in einem vereinfachtem Modell die Kollisionskarte. Hierbei wird durch gelb markierte Zellen auf die möglichen Kollisionen hingewiesen.
Diese Kollisionskarte soll auf zwei Ebenen Informationen enthalten: Zum Einen ob in der jeweiligen Zelle eine Kollision auftritt und zum Andere wenn ja, den dazugehörigen Berechnungswinkel der Kollision.

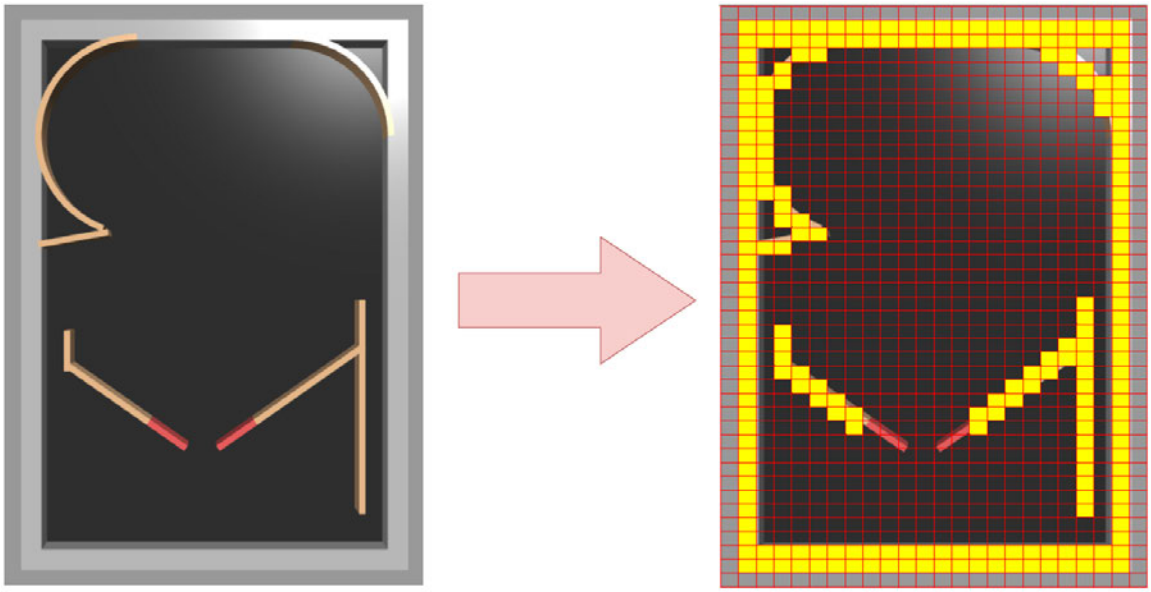


Abbildung 6.23: Konzept der Kartengestaltung für Hindernisse

6.3.5 Kollisionserkennung mit Spielfeldobjekten

Die Funktion `detect_and_process_collision()` überprüft, ob die Kugel mit einem auf der Karte eingetragenen Hindernis kollidiert. Grundlage fierfür ist eine sogenannte Kollisions-LUT (Look-Up Table).

Spielfeldgrenzen und Kugelposition

Das Spielfeld wird über seine Dimensionen und Ränder beschrieben:

$$x = x_{\text{Kugel}} + \tfrac{1}{2}\big(2\,B_x + L_x\big), \tag{6.38}$$

$$y = y_{\text{Kugel}} + \tfrac{1}{2}\big(2\,B_y + L_y\big), \tag{6.39}$$

hierbei stellen B_x, B_y die Randbreiten und L_x, L_y die Feldgrößen in x- und y-Richtung dar. Diese Konvertierungsformel sorgt dafür, dass die Koordinaten von der VPhytonumgebung auf die LUT angewendet werden können. Anders als bei der physischen Umgebung müssen der Rahmen mit in die Karte eingetragen und beachtet werden, sodass die Berechnung der Kollision mit den Banden getätigt werden kann.

Kollisionsprüfung über Kreisschnitt (`point_in_circle`)

Die Funktion `point_in_circle()` überprüft, ob der Kreis mit Zentrum (c_x, c_y) und Radius r einen Eintrag der LUT überdeckt, der mit `1` markiert ist (d. h. ein Hindernis). Mathematisch wird das Gebiet des Kreises mit den Hindernispunkten der LUT verglichen:

$$(x - c_x)^2 + (y - c_y)^2 \leq r^2 \tag{6.40}$$

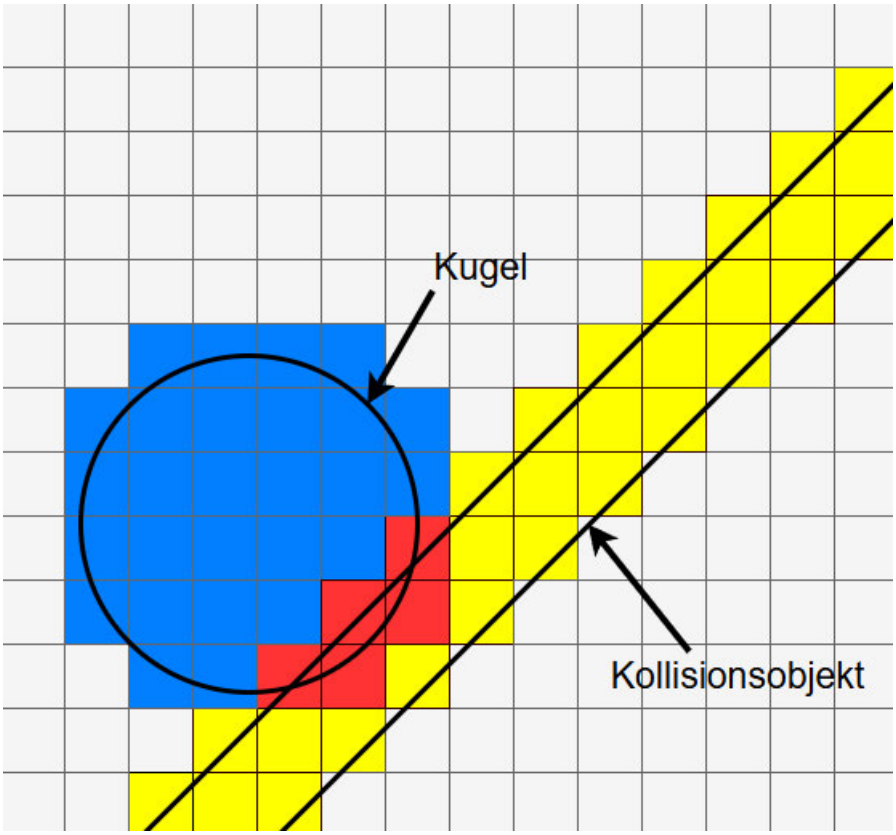


Abbildung 6.24: Vereinfachte Darstellung der Kollisionüberprüfung mit statischen Objekten

Treffen mindestens ein Punkt (x,y) aus dem Kreisgebiet und ein Hindernispixel zusammen, gilt eine Kollision als erkannt. Zusätzlich liefert die Funktion eine Liste aller betroffenen Rasterpunkte. Abbildung 6.24 zeigt wie die Kollision der Kugel mit statischen Objekten von der Funktion `point_in_circle`) betrachtet wird. Die rot markierten Zellen stellen die Kollision dar.

Korrektur der Kugelposition

Wird eine Kollision festgestellt, setzt die Simulation die Kugel zunächst auf die vorherige Position $\mathbf{p}_{t-\Delta t}$ und Geschwindigkeit $\mathbf{v}_{t-\Delta t}$ zurück. Danach wird überprüft, ob die aktuelle Kugelposition immer noch in den Kollisionspunkten liegt. Für jeden Kollisionspunkt $\mathbf{c}_i$ wird der Vektor berechnet:

$$\mathbf{d}_i = \mathbf{p}_{\text{Kugel}} - \mathbf{c}_i \tag{6.41}$$

Der Betrag $\|\mathbf{d}_i\|$ beschreibt den Abstand. Liegt dieser unterhalb eines Sicherheitswerts wird eine Korrektur vorgenommen. Der Sicherheitsabstand von $< 1{,}5 \cdot r$ zwischen dem Kugelmittelpunkt und der äußeren Kante des Kollisionsobjekts hat sich in der Entwicklungs- und Testphase als robuster Wert erwiesen. Daraufhin konnten keine folgenden Fehler in

der Kollisionsberechnung ermittelt werden:

$$\mathbf{p}_{\text{Ball}} \leftarrow \mathbf{p}_{\text{Ball}} + \left(\frac{\mathbf{d}_i}{\|\mathbf{d}_i\|} \right) \cdot \left(1{,}5 \cdot r - \|\mathbf{d}_i\|\right). \tag{6.42}$$

Damit wird der Kugel aus dem Hindernisbereich herausgeschoben, sodass keine Überschneidung verbleibt.

Neigungsinformation (Slope)
Nach der Positionskorrektur werden für alle Kollisionspunkte die Neigungswinkel θ_i aus der LUT gelesen und in eine Liste geschrieben. Die am häufigsten auftretende Neigung wird ausgewählt:

$$\theta_{\text{dom}} = \text{argmax}_\theta \left(\text{Häufigkeit}(\theta_i)\right). \tag{6.43}$$

Dieser „dominante Winkel“ beschreibt die Hauptorientierung des Hindernisses, mit dem die Kugel kollidiert ist, und wird im nächsten Schritt der Kollisionsdynamik (z. B. zur Berechnung des Abprallwinkels) verwendet. Dieser Schritt wird dann wichtig, wenn die Kugel in einem Schritt auf zwei Objekte gleichzeitig trifft und mit unterschiedlichen Winkel in Kontakt kommt.

Funktion `check_point_in_circle`
Die Funktion `check_point_in_circle()` ruft intern `point_in_circle()` mit der aktuellen Kugelposition auf. Sie aktualisiert die Flags `collision_detected` und `collision_list`,
sodass `detect_and_process_collision()` darauf reagieren kann.
Formal überprüft sie:

$$\texttt{h} = \Big(\exists (x,y) \in \text{LUT} : (x - c_x)^2 + (y - c_y)^2 \leq r^2 \wedge \text{LUT}(x,y) = 1\Big). \tag{6.44}$$

Wobei h die boolesche Variable für `collision_detected` steht.
Damit liefert die Funktion die physikalisch konsistente Grundlage, auf der anschließend die eigentliche Kollisionsdynamik berechnet wird.

6.3.6 Kollision mit den Flipperarmen

Die Funktion `flipper_collision()` überprüft in jedem Simulationsschritt, ob die Kugel mit einem der beiden Flipper kollidiert, und berechnet im Fall einer Kollision die zusätzliche Geschwindigkeit, die aus der Flipperbewegung am Kollisionspunkt resultiert und den vom Flipper gegebenen Kollisionswinkel. Die Implementierung kombiniert geometrische Prüfungen (Abstand von Kugelmittelpunkt zu Liniensegmenten der Flipperkante

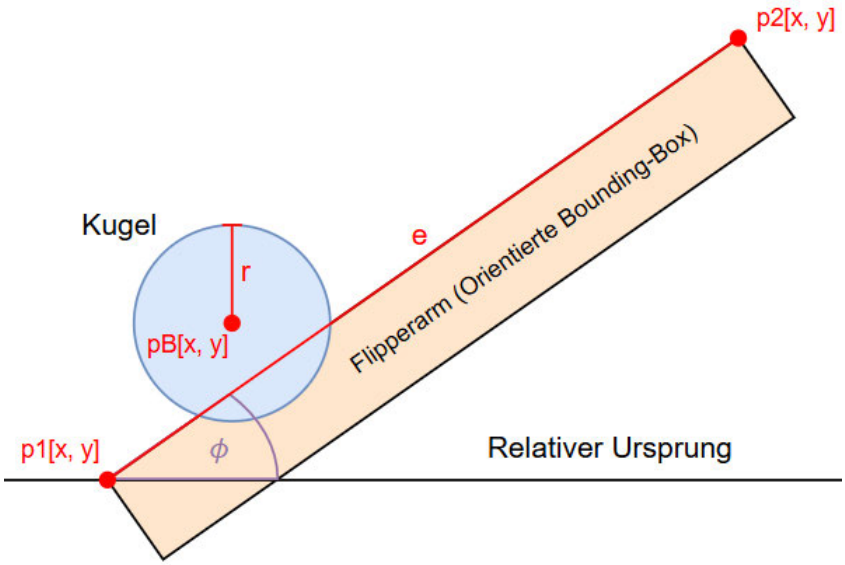


Abbildung 6.25: Vereinfachte Darstellung einer Kollision mit eingetragenen Grundgrößen

und Durchgangstest der Kugelbahn durch eine Kante) mit einer kinematischen Berechnung der Flipper-Tangentialgeschwindigkeit.

Geometrische Grundgrößen

Die Kugel wird durch ihren Mittelpunkt $\mathbf{p}_B$ und Radius r beschrieben; ein Flipper wird als orientierte rechteckförmige Box (OBB, Orientierte Bounding-Boxen) um seinen Drehpunkt modelliert. Jede Kante eines Flippers sei durch zwei Punkte $\mathbf{p}_1$ und $\mathbf{p}_2$ gegeben; der Vektor dieser Kante ist $\mathbf{e} = \mathbf{p}_2 - \mathbf{p}_1$. Abbildung 6.25 zeigt die genannten Größen in einer vereinfachten Darstellung einer Kollision zwischen Kugel und Flipperarm.

