Bachelorarbeit

Untersuchungen zum Ende-zu-Ende-Lernen im simulierten Roboterfußball mit tiefem optimierendem Lernen

Fachbereich 2: Informatik und Ingenieurwissenschaften

vorgelegt von:	Philipp Klöppner
Matrikelnummer:	1183638
Abgabedatum:	25.10.2019

Erstgutachter:	Prof. Dr. Thomas Gabel
Zweitgutachterin:	Prof. Dr. Ute Bauer-Wersing

Inhaltsverzeichnis

Abbildungsverzeichnis

Verzeichnis der Quellcode-Ausschnitte

1 Einleitung

Bei der Roboterfußball-Weltmeisterschaft RoboCup treten jährlich über dreihundert Teams in mehr als einem Dutzend verschiedener Ligen, sowohl mit echten Robotern als auch mit simulierten, an [1]. Die älteste der Simulationsligen ist die RoboCup 2D Soccer Simulation League, also die 2D-Fußballsimulationsliga. Abbildung 1.1 zeigt eine Visualisierung dieser Simulation.

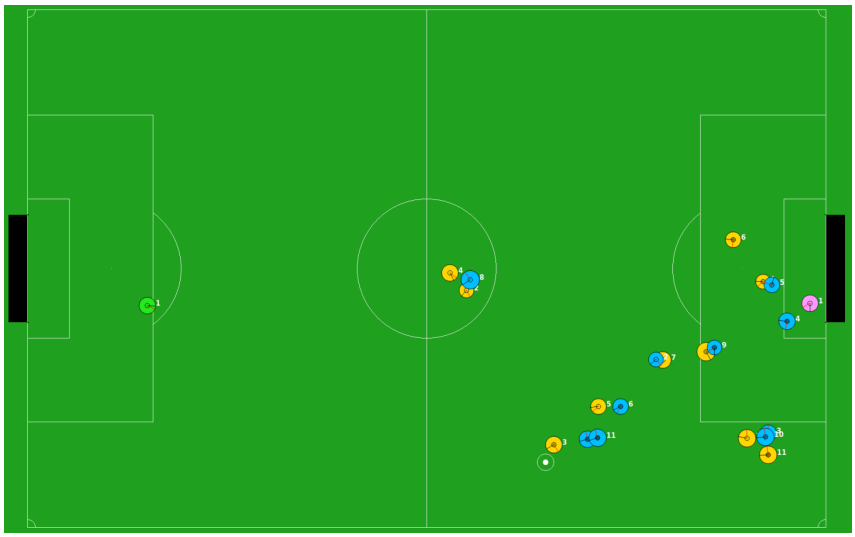

Abbildung 1.1: Bildschirmfoto einer Visualisierung des RoboCup 2D-Fußballspiels. Da Simulation und Visualisierung getrennt sind, kann die Visualisierung auch anders als hier abgebildet präsentiert werden. Bildquelle: Eigenes Werk; zeigt den RCSS-Monitor (`https://github.com/rcsoccersim/rcssmonitor`).

Es treten im Wettkampf zwei Teams mit je elf Spielfiguren an. Jede Spielfigur wird durch eine Instanz eines Steuerungsprogramms, durch einen *Agenten*, gesteuert. Ziel ist es, dass die Agenten intelligent miteinander spielen und dadurch Tore erzielen sowie Gegentore vereiteln. Diese Problemstellung macht die RoboCup-Fußballsimulation von Beginn an zu einer spannenden Forschungsumgebung für künstliche Intelligenz und maschinelles Lernen.

Im optimierenden Lernen, einer Methode des maschinellen Lernens, spricht man von Ende-zu-Ende-Lernen, wenn ein Algorithmus die rohen Daten der Umgebung erhält und anhand dieser eine optimale Aktion bestimmt. Dies grenzt sich zu anderen Ansätzen dadurch ab, dass nicht erst ein Modell der Umgebung erstellt wird. Ziel der Arbeit soll es sein,

zu zeigen, dass Ende-zu-Ende-Lernen in der RoboCup-2D-Fußballsimulation prinzipiell möglich ist.

Im nachfolgenden Kapitel werden die eingesetzte Software, der theoretische Hintergrund der eingesetzten Lernverfahren und einige verwandte Arbeiten präsentiert. Kapitel 3 stellt die im Rahmen dieser Arbeit untersuchten Problemstellungen im simulierten Roboterfußball vor. Um diese Probleme mit optimierendem Lernen lösen zu können, musste ein selbstlernender Agent entwickelt werden. Dieser Prozess wird in Kapitel 4 ausführlich behandelt. Der vorletzte Teil dieser Arbeit stellt die Ergebnisse vor und diskutiert diese. Im letzten Kapitel wird ein Fazit über die Arbeit gezogen und es werden wissenschaftliche Anschlussmöglichkeiten angesprochen.

2 Stand der Technik

Diese Arbeit wird im Kontext der RoboCup 2D-Fußballsimulation durchgeführt und nutzt bewährte Lernverfahren des optimierenden Lernens. In diesem Kapitel werden zunächst der Simulationsserver und bereits vorhandene Spielerprogramme (Agenten) vorgestellt. Der darauffolgende Abschnitt erklärt die theoretischen Grundlagen der Arbeit und die eingesetzten Lernverfahren. Die wichtigsten im Projekt eingesetzten Softwarebibliotheken werden ebenfalls dargelegt. Abschnitt 2.3 setzt diese Arbeit in den Kontext ähnlicher Publikationen.

2.1 Software

Bei der Durchführung des Projekts wurden verschiedene etablierte Softwarelösungen eingesetzt oder als Referenz zu Rate gezogen. Diese werden im Folgenden vorgestellt. Zunächst werden die Simulationsumgebung und zwei Agenten präsentiert. Im Anschluss daran werden Python und die in der Arbeit genutzten Bibliotheken eingeführt.