Projektions- und Abstandsberechnung

Zur Bestimmung des nächsten Punktes auf der Kante und des Abstands wird die Projektion des Kugelmittelpunkts $\mathbf{p}_B$ auf das Liniensegment verwendet. Der parametrisierte Punkt auf der Kante ist $\mathbf{p}_1 + t\mathbf{e}$ mit dem Parameter t. Dieser berechnet sich durch

$$t = \frac{(\mathbf{p}_B - \mathbf{p}_1) \cdot \mathbf{e}}{\mathbf{e} \cdot \mathbf{e}}. \tag{6.45}$$

Für eine unendliche Linie ist dieser Wert ausreichend. Da im Flippermodell jedoch nur ein endliches Segment zwischen $\mathbf{p}_1$ und $\mathbf{p}_2$ existiert, wird t auf den Bereich [0,1] beschränkt:

$$t = \max\Big(0, \min\Big(1, \tfrac{(\mathbf{p}_B - \mathbf{p}_1)\cdot\mathbf{e}}{\mathbf{e}\cdot\mathbf{e}}\Big)\Big). \tag{6.46}$$

Diese Operation wird im Code mit `max` und `min` umgesetzt und stellt sicher, dass die Projektion nicht außerhalb des Liniensegments liegt („clamping“). Physikalisch entspricht

dies der Tatsache, dass die Kugel nicht mit einer unendlichen Linie, sondern nur mit dem realen Flipper kollidieren kann.
Der nächstgelegene Punkt auf dem Segment lautet:

$$\mathbf{p}_{\text{nearest}} = \mathbf{p}_1 + t\mathbf{e}. \tag{6.47}$$

Der Abstand d vom Kugelmittelpunkt zur Kante ergibt sich durch:

$$d = \|\mathbf{p}_{\text{B}} - \mathbf{p}_{\text{nearest}}\|. \tag{6.48}$$

Physikalisch bedeutet $d < r$, dass die Kugel mit der Kante in Kontakt steht (Kollision). In der Implementierung prüft `check_circle_line_collision` genau diese Bedingung. Wird die Kollision erkannt, so wird zusätzlich der Normalenvektor am Kollisionspunkt berechnet:

$$\mathbf{n} = \frac{\mathbf{p}_{\text{B}} - \mathbf{p}_{\text{nearest}}}{\|\mathbf{p}_{\text{B}} - \mathbf{p}_{\text{nearest}}\|}. \tag{6.49}$$

Durchquerungstest (Line-Crossing)
Um Fälle zu erfassen, in denen die Kugel zwischen zwei Simulationsschritten eine Flipperkante durchquert, wird geprüft, ob das Liniensegment der Kugelbahn vom vorherigen Mittelpunkt $\mathbf{p}_{\text{B}}^{\text{prev}}$ zum aktuellen Mittelpunkt $\mathbf{p}_{\text{B}}$ die Flipperkante schneidet. Mathematisch ist dies ein Segment-Segment-Intersectionsproblem; die Implementierung nutzt hierfür `LineString(...).intersects(...)`. Ein positives Ergebnis signalisiert eine Durchquerung, auch wenn in der aktuellen Position kein direkter Abstandskontakt vorliegt.

OBB-Handling und Rotation
Die Funktion `check_obb_collision_and_crossing` konstruiert aus den Flipperparametern die Eckpunkte eines Rechtecks und rotiert diese um den Flipperwinkel ϕ:

$$R(\phi) = \begin{pmatrix} \cos\phi & -\sin\phi \\ \sin\phi & \cos\phi \end{pmatrix}. \tag{6.50}$$

Die globalen Eckpunkte lauten:

$$\mathbf{c}_k^{\text{global}} = \mathbf{p}_{\text{Flipper}} + R(\phi)\,\mathbf{c}_k. \tag{6.51}$$

Jede Kante des Rechtecks wird mit den oben beschriebenen Methoden ((6.46)–(6.49)) geprüft.

Tangentialgeschwindigkeit des Flippers

Wird eine Kollision festgestellt, berechnet `flipper_velocity_at_collision` die Geschwindigkeit, die durch die Drehung des Flippers am Kollisionspunkt erzeugt wird. Mit $\mathbf{p}_\mathrm{A}$ als Flipperachse und $\mathbf{p}_\mathrm{K}$ als Kollisionspunkt gilt:

$$\mathbf{r} = \mathbf{p}_\mathrm{K} - \mathbf{p}_\mathrm{A}, \qquad r = \|\mathbf{r}\|. \tag{6.52}$$

Bei einer Winkelgeschwindigkeit ω entsteht am Kollisionspunkt die Tangentialgeschwindigkeit:

$$v_\mathrm{tang} = \omega\, r. \tag{6.53}$$

Die Richtung ist senkrecht zum Radiusvektor:

$$\hat{\mathbf{t}} = \frac{1}{r}\,(-r_y, r_x). \tag{6.54}$$

Damit ergibt sich der Geschwindigkeitsvektor:

$$\mathbf{v}_\mathrm{Flipper} = v_\mathrm{tang} \cdot \hat{\mathbf{t}} = \omega\,(-r_y, r_x). \tag{6.55}$$

Konditionale Aktivierung und Rückgabe

Die Geschwindigkeit $\mathbf{v}_\mathrm{Flipper}$ wird nur dann berücksichtigt, wenn der Spieler den jeweiligen Flipper betätigt (`left_pressed` bzw. `right_pressed`) und die Flipperstellung innerhalb der erlaubten Winkel liegt. Andernfalls wird $\mathbf{v}_\mathrm{Flipper} = \mathbf{0}$ gesetzt.
Die Funktion liefert schließlich: (i) die durch die Flipperbewegung erzeugte Geschwindigkeit, (ii) den aktuellen Flipperwinkel und (iii) ein Wahrheitsflag, das eine Kollision signalisiert. Diese Werte fließen in die nachfolgende Kollisionsdynamik ein und erlauben die Simulation des durch den Spieler kontrollierten Kugelimpulses.

6.3.7 Allgemeine Kollisionsberechnung

Die Methode `process_collision()` beschreibt die physikalische Reaktion der Kugel bei einem Zusammenstoß mit einem Hindernis. Ausgangspunkt sind die aktuellen Geschwindigkeits - und Positionswerte der Kugel. Das Hindernis wird durch einen Neigungswinkel φ charakterisiert, aus dem der Normalenvektor bestimmt wird:

$$\mathbf{n} = \big(-\sin(\varphi), \cos(\varphi)\big) \tag{6.56}$$

Dieser Vektor steht senkrecht zur Oberfläche und ermöglicht die Zerlegung der Geschwindigkeit in Normal- und Tangentialkomponenten.

Die Geschwindigkeit der Kugel vor der Kollision ist:

$$\mathbf{v} = (v_x, v_y). \tag{6.57}$$

Die Projektion von $\mathbf{v}$ auf den Normalenvektor liefert die normale Komponente

$$\mathbf{v}_{\text{normal}} = (\mathbf{v} \cdot \mathbf{n})\,\mathbf{n}, \tag{6.58}$$

Daneben wird die tangentiale Komponente als Restvektor beschrieben wird:

$$\mathbf{v}_{\text{tangential}} = \mathbf{v} - \mathbf{v}_{\text{normal}}. \tag{6.59}$$

Die Kollision selbst wird durch eine Spiegelung der Normalgeschwindigkeit modelliert. Dabei kehrt sich die Richtung um und die Geschwindigkeit wird mit einem Reibungskoeffizienten μ skaliert, wodurch Energieverluste berücksichtigt werden:

$$\mathbf{v}_{\text{normal}}^{\text{neu}} = -\mu\,\mathbf{v}_{\text{normal}}. \tag{6.60}$$

Die Tangentialkomponente bleibt im Wesentlichen erhalten, sodass keine Beschleunigung senkrecht zur Oberfläche wirkt. Zusätzlich kann eine durch Flipperbewegung erzeugte Geschwindigkeit $\mathbf{v}_{\text{flipper}}$ hinzukommen. Die resultierende Geschwindigkeit nach der Kollision ergibt sich:

$$\mathbf{v}^{\text{neu}} = \mathbf{v}_{\text{tangential}} + \mathbf{v}_{\text{normal}}^{\text{neu}} + \mathbf{v}_{\text{flipper}}. \tag{6.61}$$

Zur Charakterisierung der neuen Bewegungsrichtung wird der Austrittswinkel bestimmt:

$$\alpha_{\text{exit}} = \arctan\left(\frac{v_y^{\text{neu}}}{v_x^{\text{neu}}}\right). \tag{6.62}$$

Die neue Position der Kugel nach dem Zeitschritt Δt folgt aus der Integration der Geschwindigkeit:

$$x(t + \Delta t) = x(t) + v_x^{\text{neu}}\,\Delta t, \tag{6.63}$$

$$y(t + \Delta t) = y(t) + v_y^{\text{neu}}\,\Delta t. \tag{6.64}$$

Physikalisch beschreibt diese Vorgehensweise einen elastisch gedämpften Stoß mit Oberflächenreibung. Der Energieverlust hängt von μ ab, während die Tangentialbewegung weitgehend erhalten bleibt. Durch den Zusatzterm $\mathbf{v}_{\text{flipper}}$ können Flipper gezielt zusätzliche Impulse übertragen, sodass die Kugel nicht nur reflektiert, sondern auch aktiv beschleunigt werden kann. Auf diese Weise wird ein realistisches und steuerbares Bewegungsverhalten im Flipper erzeugt.

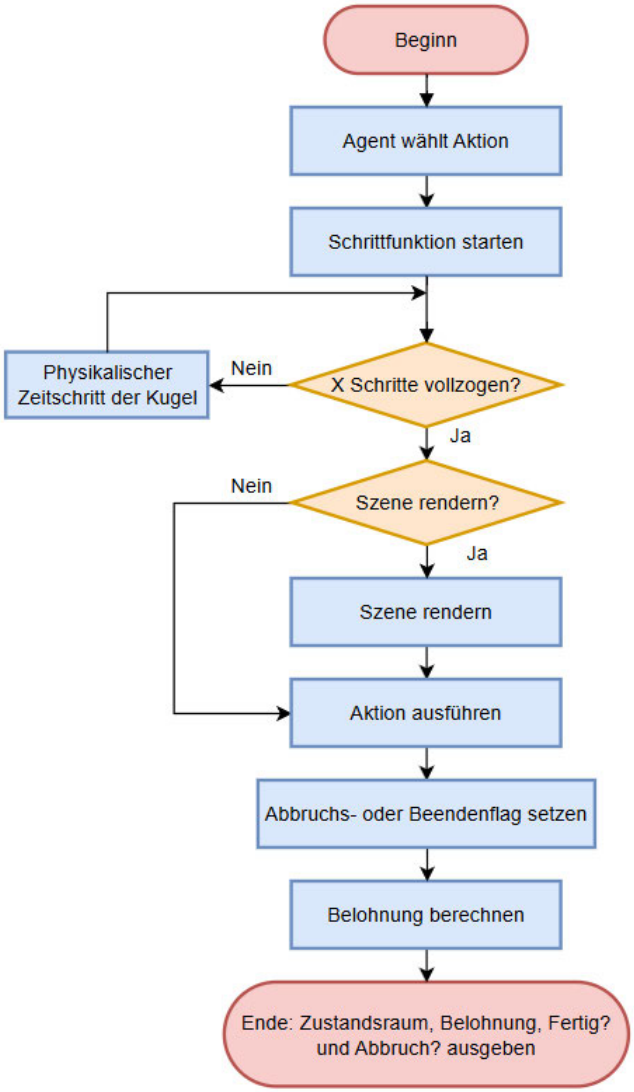


Abbildung 6.26: Vereinfachtes Ablaufdiagramm der virtuellen Umgebung

Abbildung 6.26 zeigt den Ablauf der Schrittfunktion `step()` vom Aufruf mit der Aktion durch den Agenten bis zum Ende und Ausgabe der Zustandsräume sowie die Belohnung des Schritts. X stellt die Anzahl der Millisekunden dar, die das System für die Aufnahmen des Bildes beim physischen Demonstrator benötigt. Die Differenzzeit stellt die größte Herausforderung des Agenten dar, da sie dem System die nächsten Zustände vorauszusehen und vorzeitig zu handeln aufzwingt. Die Systemstruktur wird durch Abbildung 6.27 präsentiert und stell ein vereinfachtes Klassendiagramm dar.

6.4 Entwicklung des RL-Agenten

6.4.1 Wahl des RL-Frameworks

Für die Realisierung wurde das Framework PyTorch als zentrales Entwicklungstool für das Training und die Umsetzung der neuronalen Netze ausgewählt. Die Wahl wurde auf

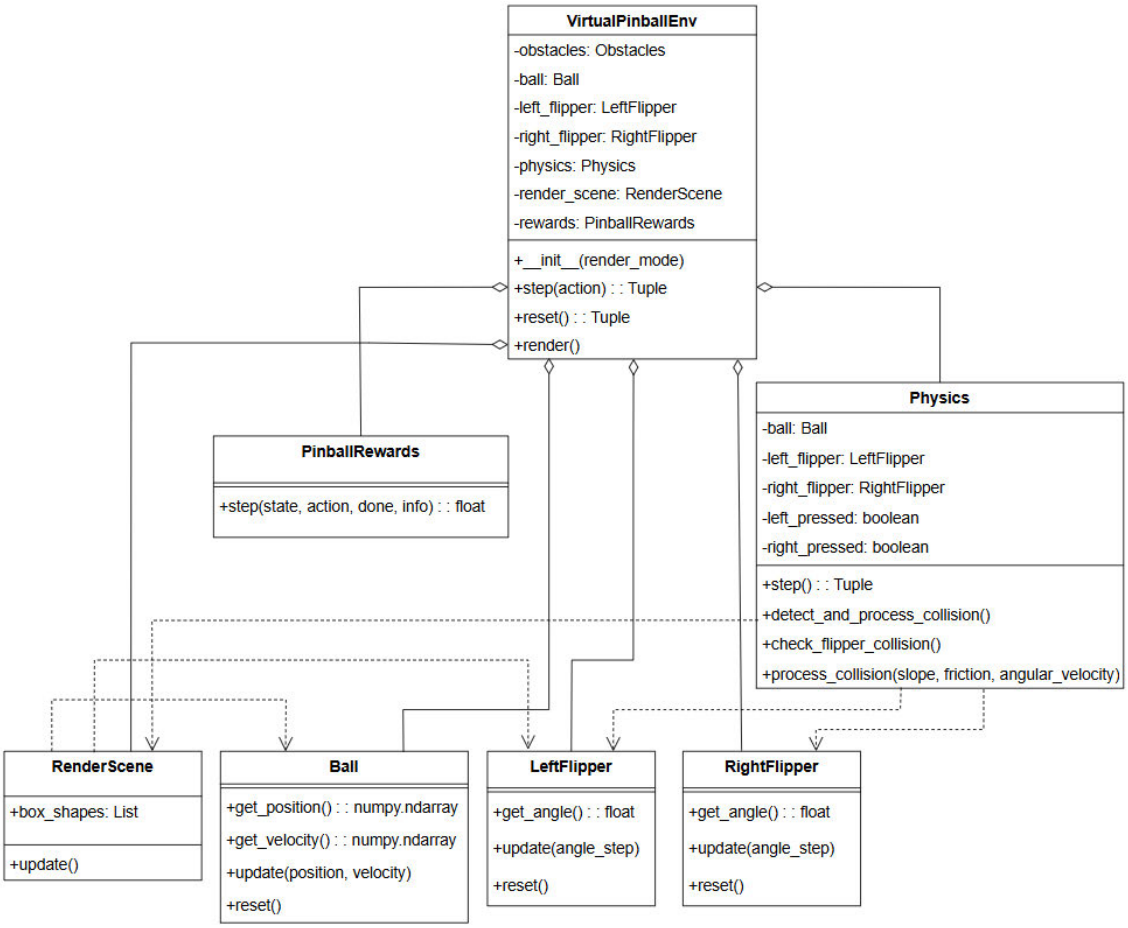


Abbildung 6.27: Vereinfachtes Klassendiagramm der virtuellen Umgebung

Basis einer Abwägung gegenüber alternativen Frameworks wie TensorFlow/Keras, JAX oder klassischen Machine-Learning-Bibliotheken (z. B. Scikit-learn) getroffen.
PyTorch ermöglicht eine besonders flexible Modellierung von Netzarchitekturen und erleichtert das Debugging und die iterative Entwicklung. Letzteres ist gerade in einem Forschungs- und Prototypenkontext wie einer Masterarbeit entscheidend. Im Gegensatz dazu arbeitet TensorFlow in seiner ursprünglichen Version mit statischen Berechnungsgrafen, sodass Fehlersuche und Modellanpassung erschwert werden[22] [23].
Ein weiterer entscheidender Faktor ist die breite Unterstützung durch die Forschungscommunity. PyTorch hat sich in den letzten Jahren in der wissenschaftlichen Forschung weitgehend durchgesetzt, insbesondere im Bereich des Deep Reinforcement Learnings und der Computer Vision[24].
JAX bietet zwar eine sehr hohe numerische Effizienz und automatische Differenzierung, ist jedoch primär auf experimentelle Forschung ausgerichtet und verfügt nicht über denselben Reifegrad und Umfang an Modellbibliotheken wie PyTorch[25].

[22] `https://phoenixnap.de/blog/pytorch-vs-tensorflow`, abgerufen am 30. September 2025.
[23] `https://www.geeksforgeeks.org/deep-learning/`, abgerufen am 30. September 2025.
[24] `https://arxiv.org/abs/2107.14171`, abgerufen am 30. September 2025.
[25] `https://www.techbloat.com/jax-vs-pytorch-differences-and-similarities-2025.`

Scikit-learn wiederum ist für klassische ML-Methoden hervorragend geeignet, deckt aber moderne tiefe neuronale Netze nur unzureichend ab.
Darüber hinaus unterstützt PyTorch die Integration mit Hardwarebeschleunigung (GPU/TPU) in einer besonders nutzerfreundlichen Weise. Für das Training von komplexen visuellen RL-Modellen, die durch die Kamerasensorik des Flipperautomaten angetrieben werden, ist diese GPU-Nutzung unverzichtbar. TensorFlow bietet hier zwar ähnliche Möglichkeiten, jedoch wird PyTorch aufgrund seiner transparenten und einfachen Handhabung von Tensors und CUDA-Operationen in der Praxis oft bevorzugt[26].

6.4.2 Wahl der API

Bei der Entwicklung der Reinforcement-Learning-Umgebung wurde sich für Gymnasium entschieden, da es sich als Standard-API im Bereich des Reinforcement Learnings etabliert hat. Ein entscheidender Vorteil liegt in der einheitlichen Schnittstelle, die alle Umgebungen bereitstellen: Mit Methoden wie `reset`, `step` und `render` lassen sich Agenten konsistent ansteuern, unabhängig von der spezifischen Domäne oder dem Schwierigkeitsgrad der Umgebung. Dadurch können Implementierungen unkompliziert auf unterschiedliche Szenarien übertragen werden. Zudem erlaubt die klare Struktur von Gymnasium, eigene Umgebungen zu definieren, ohne die Kompatibilität mit der bestehenden Infrastruktur zu verlieren. Auch die große Community rund um Gymnasium trägt dazu bei, dass Dokumentation, Tutorials und Beispielimplementierungen leicht verfügbar sind und so die Entwicklung beschleunigen.
Insgesamt bietet die Verwendung von Gymnasium eine robuste, flexible und zukunftssichere Grundlage für die Arbeit mit Reinforcement-Learning-Algorithmen[27].

6.4.3 Entwicklung des Agenten

Die Entwicklung des Agenten basiert auf dem Prinzip des Deep Q-Learning (DQN). Dieses Verfahren kombiniert klassische Q-Learning-Ansätze des Verstärkungslernens mit neuronalen Netzen, um auch in hochdimensionalen Zustandsräumen effektive Strategien zu lernen. Ziel ist es, die Q-Funktion

$$Q(s,a) \approx \mathbb{E}\left[r + \gamma \max_{a'} Q(s',a') \,\middle|\, s,a\right], \tag{6.65}$$

durch ein neuronales Netz zu approximieren. Hierbei beschreibt s den aktuellen Zustand, a die gewählte Aktion, r die erhaltene Belohnung, s' den Folgezustand und $\gamma \in [0,1]$ den Diskontfaktor, der zukünftige Belohnungen abwertet.

`html`, abgerufen am 30. September 2025.

[26] `https://www.digitalocean.com/community/tutorials/pytorch-vs-tensorflow`, abgerufen am 30. September 2025.

[27] `https://arxiv.org/abs/2407.17032`, abgerufen am 30. September 2025.

Initialisierung

In der Konstruktion des Agenten werden zunächst die Dimensionen des Zustands- und Aktionsraumes festgelegt, welche die Schnittstelle zwischen Umgebung und Agent definieren. Anschließend werden die Hyperparameter aus einer Konfigurationsdatei geladen. Dazu zählen unter anderem:

- die Lernrate α, die die Schrittweite für die Gewichtsaktualisierung des neuronalen Netzes steuert,
- die Größe des Replay Buffers N, der als Speicher für vergangene Erfahrungen dient,
- die Batchgröße B, die bestimmt wie viele Erfahrungen pro Trainingsschritt genutzt werden,
- die Frequenz, mit der das Training ausgelöst wird (*learn period*),
- die Parameter der Epsilon-Greedy-Strategie, bestehend aus ϵ (initiale Explorationsrate), einer Absenkungsrate ϵ_{decay} und einem Minimalwert ϵ_{min}.

Darüber hinaus wird das Q-Netzwerk initialisiert, das die Q-Werte approximiert. Es besteht aus mehreren vollständig verbundenen Schichten, deren Größe variabel gewählt werden kann. Für das Training werden der Adam-Optimierer sowie die mittlere quadratische Abweichung (*Mean Squared Error, MSE*) als Verlustfunktion verwendet. Zusätzlich wird ein Replay Buffer angelegt, der Erfahrungen in der Form

$$(s,a,r,s',\text{done})$$

speichert. Dieser Mechanismus reduziert Korrelationen zwischen aufeinanderfolgenden Zuständen und stabilisiert den Lernprozess.

Aktionswahl

Die Funktion `select_action` übernimmt die Auswahl der nächsten Aktion des Agenten. Dies geschieht auf Basis einer Epsilon-Greedy-Strategie. Mit einer Wahrscheinlichkeit ϵ wird eine zufällige Aktion gewählt. Dies dient der Exploration. Mit der Wahrscheinlichkeit $1-\epsilon$ wird hingegen diejenige Aktion ausgeführt, die das aktuelle Q-Netzwerk als optimal einschätzt (Exploitation).
Innerhalb der Funktion wird das neuronale Netz aufgerufen, um für den aktuellen Zustand s einen Q-Wert für jede mögliche Aktion zu berechnen. Die Aktion mit dem höchsten Wert wird gewählt, sofern nicht die explorative Alternative greift. Durch diesen Mechanismus werden bekannte Strategien verstärkt genutzt, während gleichzeitig neue Möglichkeiten erforscht werden.

Speicherung von Erfahrungen

Nach jeder Interaktion mit der Umgebung ruft der Agent die Funktion `step` auf. In dieser Funktion werden die beobachteten Daten – aktueller Zustand s, gewählte Aktion a, Belohnung r, Folgezustand s' sowie ein Indikator, ob die Episode abgeschlossen ist – in den Replay Buffer eingetragen. Der Buffer speichert diese Erfahrungen bis zur definierten Kapazität N. Wird diese überschritten ersetzt ein neuer Eintrag den ältesten. Dieser Mechanismus ermöglicht, dass das Training nicht ausschließlich auf hochkorrelierenden, unmittelbar aufeinanderfolgenden Erfahrungen basiert, sondern auf einem diversifizierten Pool von Daten.

Lernprozess

Der eigentliche Lernschritt wird durch die Funktion `__batch_temporal_difference_step` realisiert. Diese Funktion wird periodisch aufgerufen, abhängig vom Parameter `learn_period`. Zunächst wird ein zufälliges Batch von B Erfahrungen aus dem Replay Buffer gezogen. Diese Stichprobe ermöglicht es, dass Gradientenabstiege auf einer repräsentativen Teilmenge der Erfahrungen basieren.
Für jede dieser Erfahrungen wird der sogenannte TD-Fehler berechnet:

$$\delta = r + (1 - \text{done}) \cdot \gamma \cdot \max_{a'} Q(s',a';\theta) - Q(s,a;\theta). \tag{6.66}$$

Dieser Fehler beschreibt die Differenz zwischen der aktuellen Schätzung des Q-Wertes und dem durch die Belohnung sowie den Folgezustand bestimmten Zielwert. Die Funktion nutzt den Adam-Optimierer, um die Parameter θ des neuronalen Netzes so anzupassen, dass dieser Fehler minimiert wird. Dies geschieht über die Verlustfunktion, die die mittlere quadratische Abweichung des TD-Fehlers innerhalb des Batches misst:

$$L(\theta) = \frac{1}{B} \sum_{i=1}^{B} \left(\delta_i\right)^2. \tag{6.67}$$

Anpassung der Explorationsrate

Die Funktion `decay_epsilon` reguliert den Wert von ϵ nach jeder Episode. Dieser Wert wird multiplikativ mit dem Faktor ϵ_{decay} reduziert, jedoch nicht unter einen festgelegten Minimalwert ϵ_{min}. Formal ergibt sich:

$$\epsilon \leftarrow \max(\epsilon \cdot \epsilon_{\text{decay}}, \epsilon_{\text{min}}). \tag{6.68}$$

Dadurch verringert sich der Anteil zufälliger Aktionen im Verlauf des Trainings, sodass der Agent im späteren Verlauf vermehrt auf die bereits gelernten Strategien zurückgreift.

Speichern und Laden des Modells

Zur Sicherung des Trainingsfortschritts enthält der Agent Funktionen zum Speichern und Laden der trainierten Netzwerkgewichte. Dies ermöglicht es, ein Modell nach Abschluss einer Trainingsphase weiterzuverwenden oder erneut zu laden, um zusätzliche Episoden anzuschließen. Die Gewichte werden typischerweise in Dateien abgelegt, sodass sie unabhängig von der Laufzeitumgebung verfügbar sind.

Der Agent durchläuft in jeder Episode denselben Ablauf: Zunächst wird eine Aktion durch `select_action` gewählt, die resultierenden Erfahrungen werden durch `step` gespeichert, periodisch findet eine Trainingsphase mit `__batch_temporal_difference_step` statt und am Ende der Episode wird die Explorationsrate mittels `decay_epsilon` angepasst. Unterstützt wird der Prozess durch die Möglichkeit, Modelle zu speichern und wieder zu laden.

Die Verknüpfung dieser Funktionen stellt sicher, dass der Agent seine Strategie schrittweise verbessert und eine Balance zwischen dem Erkunden neuer Möglichkeiten und dem Ausnutzen bereits erlernter Erfahrungen findet.

6.4.4 Architektur und Funktionsweise des tiefen Q-Netzwerks (DQN)

Das Q-Netzwerk bildet das zentrale Element der Wertfunktionapproximation im Deep-Q-Learning. Es handelt sich hierbei um ein künstliches neuronales Netz, das die Abbildung von Zuständen s auf geschätzte Q-Werte $Q(s,a)$ für jede mögliche Aktion a realisiert. Auf diese Weise wird der Agent befähigt, ausgehend von einem gegebenen Zustand diejenige Aktion zu identifizieren, die nach der aktuellen Schätzung die höchste zu erwartende Belohnung generiert.

Netzwerkarchitektur

Die Architektur des Q-Netzes ist flexibel gestaltet und kann abhängig von den Hyperparametern variieren. In der Initialisierung werden bis zu drei vollständig verbundene Zwischenschichten (*Fully Connected Layers, fc*) erzeugt. Die Anzahl der Neuronen pro Schicht ist in den Hyperparametern `fc1`, `fc2` und `fc3` definiert. Falls einzelne Parameter auf den Wert `None` gesetzt sind, entfällt die entsprechende Schicht. Abbildung 6.28 stellt das DQN in einem vereinfachten Modell dar.

Damit lässt sich die Netzwerktiefe dynamisch zwischen ein und drei verborgenen Schichten variieren.