2.1.1 RoboCup Soccer Simulation Server

Der RoboCup Soccer Simulation Server ist eine quelloffene Software, die seit 1995 entwickelt wird[1]. Das Programm bietet die Simulationsplattform, mit der Spiele der 2D-Simulationsliga des RoboCup ausgetragen werden. Zu Beginn einer Partie verbinden sich Spielerprogramme, sogenannte Agenten, mit dem Server. Sie erhalten anschließend regelmäßig Informationen über den Zustand der Simulation und senden Steuersignale an den Server, um den Spieler mit der Welt interagieren zu lassen [2; 3].

In Abbildung 2.1 ist die Kommunikation zwischen Agenten und Server visualisiert.

Der Agent sendet zunächst an einen UDP-Socket, der per Konfigurationsdatei spezifiziert wird (Standardeinstellung: Port 6000), die Initialisierungsnachricht. Der Server öffnet dann einen weiteren UDP-Socket, über den anschließend die restliche Kommunikation abgewickelt wird.

[1] Die Ursprungsversion des *soccerserver* wurde 1993 entwickelt. Die heute genutzte Server-Software entstand 1995 daraus.

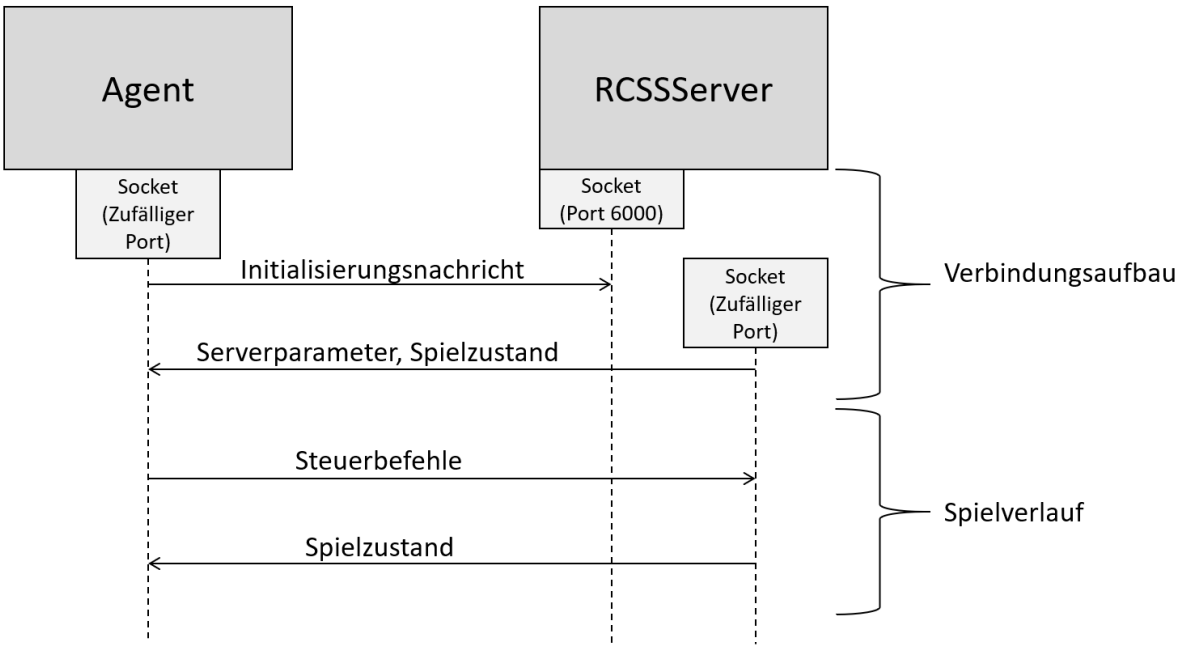

Abbildung 2.1: Verbindungsaufbau zwischen Agenten und Simulationsserver. Bildquelle: Eigenes Werk; mit Informationen aus [4] erstellt.

Die Kommunikation mit dem Server kann synchron oder asynchron erfolgen. Im asynchronen Modus wartet das Programm eine feste Zeitspanne zwischen jedem Simulationsschritt. Agenten können zwischen zwei Zeitschritten jederzeit ihre berechneten Aktionen an den Server senden, die im nächsten Simulationsschritt berücksichtigt werden. Im synchronen Modus wird der nächste Simulationsschritt sofort ausgeführt, nachdem Aktionen von allen Agenten eingegangen sind. Dadurch kann Zeit gespart werden, da die Länge einer Partie nur noch abhängig von der Rechenleistung ist.

Den Zustand der Simulation erhalten Agenten entweder in Form von simulierten Sensormesswerten oder, falls per Server-Konfiguration der *fullstate*-Modus eingeschaltet ist, als genaue Positionen und Attribute aller Feldspieler.

Neben den Spielern können sich auch Trainer mit dem Simulationsserver verbinden. Hier sind zwei Arten zu unterscheiden. Einfache Trainer erhalten Informationen über den Zustand des Spiels, können mit Spielern kommunizieren und Spielerwechsel durchführen. Ein Coach kann hingegen die Simulation direkt beeinflussen, indem er Spieler und Ball über das Feld teleportiert oder den Spielmodus ändert [4].

Bestehende Agenten

FRA-UNIted ist das RoboCup-Framework der Frankfurt University of Applied Sciences und ist eine Weiterentwicklung der C++-Codebasis der Brainstormers 2D [5]. Der Agent hat sich auf einigen Wettkämpfen bewährt und bereits Weltmeistertitel errungen. Dazu wird in einige Modulen beziehungsweise einzelnen Verhaltensweisen maschinelles Lernen verwendet [6]. Als Hilfsbibliothek wird N++ für künstliche neuronale Netzwerke eingebunden [7].

Agent2D ist ein quelloffener RoboCup-2D-Agent, der vom japanischen Team HELIOS freigegebenen wurde [8]. Er ist ebenso wie FRA-UNIted in C++ implementiert und zeichnet sich durch seine Simplizität aus. Dadurch kann problemlos auf ihm aufgebaut werden, um einen neuen Agenten zu implementieren. Alternativ lässt sich der offene Quellcode als gutes Nachschlagewerk für die Syntax des Protokolls des *rcssserver* verwenden.