Jede Schicht wird durch eine lineare Abbildung der Form beschrieben:

$$z^{(l)} = W^{(l)} x^{(l-1)} + b^{(l)}. \tag{6.69}$$

Wobei $W^{(l)}$ die Gewichtsmatrix, $b^{(l)}$ der Biasvektor und $x^{(l-1)}$ die Eingaben der Schicht sind. Das Ergebnis $z^{(l)}$ wird durch eine Aktivierungsfunktion transformiert. Zusätzlich

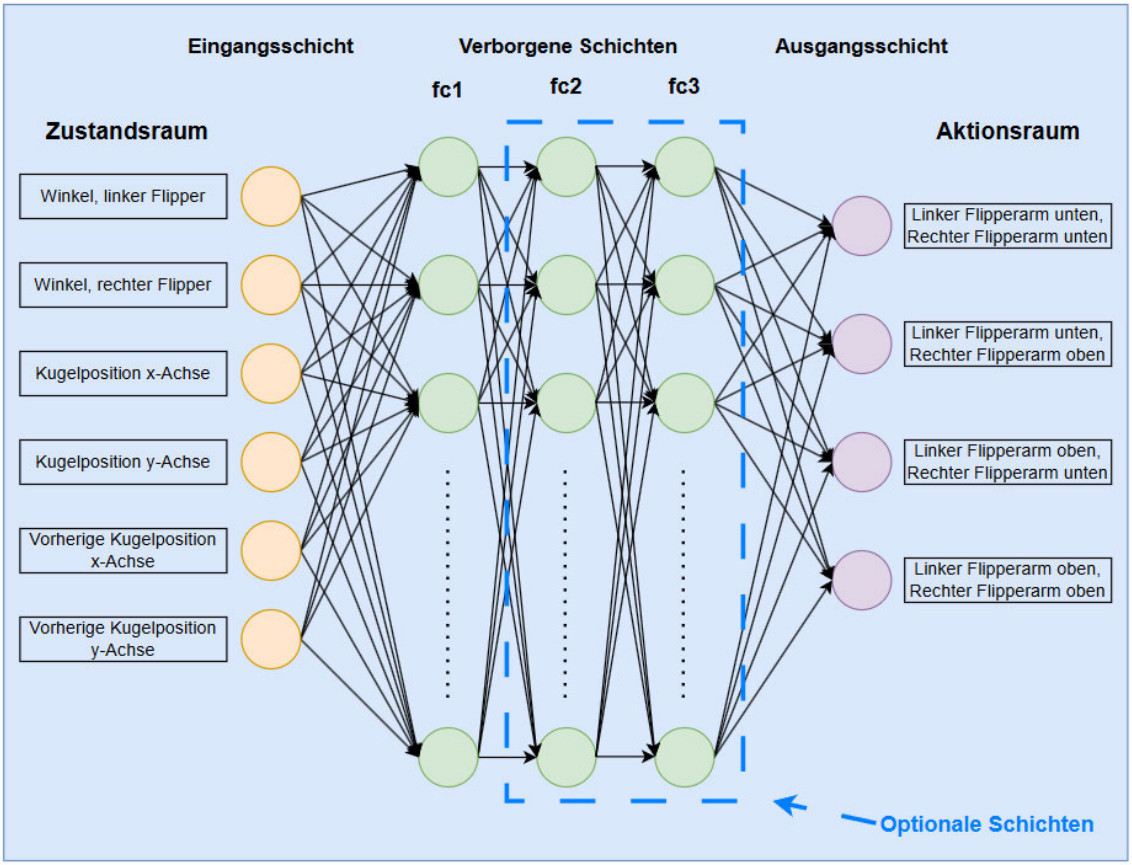


Abbildung 6.28: Vereinfachtes Modell des tiefen Q-Netzwerks

wird in jeder verborgenen Schicht eine Batch-Normalisierung durchgeführt, die die Ausgaben standardisiert und dadurch den Lernprozess stabilisiert. Die Ausgabe der letzten Schicht ist ein Vektor der Länge $|A|$, wobei $|A|$ die Anzahl der möglichen Aktionen beschreibt. Jedes Element dieses Vektors entspricht dem geschätzten Q-Wert für eine Aktion im gegebenen Zustand.

Gewichtsinitalisierung

Die Gewichte der linearen Schichten werden durch die He-Initialisierung gesetzt. Es handelt sich dabei um eine Initialisierung aus einer gleichverteilten Zufallsvariablen, die die Varianz der Aktivierungen über die Schichten hinweg stabil hält:

$$W^{(l)} \sim U\left[-\sqrt{\frac{6}{n_l}}, \sqrt{\frac{6}{n_l}}\right]. \tag{6.70}$$

Wobei n_l die Anzahl der Eingaben in Schicht l ist. Auf diese Weise wird verhindert, dass die Aktivierungen in tiefen Netzen entweder verschwinden oder explodieren.

Vorwärtsdurchlauf

Die Funktion `forward` beschreibt den Ablauf eines Vorwärtsdurchlaufs durch das Netz-

werk. Ein Eingabevektor $s \in \mathbb{R}^n$, der den Zustand der Umgebung repräsentiert, wird Schicht für Schicht transformiert. Abhängig von der Netzwerkgröße ergibt sich der Rechenweg:

$$x^{(1)} = \sigma\big(\mathrm{BN}_1(W^{(1)}s + b^{(1)})\big), \tag{6.71}$$

$$x^{(2)} = \sigma\big(\mathrm{BN}_2(W^{(2)}x^{(1)} + b^{(2)})\big), \tag{6.72}$$

$$x^{(3)} = \sigma\big(\mathrm{BN}_3(W^{(3)}x^{(2)} + b^{(3)})\big), \tag{6.73}$$

$$Q(s,\cdot) = W^{(L)}x^{(L-1)} + b^{(L)}, \tag{6.74}$$

wobei BN für Batch-Normalisierung steht und L die letzte Schicht kennzeichnet. Je nach Konfiguration entfallen eine oder mehrere Zwischenschichten, sodass sich eine variable tiefe Architektur ergibt. Das Ergebnis $Q(s,\cdot)$ stellt den Vektor aller Q-Werte dar, der die Grundlage für die Aktionswahl im Agenten bildet.

6.4.5 Hauptprogramm `TrainAgent`

Im Rahmen der Entwicklung wurde ein Programm erstellt, das die Ausführung des Agenten in einer virtuellen oder physischen Umgebung ermöglicht. Ziel war es, eine flexible Infrastruktur bereitzustellen, die sowohl das Training eines neuen Agenten als auch die Evaluation und die Fortführung bereits existierender Modelle unterstützt.
Zu Beginn werden die erforderlichen Arbeitsverzeichnisse für Modelle, Plots, Score-Daten und Hyperparameter automatisiert angelegt. Durch die Vergabe von Zeitstempeln für Dateien und Ergebnisse wird eine eindeutige Zuordnung der Daten gewährleistet, wodurch die Nachvollziehbarkeit und Reproduzierbarkeit der Experimente gesichert ist.
Das Programm kann in unterschiedlichen Betriebsmodi ausgeführt werden: Im `train`-Modus wird ein Agent von Grund auf trainiert, während der `continue_train`-Modus die Optimierung eines bestehenden Modells erlaubt. Der `evaluation`-Modus dient der Analyse und Validierung bereits trainierter Agenten. Zusätzlich kann zwischen einer virtuellen und einer physischen Umgebung gewählt werden, wodurch die entwickelten Algorithmen sowohl in Simulationen als auch an einem realen Demonstrator erprobt werden können. Abbildung 6.29 stell die Struktur des Hauptprogramms dar um die Abhängigkeiten klarzustellen.
Zur systematischen Auswertung werden pro Episode der erzielte Score und die Anzahl der Schritte aufgezeichnet. Nach Abschluss der Experimente wird das trainierte Modell inklusive seiner Gewichtungen gespeichert. Darüber hinaus erfolgt eine Speicherung der Ergebnisse sowohl in Form von Plots, die die Entwicklung des Scores über die Episoden visualisieren, als auch in CSV-Dateien, die eine detaillierte Analyse der Rohdaten ermöglichen. Ergänzend werden die verwendeten Hyperparameter-Konfigurationen gesichert,

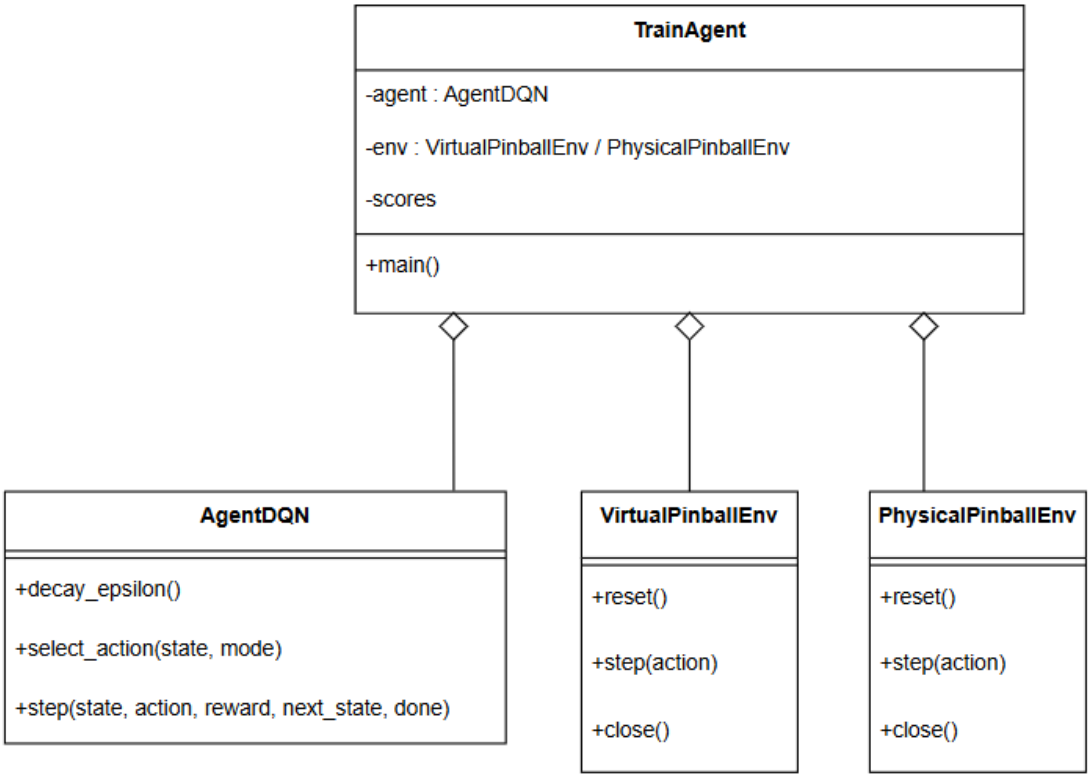


Abbildung 6.29: Vereinfachtes Klassendiagramm des Hauptprogramms `TrainAgent`

sodass eine konsistente Vergleichbarkeit zwischen unterschiedlichen Testläufen gewährleistet bleibt.
Durch diese Vorgehensweise wurde eine vollständige Experimentierumgebung geschaffen, die von der Initialisierung über die Durchführung bis hin zur Analyse alle notwendigen Schritte integriert und damit eine effiziente und reproduzierbare Untersuchung des RL-Agents ermöglicht.

6.4.6 Ermittelung der optimalen Belohnungen

Im Gegensatz zu modernen Flipperautomaten verfügt der physische Demonstrator nicht über ein integriertes Punktesystem. Daher ist es nicht möglich, den Agenten dafür zu belohnen, bestimmte Ziele mit der Kugel zu treffen. Stattdessen können lediglich die Dynamik der Kugel und die Aktivität der Flipperarme berücksichtigt werden. Auf dieser Grundlage lässt sich ein Belohnungssystem entwickeln, das sowohl das Verhalten der Kugel als auch der Flipperarme durch positive oder negative Rückmeldungen verstärkt oder abschwächt.
In den folgenden Abschnitten wird die schrittweise Entwicklung dieses Belohnungssystems in der virtuellen Umgebung erläutert.

Erste Belohnungsansätze

Die ersten Belohnungsansätze zielten darauf ab, dass die Flipperarme die Kugel in die

obere Hälfte des Spielfeld schlagen und negative Belohnungen für die Niederlage gegeben werden
Da eine Niederlage die das Ende des Spiels bedeuten würde und keine weiteren Belohnungen errungen werden können, wurde für eine bestimmte Höhe unterhalb des erlaubten Spielfeld eine Grenze bestimmt, die die Niederlage bestimmt und eine hohe negative Belohnung setzt.
Als Reaktion auf dieses Belohnungssystem hat der Agent die Strategie entwickelt, die Flipperarme dauerhaft auf Anschlag zu lassen, damit es nicht zu der Situation komm die negative Belohnung der Niederlage zu erhalten.
Abgeleitet aus dieser Strategie wollte der Agent eher negative Belohnungen vermeiden als positive zu erlangen.

Anreize durch positive Belohnungen

Ausgangspunkt war die Modifikation der Belohnungsstruktur, indem negative Belohnungen reduziert und stattdessen höhere positive Belohnungen vergeben wurden, um den Agenten dazu zu motivieren, die Kugel in den oberen Bereich des Spielfelds zu schlagen. Zusätzlich wurde eine Staffelung entlang der Y-Achse eingeführt, sodass die Belohnung proportional zur erreichten vertikalen Position der Kugel anstieg.
Diese Anpassung führte dazu, dass die Flipperarme verstärkt versuchten, die Kugel in den oberen Spielfeldbereich zu befördern. Dadurch konnten höhere durchschnittliche Belohnungspunkte pro Episode erzielt werden. Gleichzeitig zeigte sich jedoch, dass die Flipperarme in unregelmäßigen und teilweise willkürlichen Mustern aktiviert wurden, was nicht dem angestrebten, realistischen Spielverhalten entsprach.