2.1.2 Python

Python ist eine interpretierte Programmiersprache, welche dynamisch typisiert ist und sowohl objektorientierte als auch funktionale Programmierung ermöglicht [9]. Aufgrund der umfangreichen Standardbibliothek und zahlreicher weiterer Bibliotheken bietet es eine gute Grundlage für Softwareprojekte.

Bei dem Erstellen dieser Arbeit wurde zum Ausführen von Python-Code der CPython-Interpreter in Version 3.6 genutzt.

Python-Standardbibliothek

Verschiedene Teile der vorgestellten Agentenimplementierung nutzen die sehr umfangreiche Standardbibliothek von Python. Insbesondere das Modul `asyncio` war hilfreich, um die Interprozesskommunikation zwischen Agenten und Server asynchron zu ermöglichen [10].

TensorFlow

TensorFlow ist eine quelloffene Softwarebibliothek, die von Google gewartet und weiterentwickelt wird [11]. Sie bietet unter anderem einfache Schnittstellen zum Erstellen und

Trainieren künstlicher neuronaler Netzwerke. Die Berechnungen werden zum größten Teil durch eine hochoptimierte C++-Implementierung durchgeführt. Die Schnittstelle für den Benutzer ist jedoch primär in Python verfügbar.

Die Definition des Modells als Rechengraph und dessen Gewichte können serialisiert werden. So lassen sich die neuronalen Netzwerkmodelle auch in eingeschränkter Weise mit anderen Programmiersprachen verwenden, wie zum Beispiel in der Arbeit von Gabel et al. [6] gezeigt.

TensorFlow bietet zudem gute Unterstützung für die Softwarebibliothek CUDA von Nvidia [12], mit der die aufwendigen Rechenoperationen auf Grafikprozessoren ausgeführt werden können. So kann im Vergleich zu normalen Hauptprozessoren (CPUs) der Lernprozess um einen Faktor zwischen 10 und 100 schneller durchgeführt werden [13].

Numeric Python

Numeric Python, oft NumPy abgekürzt, ist eine Softwarebibliothek für Python, die Schnittstellen für effiziente numerische Berechnungen bereitstellt [14]. In der Implementierung dieser Arbeit wurden insbesondere die mehrdimensionalen Felder der Bibliothek genutzt, welche Kompatibilität mit TensorFlow bieten.

Matplotlib

Matplotlib [15] ist eine quelloffene Softwarebibliothek für Datenvisualisierung in Python. Einige der Abbildungen in dieser Arbeit wurden mit ihr erstellt.

OpenAI Gym

Gym ist ein Produkt der Non-Profit-Organisation OpenAI [16]. Es bietet unter anderem einige Umgebungen für das optimierende Lernen, mit denen Lernverfahren getestet werden können.

Die in Kapitel 4 vorgestellte Umgebung für optimierendes Lernen in der Domäne des simulierten Roboterfußballs orientiert sich an den Schnittstellen von OpenAI, sodass

Algorithmen, die für Gym-Probleme entworfen sind auch im simulierten Roboterfußball evaluiert werden können.

Google abseil

`abseil` von Google ist eine Softwarebibliothek für Python und C++, die in diesem Projekt primär für die Unterstützung von Kommandozeilenparametern verwendet wird [17].

2.2 Theoretische Grundlagen

In diesem Kapitel werden die theoretischen Hintergründe der Arbeit beleuchtet. Markov-Entscheidungsprozesse (MDP für engl. markov decision process) werden kurz vorgestellt und es wird gezeigt, wie die Roboterfußballsimulation als ein MDP beschrieben werden kann. Das optimierende Lernen und darin speziell das Q-Lernen werden als Lösungsverfahren für Markov-Entscheidungsprozesse präsentiert.

2.2.1 Markov-Entscheidungsprozess

Markov-Entscheidungsprozesse sind mathematische Modelle für Entscheidungsprobleme, in denen die Markov-Eigenschaft gilt. Das bedeutet, dass die Wahrscheinlichkeit, im Zustand s durch die Aktion a in den Zustand s' zu gelangen immer nur abhängig von s und a, nie jedoch von den vergangenen Zuständen ist. Jeder Zustandsübergang wird mit einem skalaren Belohnungswert bewertet, wobei das Ziel für den Agenten ist, die maximale Summe an Belohnungen über unendlich viele Zeitschritte zu erreichen. Aus diesen Rahmenbedingungen resultiert, dass es für jeden Zustand s eine optimale Aktion a^* gibt, die den Belohnungswert in Zukunft maximiert. Eine Strategie zu erlernen, die diese optimale Aktion für jeden Zustand findet, ist Ziel des optimierenden Lernens [18].

Simulierter Roboterfußball als stochastischer MDP

Die Problemstellung in der RoboCup 2D-Fußballsimulation lässt sich als stochastischer Markov-Entscheidungsprozess definieren. Die Menge von Zuständen S setzt sich aus den für den Agenten wahrnehmbaren Eigenschaften des Spiels zusammen. Diese sind abhängig davon, ob der *fullstate*-Modus eingeschaltet ist und können unabhängig davon unter

anderem den aktuellen Spielstand, Zeitschritt und Spielmodus umfassen. Im *fullstate*-Modus würden noch Spielerattribute wie Position, Beschleunigungsvektor und Ausdauer jedes Spielers hinzukommen. Im normalen Spielmodus hingegen erhält der Agent die simulierten Sensormesswerte.

Aktionen werden durch das Aktionsmodell des Fußballsimulationsservers definiert und umfassen zum Beispiel Laufen, Schießen und Drehen.

Die Wahrscheinlichkeiten der Zustandsübergänge sind dem Agenten nicht bekannt. Zudem sind Aktionen durch das Rauschmodell der Fußballsimulation nichtdeterministisch. Eine Aktion „Drehung um 90 Grad“ hat in der Praxis beispielsweise den tatsächlichen Effekt „Drehung um ungefähr 90 Grad“. Um wie viel Grad sich der Agent tatsächlich dreht, kann erst im nächsten Zeitschritt ermittelt werden [4].

2.2.2 Künstliche Neuronale Netzwerke

Künstliche neuronale Netzwerke (KNN) sind von biologischen Nervensystemen inspiriert und Forschungsgegenstand der Neuroinformatik. Im maschinellen Lernen werden insbesondere mehrlagige Perzeptren (MLP für engl. multilayer perceptron) verwendet, um Funktionen anzunähern.

Dies ist zum Beispiel in der Bildklassifizierung nützlich, wo ein Bild x durch die Funktion $y = f^*(x)$ der Kategorie y zugeordnet wird. Durch einen Lernvorgang werden Gewichte für das Netzwerk gefunden, durch welche dieses die optimale Klassifizierungsfunktion f^* annähert [19, vgl. S. 164].

Die Funktionsweise neuronaler Netzwerke zu erläutern würde an dieser Stelle zu weit führen, weswegen auf Goodfellow et al. [19] sowie LeCun et al. [20] verwiesen wird. In dieser Arbeit wurden künstliche neuronale Netzwerke für das optimierende Lernen eingesetzt, welches im nächsten Abschnitt vorgestellt wird.

2.2.3 Optimierendes Lernen

Das optimierende Lernen ist eine Methode des maschinellen Lernens, die es ermöglicht, eine optimale Regelungsstrategie für einen MDP zu finden, bei dem die Wahrscheinlichkeiten der Zustandsübergänge sowie die Belohnungsfunktion für den Agenten unbekannt sind [18].

Der lernende Agent interagiert mit der Ausführungsumgebung, die in der Regel als MDP beschrieben ist. Das erhaltene Belohnungssignal verwendet er anschließend zur Optimierung seiner Strategie. Dabei bildet das Quadrupel (s, a, r, s'), bestehend aus Ausgangszustand s, Aktion a, Belohnungswert r und Folgezustand s' einen Zustandsübergang, der durch das Lernverfahren verwertet wird. Häufig wird das Problem in Episoden unterteilt, wobei jede Episode so lange ausgeführt wird, bis eine Abbruchbedingung erfüllt ist. Das könnte eine maximale Anzahl an Zeitschritten (Zustandsübergängen) oder auch eine spezifische Bedingung der Umgebung, zum Beispiel das Verlassen des Spielfeldes durch den Spieler sein [18].

Es gibt eine Vielzahl verschiedener Lernverfahren im optimierenden Lernen. Ziel dieser Arbeit ist es, die Durchführbarkeit von Ende-zu-Ende-Lernen im simulierten Roboterfußball zu untersuchen. Das bedeutet, es wird kein Modell erzeugt, welches das Problem vereinfachen würde, sondern es werden lediglich die gegebenen Informationen zu Rate gezogen, um eine Entscheidung zu treffen. Für diese Art des optimierenden Lernens hat sich die Methode Temporal Difference Learning (TDL, [21]) bewährt, bei dem zeitlich versetzte Belohnungswerte bei der Wahl der Aktion berücksichtigt werden. Dieses Verfahren wurde zuerst 1992 für das Erlernen einer Strategie für das Brettspiel Backgammon eingesetzt. Seit einigen Jahren werden auch neuronale Netzwerke für diese Verfahren eingesetzt [22] und 2013 führten Minh et al. den DQN-Algorithmus (engl. deep Q-networks, tiefe Q-Netzwerke, [23]) ein, welcher Q-Lernen mit tiefen neuronalen Netzwerken verbindet.

Q-Lernen

In Markov-Entscheidungsprozessen bestimmt die optimale Wertfunktion $q^*(s, a)$ den Wert der Aktion a in Zustand s unter der optimalen Regelungsstrategie π^*. Der Wert ist dabei nicht nur durch die direkte Belohnung, sondern auch die zu erwartenden folgenden Belohnungen bestimmt. Würde man also nur die optimale Wertfunktion kennen, ließe sich wiederum die optimale Regelungsstrategie $\pi^*(s) = \max_{a \in A} q^*(s, a)$ aus ihr ableiten [24].

In der Praxis ist die optimale Wertfunktion q^* unbekannt. Beim Q-Lernen wird mithilfe dynamischer Programmierung die optimale Wertfunktion iterativ angenähert, um eine optimale Regelungsstrategie aus ihr abzuleiten.

Im klassischen Q-Lernen wird der Wert einer Aktion durch die Gleichung

$$Q_{n+1}(s_t, a_t) \leftarrow (1 - \alpha) \cdot Q_n(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q_n(s_{t+1}, a))$$

angepasst. Die Lernrate $0 \leq \alpha \leq 1$ bestimmt dabei das Ausmaß der Veränderung durch neue Informationen. Der Diskontierungsfaktor γ bestimmt die Weitsichtigkeit des Agenten. Wird $\gamma = 0$ gewählt, fließt nur der direkte Belohnungswert in die Q-Funktion ein, welche dadurch nicht mehr $q^*(s, a)$, sondern die Belohnungsfunktion $r(s, a)$ approximiert. Bei $\gamma \geq 1$ können die Werte von Q divergieren, wodurch es unmöglich wird, eine optimale Regelungsstrategie zu finden [18].

Die Q-Funktion konvergiert durch die Iteration dieses Verfahrens ohne Kenntnis der Zustandsübergangswahrscheinlichkeiten zu q^* [18], wodurch das Q-Lernen gut geeignet für die stochastische Roboterfußballsimulation ist.

Tiefe Q-Netzwerke

Das Lernverfahren DQN (engl. deep Q-networks, tiefe Q-Netzwerke) wurde 2013 durch Mnih et al. [23] eingeführt und ist eine Erweiterung des NFQ-Verfahrens [22] (engl. neural-fitted Q, Q-Lernen mithilfe neuronaler Netzwerke). Dabei wird zur Annäherung von q^* keine assoziative Datenstruktur, sondern ein tiefes neuronales Netzwerk verwendet. Dadurch wird es dem Agenten ermöglicht, mit kontinuierlichen Zustandswerten zu arbeiten. Das wäre durch die Eigenschaft der Überabzählbarkeit der reellen Zahlen mit tabellarischen Methoden nicht ohne Weiteres möglich.