Reduzierung der Aktivität der Flipperarme

Ziel der weiteren Anpassungen war es, die Aktivität der Flipperarme zu reduzieren und ein natürlicheres sowie rationaleres Spielverhalten zu erreichen. Hierzu wurde der aktuelle Winkel der Flipperarme in den Zustandsraum aufgenommen, wobei lediglich die Information berücksichtigt wurde, ob sich ein Arm in oberer oder unterer Position befand. Um die übermäßige Nutzung der Flipperarme einzuschränken, wurde eine negative Belohnung vergeben, sobald diese ausgelöst wurden.
Das Resultat war jedoch eine vollständige Inaktivität der Flipperarme. Dies deutet darauf hin, dass das Verhältnis zwischen der negativen Belohnung für die Aktivierung der Flipperarme und der Strafe für das Verlieren des Spiels unausgewogen war. Infolgedessen „entschied“ der Agent lieber den Spielverlust zu akzeptieren als durch die Aktivierung der Flipperarme zusätzliche negative Belohnungen zu riskieren.

Abhängigkeiten in der Belohnungsverteilung

Um diesem Problem entgegenzuwirken, wurde die negative Belohnung für das Auslö-

sen der Flipperarme reduziert und zusätzlich ein Zählermechanismus implementiert, der die Häufigkeit der Armaktivierungen überwachte. Wiederholte Ausführungen derselben Aktion, mit Ausnahme der Ruheposition, führten dabei zu einer progressiv steigenden negativen Belohnung, um irrationales Verhalten zu unterbinden. Darüber hinaus wurde die normalisierte Höhe der Kugel als skalierender Faktor in die Belohnungsfunktion integriert, sodass die Belohnung für die Position der Kugel abhängig von der Höhe variiert.

$$R(s)+= \begin{cases} 20 \cdot s_3, & \text{für } s_3 > 0.8 \\ 10 \cdot s_3, & \text{für } 0.6 < s_3 \leq 0.8 \\ 10 \cdot s_3, & \text{für } 0.3 < s_3 \leq 0.6 \\ (10 - 10 \cdot s_3) \cdot 10, & \text{für } s_3 \leq 0.3 \end{cases} \tag{6.75}$$

Die Formel 6.75 zeigt dass die Belohnung für die Position der Kugel eine Kombination aus Staffelung und Abhängigkeit darstellt. In Abbildung 6.30 wird verdeutlicht wie die einzelnen Zonen der Belohnungen eingerichtet sind. Folgende Formel 6.78 zeigt wie die Kalkulation des Flipperarmzählers bezüglich der Situtation berechnet wird:

$$L_{t+1} = \begin{cases} L_t + 1, & \text{falls } a_{t+1} = a_t, \\ 0, & \text{falls } a_{t+1} \neq a_t, \end{cases} \quad L_0 = 0 \tag{6.76}$$

$$R_{t+1} = \begin{cases} R_t + 1, & \text{falls } a_{t+1} = a_t, \\ 0, & \text{falls } a_{t+1} \neq a_t, \end{cases} \quad R_0 = 0. \tag{6.77}$$

Wobei L_t und R_t die Flipperzähler der jeweiligen Flipperarme respräsentieren.

$$R(a)+= \begin{cases} -L_t - R_t, & \text{für } a = 3 \\ -10, & \text{für } a \in \{1,2\} \\ 10,\ L_t \ \& \ R_t = 0 & \text{für } a = 0 \end{cases} \tag{6.78}$$

Auf diese Weise entstand ein komplexes Belohnungssystem, das sowohl die Position der Kugel als auch die Rationalität des Flipperverhaltens berücksichtigt.
Als Resultat konnte in der virtuellen Umgebung ein rationales Spielverhalten der Kugel

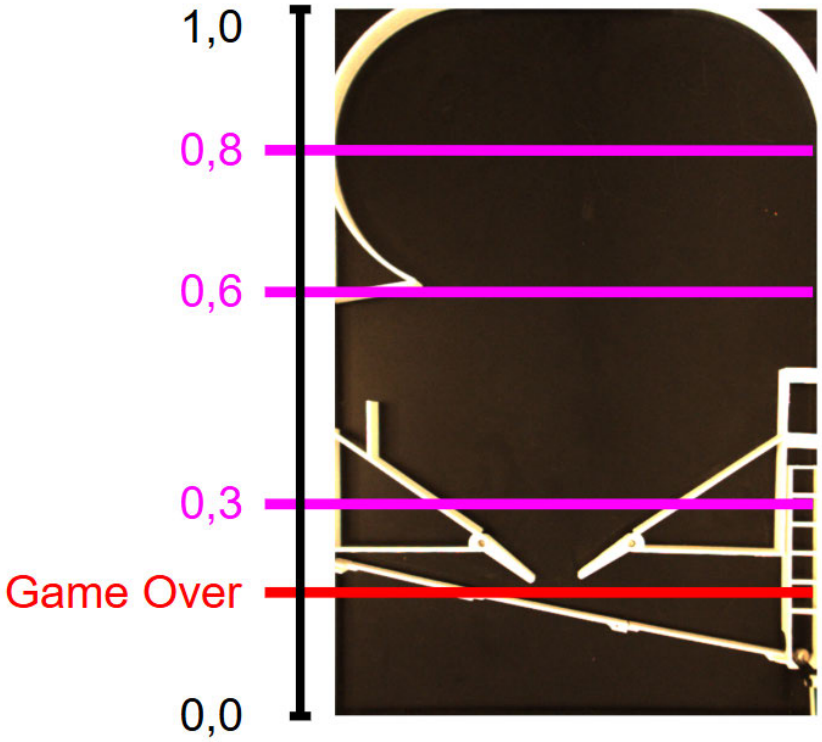


Abbildung 6.30: Konzept der Staffelung, markierte Bereiche auf dem Spielfeld

erreicht werden. Der Abfrageablauf des Belohnungssystems wird in Abbildung 6.31 dargestellt.

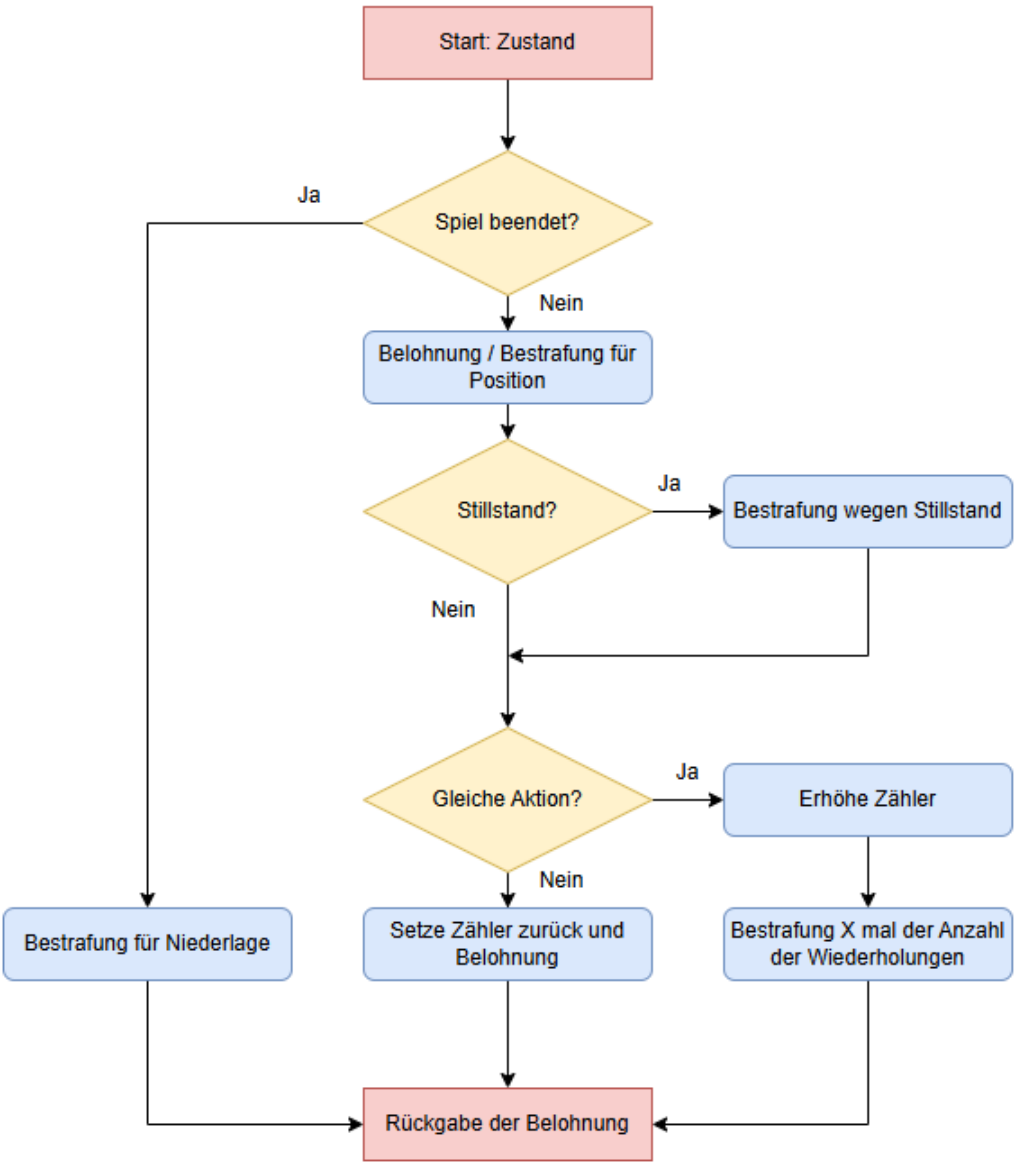


Abbildung 6.31: Ablaufdiagramm des Belohnungssystem

6.4.7 Tests der RL-Parameterbestimmung

Nachdem ein stabiles Belohnungssystem entwickelt wurde, besteht die Aufgabe den Agenten und das tiefe Q-Netzwerk zu parametrieren. Hierfür werden die virtuellen Trainingseinheiten mit je 1000 Episoden durchgeführt.
Um den DQN-Agenten systematisch zu optimieren, wurden unterschiedliche Tests durchgeführt. Jeder Test fokussiert sich auf die Variation eines einzelnen Parameters, während die übrigen Hyperparameter konstant gehalten wurden. Auf diese Weise konnte der isolierte Einfluss des jeweiligen Parameters auf das Lern- und Spielverhalten des Agenten untersucht werden.
Die folgenden Starthyperparameter wurden, sofern nicht explizit als Testparameter verän-

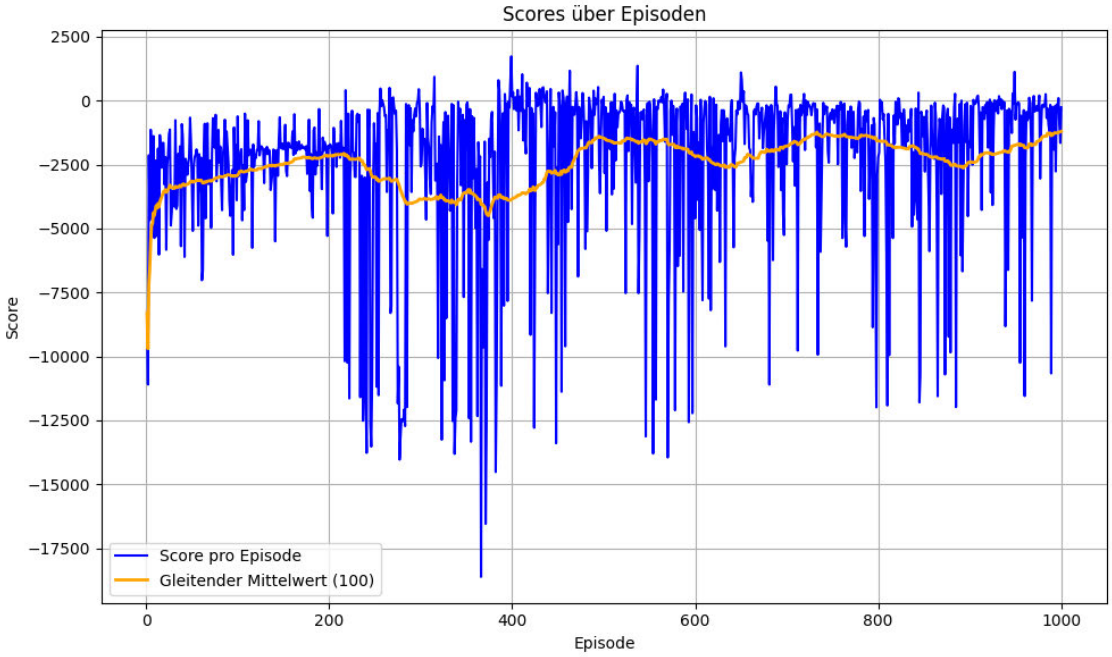


Abbildung 6.32: Erster Testdurchlauf eines Trainings mit den Starthyperparametern

dert, bei allen Versuchen beibehalten:

- Lernrate: 0,001
- Batchgröße: 64
- Lernperiode: 1
- Discountfaktor γ: 0,99
- ϵ_{min}: 0.05
- ϵ_{decay}: 0.995
- Verlustfunktion: MSE
- Aktivierungsfunktion: Leaky ReLU
- Schichtenmodell: 128, 128, 128
- Anzahl Episoden pro Trainingseinheit: 1000

Diese Hyperparameter haben sich in ersten Probeanläufen als stabil erwiesen und konnten mit ähnlichen Werten ebenfalls auch in einer vergleichbaren Masterarbeit von Sandra Verena Lassahn als robust und sinnvoll betrachtet werden [Las24]. Abbildung 6.32 zeigt den ersten Trainingsversuch in der virtuellen Umgebung mit den Startparametern.
Zur einheitlichen Bewertung der Ergebnisse wurden in den Tabellen Symbole verwendet, die die Qualität und Stabilität des Trainings beschreiben:

- – – : Kein Trainingserfolg erkennbar
- – : Schwache oder unzureichende Reaktionen
- o : Teilweise sinnvolle Aktionen, aber nicht stabil
- + : Gutes Ergebnis, erkennbar brauchbare Strategie
- ++ : Sehr gutes Ergebnis, stabile und effektive Strategie

Diese Darstellungsweise ermöglicht einen direkten Vergleich der Testreihen und hebt erfolgreiche Konfigurationen klar hervor.

Test der möglichen neuronalen Schichten im DQN

Um die optimale Architektur des DQN-Agenten zu bestimmen, wurden verschiedene Konfigurationen neuronaler Schichten getestet. Ziel der Tests war es, herauszufinden, welche Netzgröße eine effektive Steuerung der Flipperarme ermöglicht und gleichzeitig eine stabil nachvollziehbare Strategie im Spielverlauf entwickelt. In der folgenden Tabelle werden die durchgeführten Versuche sowie die beobachteten Resultate und Kommentare zusammengefasst.

Schichten	**Resultat**	**Kommentar**
32, 32	–	Es wurden keine rationalen Ausschläge erzeugt.
32, 32, 32	-	
64, 64	-	
128, 128	o	Die Flipperarme wurden teilweise zu spät ausgeführt.
128, 128, 128	o	
256, 256	+	Versuche die Kugel oben zu halten.
256, 256, 256	++	Es konnte eine Strategie der Flipperarme erkannt werden. Die Kugel gelang öfters in die obere Hälfte des Spielfelds

Tabelle 6.5: Ergebnisse für die Trainingstestdurchläufe mit verschiedenen verbogenen Schichten im DQN

Die Testergebnisse in Tabelle 6.5 zeigen, dass kleine Netzwerke mit 32 oder 64 Neuronen pro Schicht keine stabilen oder sinnvollen Aktionen erzeugen konnten. Erst ab einer Schichtgröße von 128 Neuronen wurden teilweise brauchbare Reaktionen auf das Spielgeschehen beobachtet. Die besten Ergebnisse erzielten Netzwerke mit 256 Neuronen pro

Schicht, insbesondere das dreischichtige Netzwerk (256, 256, 256), welches eine erkennbare Strategie der Flipperarme entwickelte und die Kugel häufiger in der oberen Spielfeldhälfte halten konnte. Daraus lässt sich ableiten, dass für die angestrebte Steuerungsqualität ein ausreichend tiefes und breites Netzwerk erforderlich ist, um die Komplexität des Flipper-Spiels abzubilden.

Verlustfunktionstest

Um die für das DQN-Training am besten geeignete Verlustfunktion zu identifizieren, wurden drei verschiedene Funktionen getestet: Huber Loss, Mean Absolute Error (MAE) und Mean Squared Error (MSE). Ziel war es, festzustellen, welche Funktion eine stabile und sinnvolle Lernentwicklung des Agenten ermöglicht. Die Ergebnisse der Testdurchläufe sind in Tabelle 6.6 zusammengefasst.

Verlustfunktionen	**Resultat**	**Kommentar**
Huber Loss	–	Keinerlei Erfolg beim Training
MAE	–	Ebenfalls kein Erfolg beim Training
MSE	++	Es konnten rationale Ergebnisse erzielt werden.

Tabelle 6.6: Ergebnisse für die Trainingstestdurchläufe mit drei Verlustfunktionen

Die Testdurchläufe zeigen eindeutig, dass Huber Loss und MAE für das vorliegende Flipper-Szenario keine erfolgreichen Trainingsergebnisse liefern konnten. Nur die Verwendung der MSE-Verlustfunktion ermöglichte eine sinnvolle Lernentwicklung des DQN-Agenten, sodass rationale Aktionen und Strategien erkannt werden konnten. Damit lässt sich ableiten, dass MSE als zentrale Verlustfunktion für das weitere Training des Agenten eingesetzt werden sollte.

Aktivierungsfunktionstest

Um die für das DQN-Training am besten geeignete Aktivierungsfunktion zu bestimmen, wurden drei gängige Funktionen getestet: ELU, ReLU und Leaky ReLU. Ziel war es zu untersuchen, welche Funktion eine stabile und effektive Steuerung der Flipperarme durch den Agenten ermöglicht. Die Ergebnisse der Testdurchläufe sind in Tabelle 6.7 zusammengefasst.

Aktivierungsfunktionen	**Resultat**	**Kommentar**
ELU	–	Keinerlei Erfolg beim Training
ReLU	o	Teilweise Flipperarme korrekt ausgelöst
Leaky ReLU	++	Die Flipperarme wurden häufig zu den richtigen Momenten ausgelöst.

Tabelle 6.7: Ergebnisse für die Trainingstestdurchläufe mit drei Aktivierungsfunktionen

Die Testergebnisse in Tabelle 6.7 zeigen, dass ELU für das vorliegende Flipper-Szenario keine erfolgreichen Aktionen erzeugte. ReLU führte zu teilweise korrektem Verhalten der Flipperarme, während Leaky ReLU die besten Ergebnisse lieferte: Die Flipperarme wurden häufig zu den richtigen Zeitpunkten ausgelöst und erlaubten eine effektive Steuerung des Spiels. Daraus lässt sich schließen, dass Leaky ReLU als bevorzugte Aktivierungsfunktion für das Training des DQN-Agenten eingesetzt werden sollte.

Test von Epsilon$_{min}$ und Epsilon$_{decay}$

Um das Verhalten des DQN-Agenten im Hinblick auf Exploration und Exploitation zu optimieren, wurden unterschiedliche Kombinationen der Parameter
ϵ_{min} und ϵ_{decay} getestet. Ziel war es, herauszufinden, welche Einstellungen eine effiziente Balance zwischen zufälligen Aktionen und gezieltem Spielverhalten ermöglichen. Die Ergebnisse der Testdurchläufe sind in Tabelle 6.8 dargestellt.

Epsilon$_{Decay}$	**Epsilon$_{min}$**	**Resultat**	**Kommentar**
0,95	0,10	- -	
0,95	0,05	- -	
0,95	0,01	-	Es wurden Reaktionen auf die Kugel gezeigt.
0,99	0,10	- -	Keine Reaktionen
0,99	0,05	o	Die Kugel erkannt und teilweise korrekt angeschlagen
0,99	0,01	o	
0,995	0,10	o	
0,995	0,05	++	Die Kugel häufig nach oben geschlagen.
0,995	0,01	+	
0,999	0,10	o	
0,999	0,05	+	Gezielte Schläge, mit guten Treffern.
0,999	0,01	+	

Tabelle 6.8: Ergebnisse für die Trainingstestdurchläufe mit unterschiedlichen Werten für Epsilon$_{min}$ und Epsilon$_{decay}$

Die Testergebnisse zeigen, dass zu niedrige Decay-Raten oder zu hohe Epsilon$_{min}$-Werte zu ungenügender Reaktion auf die Kugel führten. Eine zu schnelle Reduktion von Epsilon führte dagegen zu eingeschränkter Exploration und teils suboptimalen Aktionen. Die besten Ergebnisse wurden bei hohen Decay-Werten (0,995) und mittleren Epsilon$_{min}$-Werten (0,05) erzielt, bei denen der Agent gezielte Schläge ausführen und die Kugel effektiv nach oben bewegen konnte. Daraus lässt sich ableiten, dass eine sorgfältige Abstimmung von Epsilon$_{min}$ und Epsilon$_{decay}$ entscheidend für eine erfolgreiche Lernstrategie ist.

Lernperiod-/Batchsizetest

Um die Effizienz des Trainingsprozesses des DQN-Agenten zu optimieren, wurden unterschiedliche Kombinationen von Lernperiode und Batchsize getestet. Ziel war es zu bestimmen, welche Einstellungen ein stabiles und effektives Lernen ermöglichen und gleichzeitig die Trainingszeit effizient gestalten. Die Ergebnisse der Testdurchläufe sind in Tabelle 6.9 dargestellt.

Lernperiode	Batchsize	Resultat	Kommentar
1	32	o	
1	64	++	Erfolgreichstes Training
1	128	+	
2	32	-	
2	64	o	
2	128	+	

Tabelle 6.9: Ergebnisse für die Trainingstestdurchläufe mit unterschiedlichen Werten für die Lernperiode und die Batchsize

Die Testergebnisse der zeigen, dass eine kurze Lernperiode in Kombination mit einer mittleren Batchsize von 64 die besten Trainingsergebnisse liefert. Hier konnte der Agent stabile und effiziente Lernfortschritte erzielen. Größere Batchsizes führten teilweise zu geringfügig schlechteren Ergebnissen, während längere Lernperioden (z. B. 2) das Training insgesamt verlangsamen und teilweise zu weniger stabilen Ergebnissen führten. Daraus lässt sich ableiten, dass eine sorgfältige Wahl von Lernperiode und Batchsize entscheidend für den Trainingserfolg des DQN-Agenten ist.

Gammatest

Um den Einfluss des Discountfaktors γ auf das Lernverhalten des DQN-Agenten zu untersuchen, wurden verschiedene Werte getestet. Ziel war es, festzustellen, welcher Gamma-Wert eine optimale Balance zwischen kurzfristigen und langfristigen Belohnungen ermöglicht und damit eine stabile Spielstrategie unterstützt. Die Ergebnisse der Testdurchläufe sind in Tabelle 6.10 dargestellt.

Gamma	Resultat	Kommentar
0,98	- -	
0,985	- -	
0,99	++	Erfolgreichstes Training
0,995	+	Grundsätzlich ist eine Spielstrategie vorhanden, jedoch teilweise willkürliches Auslösen der Flipperarme

Tabelle 6.10: Ergebnisse für die Trainingstestdurchläufe für unterschiedliche Gammawerte

Die Testergebnisse verdeutlichen, dass zu niedrige Gamma-Werte (0,98 und 0,985) keine effektiven Spielstrategien hervorbrachten. Der Wert Gamma = 0,99 erzielte das erfolgreichste Training, mit einer stabilen und effektiven Spielstrategie. Ein etwas höherer Wert von 0,995 ermöglichte zwar grundsätzlich Spielstrategien, führte jedoch zu teilweise willkürlichem Auslösen der Flipperarme. Daraus lässt sich schließen, dass ein Gamma-Wert von 0,99 für die angestrebte Spielsteuerung des DQN-Agenten optimal ist.

6.4.8 Optimalste Hyperparameter für die virtuelle Umgebung

Die durchgeführten Tests in der virtuellen Umgebung haben gezeigt, dass bestimmte Kombinationen von Hyperparametern besonders stabile und erfolgreiche Trainingsergebnisse liefern. Als optimale Konfiguration ergaben sich:

- Neuronale Architektur: drei Schichten mit je 256 Neuronen
- Aktivierungsfunktion: Leaky ReLU
- Verlustfunktion: Mean Squared Error (MSE)
- $\text{Epsilon}_{min} = 0{,}05$ bei einem Epsilon_{decay} von 0,995
- Lernperiode = 1, Batchsize = 64
- Discountfaktor $\gamma = 0{,}99$

Diese Kombination zeigte eine klare Tendenz zu stabilen und effektiven Spielstrategien. Insbesondere die Architektur mit 256 Neuronen pro Schicht erlaubte es dem Agenten, die Komplexität der Umgebung zuverlässig abzubilden. Die Leaky ReLU-Aktivierung verhinderte das Problem „toter Neuronen“ und führte zu einer hohen Konsistenz bei den Aktionen der Flipperarme. Zudem stellte sich die MSE-Verlustfunktion als robusteste Wahl heraus, da sie eine stetige Verbesserung der Policy erlaubte.
Die Wahl von ϵ_{min} und ϵ_{decay} erwies sich als entscheidend für die Balance zwischen Exploration und Exploitation. Ein Epsilon_{decay} von 0,995 ermöglichte es dem Agenten, auch über längere Trainingsphasen hinweg ausreichend zu explorieren, während der Mindestwert von 0,05 verhinderte, dass das Verhalten zu früh deterministisch wurde. Ergänzt durch eine Batchsize von 64 und eine Lernperiode von 1 konnte der Agent eine effiziente und stabile Lernkurve entwickeln. Der Discountfaktor von 0,99 förderte langfristig orientierte Strategien, die für das Erreichen höherer Spielziele entscheidend sind.
Die in Abbildung 6.33 dargestellte Entwicklung der durchschnittlich erzielten Belohnungen über 1000 Episoden verdeutlicht, dass die genannten Parameter die besten Ergebnisse lieferten. Hierbei zeigt sich eine deutliche Steigerung der Performance, was die Wahl dieser Hyperparameterkombination bestätigt.

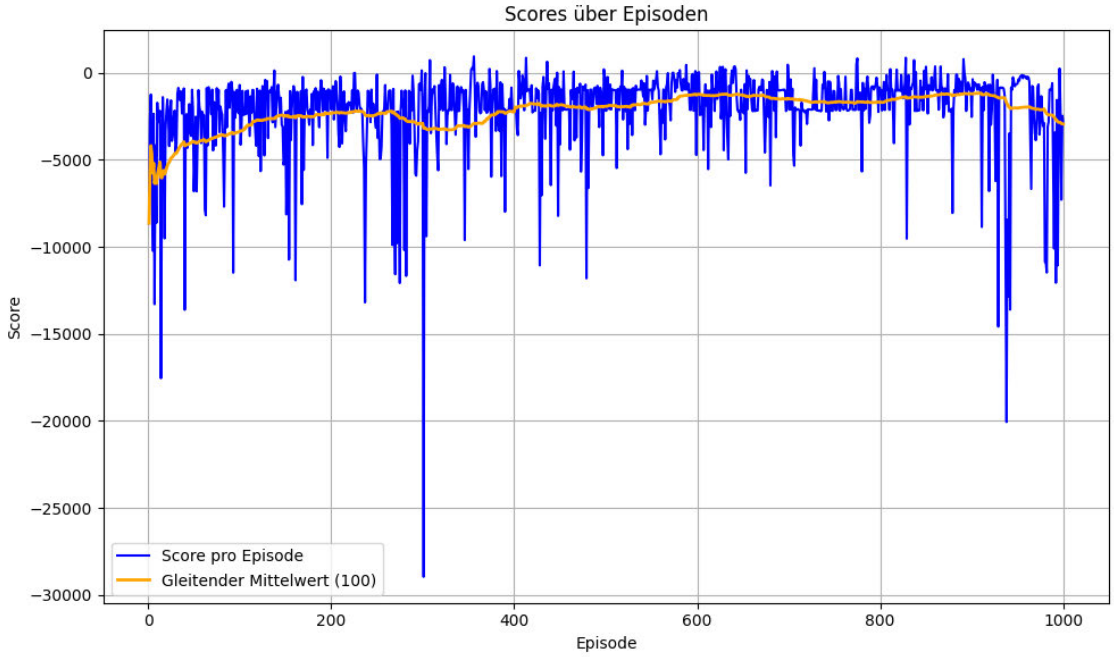


Abbildung 6.33: Erzielte Belohnungen über 1000 Episoden mit den besten Hyperparametern

6.4.9 Fortlaufendes Training in der physischen Umgebung

Nachdem der Agent über 1000 Episoden in der virtuellen Umgebung trainiert hat, konnte nun das beste Modell auf den physischen Demonstrator konvertiert werden. Das Training erfolgte über 500 Episoden. Abbildung 6.34 zeigt den Trainingsverlauf des physischen Demonstrators. Grundsätzlich lässt sich aus den Ergebnissen ableiten, dass das Training das Spielverhalten des Agenten stätig verbessert hat. Jedoch erkennt man am Verhalten des Flippers, dass es zeitliche Probleme gibt, mit denen der Agent nicht klar kommt. Zurückzuführen lässt sich das Problem an der Reaktionszeit der Motorsteuerung. Die Kommunikation bedarf zu viel Zeit und hindert den Agenten teilweise bei hoher Geschwindigkeit der Kugel, schnell genug reagieren zu können.