Die Lernmethode profitiert außerdem davon, dass hier das Experience-Replay-Verfahren [25] und Batch-Lernverfahren mit dem Q-Lernen kombiniert werden, sodass der Lernvorgang dateneffizient durchgeführt werden kann [22; 23].

Epsilon-gierige Regelungsmethode

Bei einer ϵ-gierigen Regelungsstrategie (engl. ϵ-greedy policy) gibt es einen Wahrscheinlichkeitswert ϵ, der entweder konstant ist oder über die Zeit sinkt. Bei der Regelung mit dieser Methode wird in einem Zeitschritt t mit Zustand s_t mit einer Wahrscheinlichkeit von ϵ die

Aktion a_t zufällig gewählt. Mit Wahrscheinlichkeit $1-\epsilon$ wird die von der Regelungsstrategie π_t bestimmte Aktion $a_t = \pi_t(s_t)$ gewählt.

Beim Ausführen einer zufälligen Aktion wird im optimierenden Lernen von Exploration gesprochen. Das Ausführen der durch die Strategie vorausgesagten Aktion wird als Ausnutzung (engl. exploitation) bezeichnet [18].

2.3 Verwandte Arbeiten

In mehr als 20 Jahren Geschichte des RoboCup haben sich selbstverständlich viele Forschungsgruppen mit der Anwendung maschinellen Lernens für den simulierten Roboterfußball beschäftigt. Einige dieser sollen im Folgenden vorgestellt und in den Kontext der vorliegenden Arbeit gesetzt werden.

In einem Artikel von 2007 [26] stellten Gabel und Riedmiller verschiedene Problemstellungen in der Fußballsimulation vor, die mit optimierendem Lernen gelöst wurden. Die Strategien sind in ein Agenten-Framework eingebettet, sodass nur einzelne Verhaltensweisen, nicht aber das gesamte Agentenverhalten, durch eine erlernte Regelungsstrategie gesteuert wird.

Die gleiche Arbeit behandelt auch, dass eine Variante von modellfreiem Q-Lernen bis zu einer Spielerzahl von zwei gegen zwei erfolgreich angewendet werden konnte. Darüber hinaus würde der Ansatz jedoch nicht skalieren. Nachdem Mnih et al. 2013 [23] zeigten, dass sich komplexe Problemstellungen wie das Meistern von Atari-Videospielen durch modellfreies Q-Lernen mit tiefen neuronalen Netzwerken lösen lassen, ist es naheliegend, das Problem des modellfreien Ende-zu-Ende-Lernens im simulierten Roboterfußball mit diesem weiterentwickelten Verfahren erneut zu untersuchen.

Matthew Hausknecht und Peter Stone haben in einer 2015 veröffentlichen Arbeit [27] den Lernalgorithmus DDPG (Lillicrap et al. [28]) für Half-Field Offense (HFO, [29]), einen Spielmodus des RoboCup, evaluiert. Sie konnten zeigen, dass mithilfe dieses Algorithmus eine Regelungsstrategie erlernt werden konnte, die im HFO-Modus ein ehemaliges Weltmeister-Team schlagen konnte. Anstelle der rohen Sensorinformationen wurde hier das Weltmodell des zuvor erwähnten Agent 2D [8] verwendet. Die für diese Arbeit verwendete Lernumgebung wurde 2016 freigegeben und wird in der Arbeit [30] beschrieben.

Dass prinzipiell Ende-zu-Ende-Lernen von Problemen mit mehreren Agenten mit tiefem optimierendem Lernen möglich ist, zeigten Jaderberg et al. am Beispiel des Computerspiels Quake III Arena [31]. Ob dieser Ansatz auch für den simulierten Roboterfußball funktioniert, wurde bisher nicht untersucht.

Die vorliegende Arbeit kombiniert die Idee des modellfreien Q-Lernens im RoboCup-Umfeld von Gabel und Riedmiller mit dem Lösungsansatz des tiefen optimierenden Lernens von Hausknecht und Stone.

3 Problemmodellierung

In diesem Kapitel werden einige Problemstellungen vorgestellt und wie in diesen die Zustände, Aktionen und Belohnungen modelliert sind. Dabei wird zunächst die Problemstellung des omnidirektionalen Laufs auf dem Spielfeld vorgestellt. Hierbei werden zwei Varianten unterschieden. Eine Diskretisierung des Zustandsraumes wird vorgenommen, um Lernen mit dem klassischen Q-Lernen nach Watkins [24] zu ermöglichen. Die zweite Variante verzichtet auf diese Diskretisierung, hier werden nur die „rohen“ Zustände der Simulation an den Agenten gegeben.

Als Nächstes wird ein anspruchsvollerer Lauf, bei dem nur Vorwärtsbewegungen und Drehungen erlaubt sind, vorgestellt. Zuletzt wird eine richtige Problemstellung des Fußballspiels präsentiert, bei der das Ziel ist, den Ball in das gegnerische Tor zu schießen.

3.1 Problemstellung 1: Omnidirektionaler Lauf in diskreter Umgebung

Ziel von Experimenten mit dieser Problemstellung soll sein, die Lernumgebung erstmals für maschinelles Lernen zu testen. Dazu bietet sich das Q-Lernen an, da es einfach zu implementieren ist und weniger potenzielle Fehlerquellen bietet. Da Q-Lernen von diskreten Zustandsräumen ausgeht, wurden die kontinuierlichen Zustände der Fußballsimulation diskretisiert.

Der Agent hat in diesem Experiment vier Aktionen zur Verfügung und erhält als Beobachtung seine Position und den Beschleunigungsvektor in diskretisierter Form. Belohnt wird er für die Annäherung an den Zielpunkt im Spielfeld.

Aktionsraum

Die in diesem Experiment möglichen vier Aktionen sind $A = \{$Vorwärtslauf, Rückwärtslauf, Seitwärtslauf links, Seitwärtslauf rechts$\}$[1].

[1] Die Aktionen entsprechen den *rcssserver*-Kommandos „(dash 100 0)“, „(dash 100 180)“, „(dash 100 -90)“ und „(dash 100 90)“.

Zustandsraum

Für die Zustände in diesem Experiment werden die Position und der Beschleunigungsvektor des Spielers, welche durch die Simulation als Fließkommazahlen bereitgestellt werden, diskretisiert.

Für die Spielerposition wurde die Funktion

$$f : \mathbb{R}^2 \to \{-104, -103, ..., 104\} \times \{-66, -65, ..., 66\}$$

mit

$$f(p) = (max(-104, min(104, \lfloor p_x * 2 \rfloor)), max(-66, min(66, \lfloor p_y * 2 \rfloor)))$$

definiert. Die Koordinaten werden zwischen den Spielfeldrändern in Schritten von 0,5 abgebildet. Prinzipiell hätte eine einfache Abbildung durch Abrundung ausgereicht, jedoch wäre in diesem Fall die Gefahr größer, dass sich der Spieler nach einer Aktion noch im selben Zustand befindet, z.B. im Fall $s_t^{roh} = (1{,}01, 1{,}5), a =$ Vorwärtslauf, $s_{t+1}^{roh} = (1{,}98, 1{,}5)$.

Der Beschleunigungsvektor wird im Zustand sehr stark vereinfacht dargestellt und dient nur dem Zweck, die Beschleunigungsrichtung anzugeben. Dazu wurde folgende Funktion definiert:

$$g : \mathbb{R}^2 \to \{0, 1, 2, 3\}$$

mit

$$g(\vec{v}) = \left\lfloor \frac{atan2(\vec{v}) + \frac{\pi}{4}}{\frac{\pi}{2}} + 1 \right\rfloor,$$

sodass $g(\vec{v}) = 0$, wenn $\vec{v}$ in die positive x-Richtung zeigt, $g(\vec{v}) = 1$, wenn $\vec{v}$ in die positive y-Richtung zeigt, und so weiter. Diese Vereinfachung erlaubt das Ausnutzen des Schwungs, ohne den Zustandsraum zu stark zu vergrößern.

Der Zustandsraum hat somit eine Kardinalität von $209 \times 133 \times 4 \approx 1{,}1 * 10^5$

Abbruchkriterien

Die Episoden terminieren in diesem Experiment, wenn der Spieler den Spielfeldrand überschreitet oder wenn die maximale Anzahl an Zeitschritten erreicht ist.

Belohnungsfunktion

Die Belohnungsfunktion belohnt das Annähern und bestraft das Entfernen von der Zielposition. Darüber hinaus wird ein konstanter Wert zur Belohnung addiert, wenn das Ziel erreicht wurde.

Sei P^z die Zielposition, P^p_s die Position des Spielers im Zustand s und $d(p,q) = \|p - q\|$ die euklidische Distanz von p und q. Dann ist die Belohnungsfunktion:

$$r : S \times A \times S \to \mathbb{R}$$

mit

$$r(s,a,s') = d(P^z, P^p_s) - d(P^z, P^p_{s'}) + c \text{ wobei } c = \begin{cases} 10, & \text{wenn } d(P^z, P^p_{s'}) \leq 5 \\ 1, & \text{wenn } 5 < d(P^z, P^p_{s'}) < 25 \\ 0, & \text{anderenfalls.} \end{cases}$$

3.2 Problemstellung 2: Omnidirektionaler Lauf in kontinuierlicher Umgebung

Diese Problemstellung ist der in Abschnitt 3.1 sehr ähnlich, mit dem entscheidenden Unterschied, dass die Zustände nicht diskretisiert werden und somit der Zustandsraum $S \subseteq \mathbb{R}^4$, bestehend aus X- und Y-Koordinate der Spielerposition sowie X- und Y-Beschleunigung des Spielers, ist.

Dadurch disqualifiziert sich das klassische Q-Lernen als Lernverfahren, da es nur mit diskreten Zustandsräumen praktikabel ist. Es ist vorgesehen, dass Versuche mit dieser Problemstellung durch DQN oder ähnliche Verfahren für kontinuierliche Zustandsräume gelöst wird.

Die Belohnungsfunktion und Abbruchkriterien wurden wie bei der ersten Problemstellung gewählt.

3.3 Problemstellung 3: Vorwärtslauf und Drehung in kontinuierlicher Umgebung

Diese Problemstellung ist eine schwierigere Variante der Zweiten, da der Agent hier nicht omnidirektional laufen kann, sondern gezwungen ist, sich einem Laufbefehl in die gewünschte Richtung zu drehen. Belohnungsfunktion und Abbruchkriterien wurden wie bei den ersten beiden Problemstellungen gewählt.

Zustandsraum

Der Zustandsraum erweitert den Zustandsraum von Problemstellung 3.2 um den Drehwinkel des Spielers, sodass $S \subseteq \mathbb{R}^5$ ist.

Aktionsraum

Verfügbare Aktionen in dieser Problemstellung sind $A = \{$Vorwärtslauf, Linksdrehung, Rechtsdrehung$\}$[2].

3.4 Problemstellung 4: Torschuss in kontinuierlicher Umgebung

In diesem Abschnitt wird eine Problemstellung diskutiert, die tatsächliches Fußballspiel beinhaltet. Der Agent soll den Ball ins Tor schießen. Der Zustandsraum wird signifikant erweitert. Als weitere Aktion wird der Schuss eingeführt und der Agent wird durch die Belohnungsfunktion motiviert, zum Ball zu laufen und ihn ins Tor zu befördern.

Zustandsraum

Der Zustandsraum hat in dieser Problemstellung die Dimensionen aus Abschnitt 3.3 mit Erweiterung des Balls (Position und Beschleunigungsvektor) sowie dem Punktestand des eigenen Teams (Ganzzahl). Da der Punktestand ebenfalls als reelle Zahl gehandelt wird, ist der Zustandsraum $S \subseteq \mathbb{R}^{10}$.

[2]Die Aktionen entsprechen den *rcssserver*-Kommandos: „(dash 100)“, „(turn -90)“ und „(turn 90)“.

Aktionsraum

Der Agent kann hier die Aktionen $A = \{$Vorwärtslauf, Linksdrehung, Rechtsdrehung, Schuss$\}$[3] ausführen.