6.4.10 Probleme während des Trainings

Während des Trainings traten bei beiden Umgebungen technische Herausforderungen auf, die den Trainingsprozess beeinflussten. Zum einen kam es gelegentlich zu sogenannten Hardlocks: Der Ball wurde über einen längeren Zeitraum innerhalb des Flipperbereichs gehalten, da die Ballposition nicht korrekt aktualisiert wurde. Infolgedessen führte der Agent wiederholt die gleiche Aktion – den Flipper oben halten – aus, wodurch das System in einer Endlosschleife gefangen war und nur durch eine zeitliche Begrenzung befreit werden konnte. In Abbildung 6.34 beschreiben die starken negativen Spitzen solche Hard-

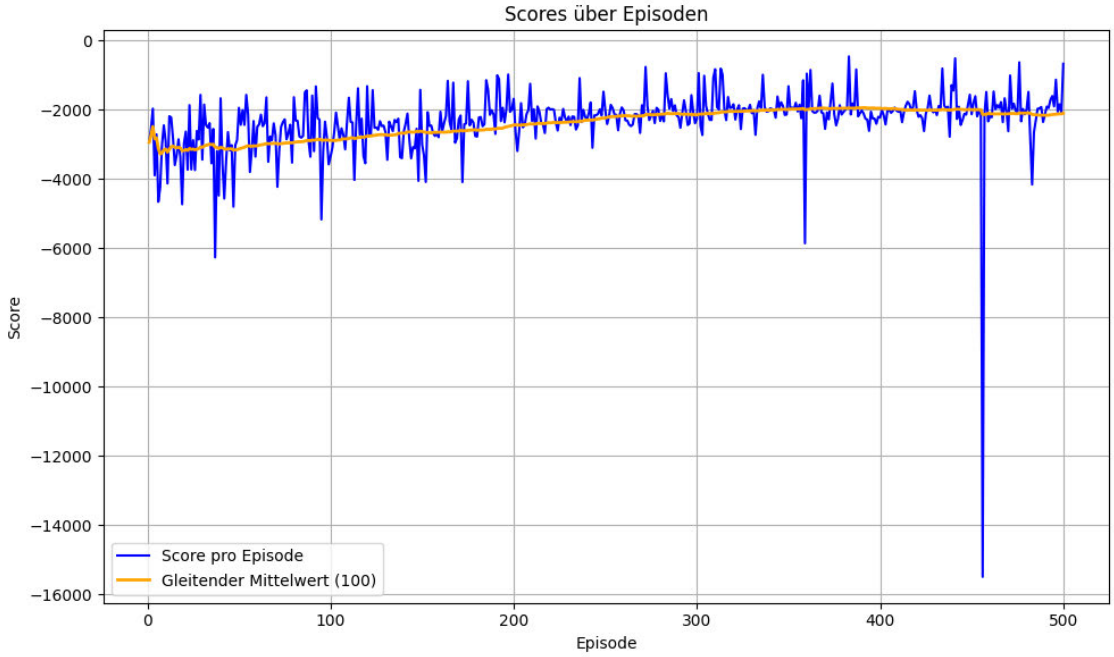


Abbildung 6.34: Erzielte Belohnungen über 500 Episoden mit den besten Hyperparametern am physischen Demonstrator

locks.

Ein weiteres Problem trat beim physischen Demonstrator auf. In seltenen Fällen rollte die Kugel nicht wie erwartet die Rückführbahn hinunter. Dies war darauf zurückzuführen, dass der Reibungsrollfaktor der Kugel größer war als die Hangkraft, wodurch die Kugel stehenblieb. In diesen Fällen war ein manuelles Eingreifen erforderlich, um den Spielablauf fortzusetzen.

Beide Probleme verdeutlichen die Grenzen sowohl der digitalen Simulation als auch der physischen Umsetzung des Flippers. Während die KI durch ungenaue Positionsupdates in Entscheidungsfallen geraten konnte, zeigte die reale Mechanik, dass physikalische Faktoren wie Reibung und Gravitation entscheidenden Einfluss auf das Verhalten der Kugel haben.

6.4.11 Allgemeines Ergebnis des Trainings am physischen Demonstrator

Während das Training des Agenten in der virtuellen Simulationsumgebung sehr gute Ergebnisse erzielt hat, zeigte sich bei der Übertragung auf den physischen Demonstrator ein deutliches Leistungsdefizit. Hauptursächlich hierfür sind die Verzögerungen in der Bildverarbeitung sowie in der Ausgabe der berechneten Aktionen. Die notwendige Verarbeitung der Kamerabilder beansprucht Zeit, wodurch die Entscheidungsfindung des Agenten im Vergleich zur Geschwindigkeit der Kugel oftmals zu langsam ist. Dies führt dazu,

dass der Agent in realen Spielsituationen nicht zuverlässig reagieren kann und die Kugel häufiger verloren geht. Damit wird deutlich, dass die Reaktionsfähigkeit des Systems ein zentrales Problem darstellt, das sowohl durch effizientere Algorithmen als auch durch leistungsfähigere Hardware adressiert werden muss.

7 Evaluierung des Systems

7.1 Anforderungsabgleich

In den folgenden Abschnitten werden die Anforderungen aus der Anforderungsanalyse erneut besprochen und überprüft ob diese erfüllt wurden.

7.1.1 Anforderungsabgleich der Bildverarbeitung (BV)

Funktionale Anforderungen (F)

BV-F1: *Erfüllt* Das Spielfeld wird durch die Klasse `FieldCropper` aus dem Rohbild ausgeschnitten. Danach wird es von der Funktion `field_transform` in das korrekte Seitenverhältnis transformiert.

BV-F2: *Erfüllt* Die Klasse `FieldCropper` ermöglicht es mit manueller Eingabe das Spielfeld auszuschneiden und den exakten Interessensbereich abzudecken.

BV-F3: *Erfüllt* Durch die Funktion `detect_green_in_bottom_right()` wird eine grüne Fläche auf dem Startarm erkannt. Es wird zwar nicht die genaue Position erkannt, dennoch erfüllt es die Notwendigkeit zu erkennen, ob sich der Startarm in einer blockierenden Position befindet oder nicht.

BV-F4: *Erfüllt* Die Winkel werden in der Kalibrierungsphase durch die Funktion `detect_flipper_arms` erkannt und zur Verfügung gestellt.

BV-F5: *Teilweise erfüllt* Über die Höhe der Kugel auf dem Spielfeld wird abgeleitet, ob das Spiel weitergeht. Zwar wird der Drain nicht erkannt, jedoch ist über die Höhe der Kugel ableitbar, ob diese in den Bereich unterhalb der Flipperarme gelangt ist. Im Training war es somit möglich zu erkennen, ob eine Spielrunde beendet wurde und eine neue Episode gestartet werden kann.

BV-F6: *Erfüllt* Die Schleife `t_detect_flipper_ball()` erkennt bei neuen Bilddaten die Kugel mit einer Berechnungszeit von durchschnittlich 8,996 ms.

Nicht-funktionale Anforderungen (NF)

BV-NF1: *Erfüllt* Die Belichtungszeit der Kamera wurde auf 10 ms gesetzt. Die maximale Bildrate, welche die Kamera aufnehmen kann liegt bei 61 FPS. Die Aufnahmezeit beträgt durchschnittlich 16,895 ms. Hieraus lässt sich schließen, dass die Kamera für eine Bildrate von 59 Bildern pro Sekunde neben der Belichtungszeit rund 6,895 ms für weitere Operationen benötigt.

BV-NF2: *Erfüllt* Durch Filteranwendungen wie den Gaussfilter, werden Rauscheffekte entfernt und fehlerhafte Bildartefakte vermieden. In der Funktion `t_detect_flipper_ball()` wird eine Stabilisierung durch ein Durchschnittsbild erzeugt. Es sorgt dafür, dass starke Veränderungen des Bildes, beispielsweise durch Vibrationen der Motoren, abgeschwächt werden.

7.1.2 Anforderungsabgleich der virtuellen Umgebung (VU)

Funktionale Anforderungen (F)

VU-F1: *Erfüllt* Durch die entwickelte *Physik-Engine* ist das Verhalten der virtuellen Kugel realitätsgetreu.

VU-F2: *Erfüllt* Nachdem die Kugel unter den Flipperarmen gelangt wird das System durch die `reset()`-Funktion wieder in den Anfangszustand zurückgesetzt.

VU-F3: textitErfüllt Durch die entwickelte Physik-Engine ist das Verhalten der virtuellen Kugel realitätsgetreu. Die Kollisionen mit statischen Objekten wurde durch eine Kollisionskarte realisiert, um Berechnungszeit zu sparen. Kollisionen mit den Flipperarmen werden mit jedem Zeitschritt überprüft und umgesetzt.

VU-F4: *Erfüllt* Die Klasse `RenderScene` ermöglicht es im Rendermodus eine Szene zu erzeugen um eine Nachvollziehbarkeit für den Anwender zu schaffen. Die Dimensionierung wurde an den physischen Demonstrator angepasst.

VU-F5: *Erfüllt* Beim Neustart wird die Kugel auf die Startposition mit einem zufälligem Offset von [-0,05, 0,05] gesetzt. Die Startgeschwindigkeit erhält einen zufälligen Offset von [0, 10]

Nicht-funktionale Anforderungen (NF)

VU-NF1: *Erfüllt* Die gesamte virtuelle Umgebung wurde in Python implementiert.

VU-NF2: *Erfüllt* Die virtuelle Kugel wurde nach der realen Beschaffenheit modelliert. Sie besitzt einen Radius von 0,635 cm.

VU-NF3: *Erfüllt* Die Neigung des physischen Demonstrator von 7° wurde in die *Physik-Engine* implementiert und erzeugt eine entsprechend Hangkraft die auf das Rollverhalten der Kugel wirkt.

VU-NF4: *Erfüllt* Durch das Erstellen der Kollisionskarte wurde ein Spielfeld erzeugt, welches den Vorgaben der physischen Umgebung entspricht. Die Kollisionskarte beinhaltet alles statischen Objekte, die im physischen Demonstrator verbaut sind.

VU-NF5: *Nicht erfüllt* Die echte Rotationsgeschwindigkeit konnte nicht ermittelt werden. Für die virtuelle Umgebung wurde eine Winkelgeschwindigkeit von 2 °/ms implementiert. Diesem Wert wurde sich in der Entwicklungsphase angenähert. Schließlich konnte dieser Wert als ausreichend festgestellt werden, da dieser keine Benachteiligung bezüglich der Konvertierung auf den physischen Demonstrator aufweist.

7.1.3 Anforderungsabgleich der künstlichen Intelligenz (KI)

Funktionale Anforderungen (F)

KI-F1: *Erfüllt* Durch das gleiche Interface der beiden Umgebungen ist es möglich den Agenten, welcher in der virtuellen Umgebung trainiert wurde, in der physischen Umgebung weiter zu trainieren.

KI-F2: *Erfüllt* Es ist möglich im Hauptprogramm einen von drei Modi auszuwählen. `train`, `continue_train` und `evaluation` erlauben es zwischen dem Training eines neuen Modells, dem Training eines bereits bestehenden Modells und dem Testen von trainierten Modellen zu entscheiden.

Nicht-funktionale Anforderungen (NF)

KI-NF1: *Erfüllt* Für das Training des Systems wurde die Methode des Reinforcement Learnings angewandt.

KI-NF2: *Erfüllt* Sowohl die virtuelle Umgebung als auch die physische weisen einen Zustandsraum von sechs Zuständen und und einen Aktionsraum von vier verschiedenen Aktionen auf. Beide Umgebungen erhalten im Zustandsraum die Winkel der Flipperarme, die aktuelle Position der Kugel und die vorherige Position der Kugel. Der Aktionsraum beinhaltet die Steuerbefehle der Motorsteuerung, welche die Flipperarme ansteuern sollen.

KI-NF3: *Erfüllt* Die virtuelle und die physische Umgebung besitzen das gleiche Interface. Es bietet eine Schrittfunktion, eine Resetfunktion und eine Renderfunktion.

KI-NF4: *Erfüllt* Der Agent wird mittels eines DQN und der Epsilon-Greedy-Strategie trainiert. Es ist möglich die Hyperparameter des Netzwerks über eine YAML-Datei zu ändern.

7.1.4 Anforderungsabgleich des physischen Demonstrator (PD)

Funktionale Anforderungen (F)

PD-F1: *Erfüllt* Durch die bereits vorhandene Motorsteuerung ist es möglich die Flipperarme durch eine Aktion des Agenten zu bewegen. Die ausgewählte Aktion aus dem DQN wird direkt an die Motorsteuerung weitergeleitet.