Abbruchkriterien

Eine Episode terminiert hier nach Zeitüberschreitung, wenn der Spieler das Spielfeld verlässt oder sobald ein Tor gefallen ist.

Belohnungsfunktion

Die Belohnungsfunktion ist in drei Einzelfunktionen unterteilt. Funktion r_1 belohnt gefallene Tore, r_2 die Annäherung des Spielers an den Ball und r_3 die Annäherung des Balls an das Tor. Zusammen mit den Gewichtungsfaktoren w_1, w_2 und w_3 ergeben diese die Belohnungsfunktion $r(s, a, s') = \sum_{i=1}^{3} w_i r_i(s, a, s')$.

Diese Funktionen sind wie folgt definiert:

$$r_1(s, a, s') = \begin{cases} 1, & \text{wenn der Punktestand in } s' > 0 \text{ ist} \\ 0, & \text{anderenfalls.} \end{cases}$$

Sei P_s^p die Spielerposition in Zustand s, P_s^b die Ballposition im Zustand s, P_s^g der dem Ball nächstgelegene Punkt des Tors und $d(p, q) = \|p - q\|$ die euklidische Distanz von p und q. Dann ist

$$r_2(s, a, s') = d(P_s^p, P_s^b) - d(P_{s'}^p, P_s^b)$$

und

$$r_3(s, a, s') = d(P_s^b, P_s^g) - d(P_{s'}^b, P_{s'}^g).$$

Für r_2 wird die Distanzänderung zur vorherigen Ballposition berücksichtigt, um das Treten des Balls nicht zu bestrafen.

[3] Die Aktionen entsprechen den *rcssserver*-Kommandos: „(dash 100)“, „(turn -90)“, „(turn 90)“ und „(kick 50 0)“.

4 Implementierung

In diesem Kapitel werden die softwaretechnischen Fragestellungen und Herausforderungen der Arbeit diskutiert. Zunächst wird erörtert, warum eine Eigenentwicklung an Stelle eines bereits vorhandenen Agenten sinnvoll ist. Anschließend werden die Planung und Implementierung der entstandenen Software detailliert vorgestellt.

4.1 Anforderungen

Die Anforderungen an das Lern-Framework sind neben der ausreichenden Unterstützung des RoboCup-Server-Protokolls auch die Unterstützung für künstliche neuronale Netzwerke sowie eine einfache Schnittstelle für das Durchführen von Episoden.

Sind diese grundlegenden Anforderungen erfüllt, müssen auch Konfigurationsmöglichkeiten für Lernumgebung und -verfahren vorhanden sein. Die Lernumgebung muss es dabei ermöglichen, die verschiedenen Beobachtungsschemata der vorgestellten Experimente bereitstellen zu können und diese bei Bedarf auch einfach erweitern zu können. Die Lernverfahren sollten ebenfalls gut konfigurierbar sein, da für erfolgreiche Experimente das finden passender Hyperparameter wichtig sein könnte.

4.2 Neuanfang statt Weiterentwicklung

In Abschnitt 2.1.1 wurden zwei Agenten vorgestellt, die weiterentwickelt werden könnten, um die geplanten Experimente durchführen zu können. Im Folgenden wird die Entscheidung, einen neuen Agenten für dieses Projekt zu implementieren, erklärt.

4.2.1 Weiterentwicklung eines bestehenden Agenten

Die Wahl der Weiterentwicklung eines vorhandenen Agenten ist gerade deshalb naheliegend, weil die grundlegenden Funktionalitäten bereits vorhanden sind. So wären Probleme wegen Fehlern in der Implementierung des Client-Server-Protokolls hier unwahrscheinlich, da die Agenten bereits seit Jahren aktiv genutzt und weiterentwickelt werden.

Ein entscheidender Nachteil ist jedoch die Kompatibilität mit modernen Bibliotheken für maschinelles Lernen. Für neuronale Netzwerke sollte TensorFlow verwendet werden, welches zwar eine C++-Schnittstelle anbietet, jedoch eigentlich primär Python bedient.

Wie Gabel et al. [6] erläutern, wird mindestens für das Erstellen des Rechengraphen Python benötigt. Deswegen wurde zunächst evaluiert, ob ein Hybridmodell hier Sinn ergeben könnte.

Dazu wurde die Referenzimplementierung aus der Arbeit von Gabel et al. [32] nachgebaut. Hier mussten allerdings einige Modifikationen gemacht werden. Da die Regelungsstrategie des lernenden Agenten nach jedem Zeitschritt optimiert werden sollte, musste der Graph die Optimierungsoperation auch in C++ nutzen können. Prinzipiell funktioniert diese Lösung, allerdings nur mit CPU-Unterstützung. Dies ist ein Nachteil, da Grafikprozessoren (GPUs für engl. graphics processing units) das Training künstlicher neuronaler Netzwerke deutlich beschleunigen [13].

Selbst mit GPU-Unterstützung wäre an dieser Stelle noch nicht geklärt, wie gut mit dieser hybriden Lösung automatisierte Lernexperimente durchgeführt werden könnten. Zudem muss noch untersucht werden, inwieweit der *fullstate*-Modus und der synchrone Spielmodus durch die Agenten unterstützt werden.

Diese Punkte machen die Weiterentwicklung vorhandener Agenten zunächst unattraktiv, da der Implementierungsaufwand schlecht abzuschätzen ist.

4.2.2 Entwicklung eines Python-Agenten

Die Herausforderungen bei der Implementierung eines neuen Agenten umfassen insbesondere die Gewährleistung einer robusten und fehlerfreien Kommunikation mit dem Simulationsserver. Ist diese nicht zuverlässig, können Experimente nicht korrekt durchgeführt werden.

Das Client-Server-Protokoll, welches auf einer Lisp-ähnlichen Syntax beruht, ist wenig komplex und gut in [4][1] dokumentiert. Für die Kommunikation über das Datenprotokoll UDP bietet Python eine Schnittstelle zu Systemaufrufen, sodass dies auch kein Problem darstellt.

TensorFlow einzubinden ist in Python trivial, da das notwendige Modul per Kommandozeilenbefehl installiert werden kann und mit einem Importbefehl eingebunden wird.