PD-F2: *Erfüllt* Die Rückführungsrampe sorgt dafür, dass die Kugel sobald das Spiel verloren wurde direkt in die Startrampe gelangt. Es wurden kleine Schleifarbeiten getätigt, damit der Durchlauf zur Startrampe reibungsloser verläuft. Nur in extrem seltenen Fällen gelang die Kugel nicht zurück in die Startrampe.

PD-F3: *Erfüllt* Nach jeder Niederlage werden die Winkel der Flipperarme in einer Kalibrierungsphase untersucht und, wenn notwendig, korrigiert. Der Startarm wird während der Kalibrierungsphase schrittweise so lange um 10 Schritte bewegt, bis eine garantiert nicht-blockierende Position erreicht wurde.

Nicht-funktionale Anforderungen (NF)

PD-NF1: *Erfüllt* Die Winkel auf denen die Kalibrierung eingestellt ist, sind fest auf die in der Anforderung gewählten Werte abgestimmt. Die Flipperarme erreichen somit immer die gewünschten Winkel mit einer einstellbaren Toleranz.

7.1.5 Zusammenfassung der Evaluierung

Die Evaluierung des entwickelten Systems zeigt, dass die in der Anforderungsanalyse definierten funktionalen und nicht-funktionalen Anforderungen in weiten Teilen erfolgreich erfüllt werden konnten. In der Bildverarbeitung wurden die geforderten Funktionen wie das Zuschneiden und Transformieren des Spielfeldes, die Erkennung des Startarms, die Winkelbestimmung der Flipperarme sowie die Kugelerkennung umgesetzt. Lediglich bei der Drain-Erkennung ergaben sich Einschränkungen, die jedoch im Trainingskontext kompensiert werden konnten. Die nicht-funktionalen Anforderungen, insbesondere bezüglich Bildrate und Stabilität der Erkennung, wurden ebenfalls erreicht.
Auch die virtuelle Umgebung erfüllte nahezu alle Anforderungen. Die eigens entwickelte Physik-Engine erlaubte ein realitätsnahes Verhalten der Kugel sowie eine effiziente Berechnung von Kollisionen. Durch das Rendern und die Anpassung an die Dimensionen des physischen Demonstrators konnte eine hohe Nachvollziehbarkeit erzielt werden. Bis auf die exakte Abbildung der realen Rotationsgeschwindigkeit, die nur angenähert werden konnte, wurden alle funktionalen und nicht-funktionalen Anforderungen eingehalten.
Im Bereich der künstlichen Intelligenz konnte bestätigt werden, dass der Agent sowohl in der virtuellen als auch in der physischen Umgebung eingesetzt werden kann. Unterschied-

liche Modi für Training, Weitertraining und Evaluation ermöglichten eine flexible Nutzung des Systems. Die Verwendung eines DQN in Verbindung mit der Epsilon-Greedy-Strategie und der konfigurierbaren Hyperparameter stellte sicher, dass das System lernfähig und anpassbar bleibt.
Der physische Demonstrator erfüllte ebenfalls die gesetzten Ziele. Die Motorsteuerung reagierte zuverlässig auf die Aktionen des Agenten. Die Reaktionszeiten sind jedoch durch die Kommunikation mit der Motorsteuerung ein Hindernis und verursachen teilweise zeitliche Probleme. Die Rückführungsrampe stellte einen kontinuierlichen Spielablauf sicher und die Kalibrierungsmechanismen garantierten die Funktionsfähigkeit der Flipperarme über mehrere Spielzyklen hinweg. Auch die nicht-funktionalen Anforderungen im Hinblick auf Präzision und Wiederholgenauigkeit wurden erfüllt.
Insgesamt belegt die Evaluierung, dass das entwickelte Gesamtsystem aus Bildverarbeitung, virtueller Umgebung, künstlicher Intelligenz und physischem Demonstrator die gestellten Anforderungen weitgehend erfüllt und damit die Funktionsfähigkeit des KI-gesteuerten Flipperautomaten nachweist.

8 Fazit und Ausblick

Im Rahmen dieser Masterarbeit wurde erfolgreich ein KI-gesteuerter Flipperautomat entwickelt, der auf Basis von Deep Reinforcement Learning in Kombination mit Bildverarbeitung selbstständig agieren kann. In der virtuellen Trainingsumgebung zeigte der entwickelte Agent vielversprechende Ergebnisse: Er konnte stabile Strategien erlernen, um die Kugel über längere Zeit im Spiel zu halten und sich an unterschiedliche Spielsituationen anzupassen. Dies verdeutlicht, dass die gewählte Methodik grundsätzlich geeignet ist, um komplexe Steuerungsaufgaben zu bewältigen.

Bei der Übertragung auf den physischen Demonstrator traten jedoch deutliche Herausforderungen zutage. Die Bildverarbeitung sowie die Ausgabe der Aktionen führten zu Verzögerungen, die in einer spürbar reduzierten Reaktionsgeschwindigkeit des Agenten resultierten. Dadurch war der Agent teilweise nicht in der Lage, schnell genug auf die Bewegungen der Kugel zu reagieren, was seine Effektivität im realen Spielbetrieb einschränkte. Die Arbeit zeigt somit klar auf, dass die Diskrepanz zwischen Simulation und Realität maßgeblich von technischen Faktoren wie Verarbeitungsgeschwindigkeit und Echtzeitfähigkeit beeinflusst wird.

Für zukünftige Arbeiten bietet es sich daher an, die Bildverarbeitung zu optimieren und durch effizientere Algorithmen oder leistungsfähigere Hardware zu beschleunigen. Ebenso könnten Verfahren eingesetzt werden, die Entscheidungen prädiktiver treffen, um die Verzögerung in der Aktionsausgabe zu kompensieren. Auf diese Weise ließe sich die Lücke zwischen der guten Performance in der Simulation und der eingeschränkten Reaktionsfähigkeit im physischen System verkleinern. Darüber hinaus könnten alternative RL-Methoden mit kürzeren Berechnungszeiten getestet oder hybride Ansätze mit klassischen Steuerungsverfahren kombiniert werden. So eröffnet die Arbeit trotz bestehender Einschränkungen einen wertvollen Ausgangspunkt für weitere Forschung im Bereich echtzeitfähiger KI-Systeme für physische, dynamische Umgebungen.

9 Danksagung

An dieser Stelle möchte ich meinen herzlichen Dank an all diejenigen aussprechen, die zum Gelingen dieser Masterarbeit beigetragen haben. Mein besonderer Dank gilt meinem Betreuer, Prof. Dr. Marc Hensel, der mir jederzeit mit wertvollen Anregungen, hilfreichen Ratschlägen und konstruktiver Kritik zur Seite stand. Seine Offenheit für Fragen und seine Unterstützung in allen Phasen der Arbeit haben maßgeblich dazu beigetragen, dass dieses Projekt erfolgreich umgesetzt werden konnte.
Ebenso danke ich meiner Verlobten, die mir in den intensiven Arbeitsphasen mit unermüdlicher Geduld, Verständnis und Rückhalt zur Seite stand. Ohne ihre Unterstützung wäre die Fertigstellung dieser Arbeit in dieser Form nicht möglich gewesen. Meiner Familie gebührt mein Dank für ihre stetige Ermutigung und ihr Vertrauen in mich. Ihre beständige Unterstützung war für mich Motivation und Kraftquelle zugleich.

Literaturverzeichnis

[Bis06] Christopher M. Bishop. *Pattern Recognition and Machine Learning*. Springer, 2006. ISBN: 9780387310732.

[BZ20] Phil W. Brown und Laura G. Zai. *Einstieg in Deep Reinforcement Learning*. München: Carl Hanser Verlag, 2020.

[BZB20] Jonas Buehler, Jan Zimmermann und Wolfram Burgard. „Deep Reinforcement Learning for Physical Pinball Machines“. In: *arXiv preprint arXiv:2006.14059* (2020).

[Cho+14] Kyunghyun Cho u. a. „Learning phrase representations using RNN encoder-decoder for statistical machine translation“. In: *arXiv preprint arXiv:1406.1078* (2014).

[Dom12] Pedro Domingos. „A few useful things to know about machine learning“. In: *Communications of the ACM* 55.10 (2012), S. 78–87.

[DY20] Li Deng und Dong Yu. „A review of deep learning with special emphasis on architectures, applications and recent trends“. In: *Knowledge-Based Systems* (2020).

[Fou25] OpenCV Foundation. *About OpenCV*. `https://opencv.org/about/`. Abgerufen am 15. September 2025. 2025.

[GB10a] Xavier Glorot und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*. 2010, S. 249–256.

[GB10b] Xavier Glorot und Yoshua Bengio. „Understanding the difficulty of training deep feedforward neural networks“. In: *Proceedings of the thirteenth international conference on artificial intelligence and statistics*. 2010, S. 249–256.

[GBC16a] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. MIT Press, 2016. ISBN: 9780262035613.

[GBC16b] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. MIT Press, 2016.

[He+15] Kaiming He u. a. „Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification“. In: *Proceedings of the IEEE international conference on computer vision*. 2015, S. 1026–1034.

[Hen+18] Peter Henderson u. a. „Deep reinforcement learning that matters“. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 32.1 (2018).

[HS97] Sepp Hochreiter und Jürgen Schmidhuber. „Long short-term memory“. In: *Neural Computation* 9.8 (1997), S. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

[IS15] Sergey Ioffe und Christian Szegedy. „Batch normalization: Accelerating deep network training by reducing internal covariate shift“. In: *International conference on machine learning*. PMLR. 2015, S. 448–456.

[KB15] Diederik P. Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: *arXiv preprint arXiv:1412.6980* (2015).

[Ken01] Marco Rossignoli Kent. *The Complete Pinball Book: Collecting the Game and Its History*. Schiffer Publishing, 2001.

[Las24] Sandra Verena Lassahn. „3D-Simulation und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Labyrinths“. Betreuer: Prof. Dr.-Ing. Marc Hensel, Zweitgutachterin: Prof. Dr. rer. nat. Ulrike Herster. Masterarbeit. Hamburg: Hochschule für Angewandte Wissenschaften Hamburg, Sep. 2024.

[LBH15a] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. „Deep learning“. In: *Nature* 521.7553 (2015), S. 436–444. DOI: 10.1038/nature14539.

[LBH15b] Yann LeCun, Yoshua Bengio und Geoffrey Hinton. „Deep learning“. In: *Nature* 521.7553 (2015), S. 436–444. DOI: 10.1038/nature14539.

[LeC+12] Yann LeCun u. a. „Efficient BackProp“. In: *Neural networks: Tricks of the trade* (2012), S. 9–48.

[MDP21] M. P. Mukhina, V. Yu. Derkach und A. P. Prymak. „Combination of Hough Transform and Canny Edge Detector for Improvement of Linear Object Detection Results“. In: *Electronics and Control Systems* (2021). URL: https://jrnl.nau.edu.ua/index.php/ESU/article/view/13515.

[Mir+16] Azalia Mirhoseini u. a. „Learning to Play Pinball with Machine Learning Algorithms“. In: *Proceedings of the AAAI Conference on Artificial Intelligence* 30.1 (2016).

[Mni+15] Volodymyr Mnih u. a. „Human-level control through deep reinforcement learning“. In: *Nature* 518.7540 (2015), S. 529–533.

[MRT18] Mehryar Mohri, Afshin Rostamizadeh und Ameet Talwalkar. *Foundations of Machine Learning*. 2. Aufl. MIT Press, 2018. ISBN: 9780262039406.

[NH10] Vinod Nair und Geoffrey E. Hinton. „Rectified linear units improve restricted boltzmann machines“. In: *Proceedings of the 27th International Conference on Machine Learning (ICML)*. 2010, S. 807–814.

[PEM+20] Roy Shilkrot Peer, David Millán Escriva, Vinícius G. Mendonça u. a. *Mastering OpenCV 4*. 3rd. Birmingham / Mumbai: Packt Publishing, 2020.

[PMB13] Razvan Pascanu, Tomas Mikolov und Yoshua Bengio. „On the difficulty of training recurrent neural networks“. In: *International conference on machine learning*. PMLR. 2013, S. 1310–1318.

[Ras+22] Gabrielle Ras u. a. „Explainable Deep Learning: A Field Guide for the Uninitiated“. In: *Journal of Artificial Intelligence Research* 73 (2022), S. 329–372.

[Ros58] Frank Rosenblatt. „The perceptron: a probabilistic model for information storage and organization in the brain“. In: *Psychological Review* 65.6 (1958), S. 386–408. DOI: 10.1037/h0042519.

[San+18] Shibani Santurkar u. a. „How does batch normalization help optimization?“ In: *Advances in neural information processing systems* 31 (2018), S. 2483–2493.

[SB18] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. 2. Aufl. MIT Press, 2018.

[Sha13] Santiago Shackleton. *Pinball: The Lure of the Silver Ball*. Gestalten, 2013.

[SHP23] Ayush Somani, Alexander Horsch und Dilip K. Prasad. *Interpretability in Deep Learning*. Springer, 2023.

[Stu22] Zen Studios. *Pinball FX Series*. https://www.pinballfx.com/. Commercial digital pinball game for PC and consoles. 2022.

[TH12] Tijmen Tieleman und Geoffrey Hinton. „Lecture 6.5 - RMSProp: Divide the gradient by a running average of its recent magnitude“. In: (2012).

[Wes24] Alexander Wessels. *Entwicklung und prototypischer Aufbau eines durch Reinforcement Learning gesteuerten Flipper-Automaten*. Bachelorarbeit. Hamburg, 2024.

[Zei12] Matthew D. Zeiler. „ADADELTA: An Adaptive Learning Rate Method“. In: *arXiv preprint arXiv:1212.5701* (2012).

Erklärung zur selbständigen Bearbeitung

Hiermit versichere ich, Pascal Gutthardt, dass ich die vorliegende Masterarbeit mit dem Thema:

Deep Reinforcement Learning zur autonomen Steuerung eines Flipperautomaten mit kamerabasierter Umgebungserfassung

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und 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