[1] Die Quelle ist zwar veraltet, in Kombination mit dem aktuellen Quellcode, der sich unter `https://github.com/rcsoccersim/rcssserver` finden lässt, ist sie jedoch als Referenz für das Grundprinzip brauchbar.

Die Implementierung des Lernverfahrens musste sowohl im Falle eines C++- als auch eines Python-Agenten selbst vorgenommen werden. Es wurde jedoch davon ausgegangen, dass dieser Prozess durch die Flexibilität der Sprache Python hier potenziell weniger Zeit in Anspruch nehmen könnte.

Ein weiterer wichtiger Aspekt soll die Performanz des Agenten sein. C++ hat hier als kompilierte Sprache einen Vorteil gegenüber dem interpretierten Python. Da der größte Rechenaufwand bei diesem Projekt jedoch im Trainieren der neuronalen Netzwerke liegt, und TensorFlow sowohl mit der C++- als auch der Python-Schnittstelle zur Berechnung hochoptimierten C++-Code verwendet, wird der sonstige potenzielle Geschwindigkeitsunterschied als zu vernachlässigen angesehen.

Wegen der guten Erfolgsaussichten auf eine erfolgreiche Implementierung mit überschaubarem Zeitaufwand fiel die Wahl letztlich darauf, einen Agenten in Python zu implementieren.

4.3 Das Lern-Framework in Python

Das Framework wurde so entwickelt, dass Module jedes Experiments leicht austauschbar waren. Dies sollte ermöglichen, dass beispielsweise für den Wechsel zwischen Problemstellung 1 und 2 lediglich ein Konfigurationsparameter geändert werden musste. Ein Überblick der modularen Architektur ist in Abbildung 4.1 dargestellt.

Das angewendete Lernverfahren, hier als Schnittstelle „Model“ dargestellt, kann ausgetauscht werden, um verschiedene Algorithmen im gleichen Kontext zu erproben.

Um einen Lernalgorithmus zu testen kann es zudem hilfreich sein, zunächst eine simplere Lerndomäne bzw. -umgebung einzubinden. Dazu wurde die hier implementierte Klasse „Environment“ so konzipiert, dass sie mit der Schnittstelle von OpenAI Gym [16], einer populären Sammlung von Problemstellungen des optimierenden Lernens, kompatibel ist.

In der Hauptfunktion des Lernprogramms werden ein *Environment*- und ein *Model*-Objekt für das Experiment erstellt. Diese werden an eine *Agent*-Instanz übergeben, welche für das Durchführen von Episoden verantwortlich ist.

Im Folgenden werden die erstellten Klassen vorgestellt und deren Implementierung erläutert. Das Python-Modul `learner` umfasst diese Klassen.

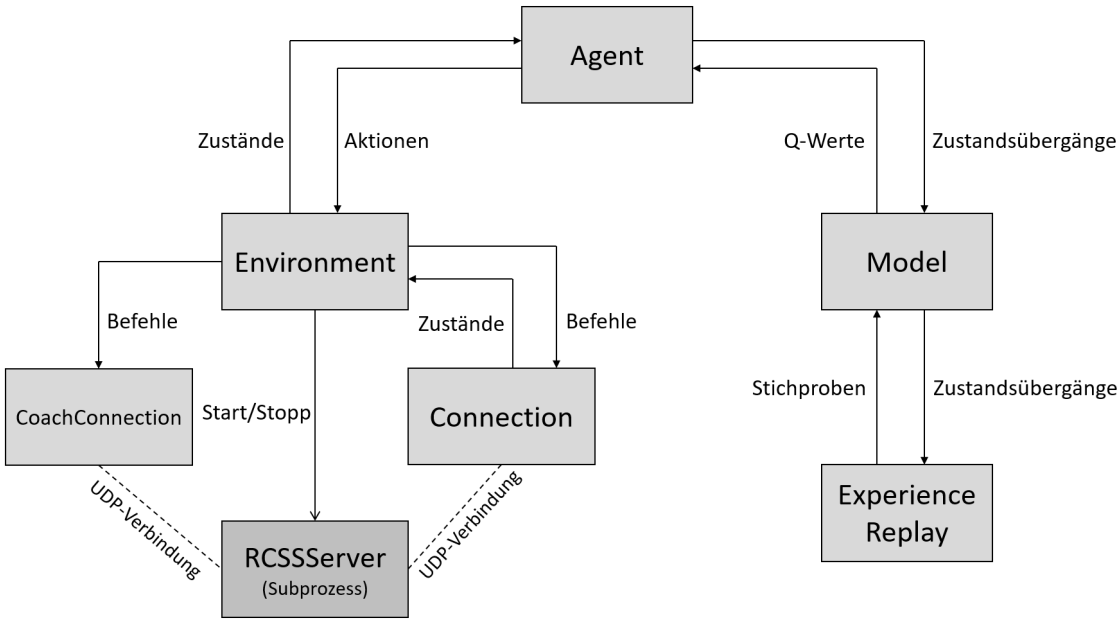

Abbildung 4.1: Überblick der Beziehungen zwischen den Klassen, die den Lernvorgang ermöglichen. Bildquelle: Eigenes Werk

4.3.1 Ausführungsumgebung

Die `Environment`-Klasse ist verantwortlich für die Bereitstellung der Simulationsumgebung und die Interaktion mit dieser durch den Agenten. Sie bietet eine `reset`-Methode, welche die Simulationsumgebung erzeugt und die initiale Beobachtung, den Startzustand des MDP, zurückgibt. Die `step`-Methode wird während der Episode wiederholt aufgerufen und nimmt die Aktion des Agenten entgegen. Diese Aktion wird an den Simulationsserver übertragen, welcher diese dann im nächsten Simulationsschritt berücksichtigt. Sie liefert ein Tripel (s', r, d) zurück, wobei s' der Folgezustand, r der skalare Belohnungswert und d ein boolescher Wert ist, der angibt, ob die Episode zu Ende ist.

Für die Verbindung mit dem Server wird eine Instanz der `Connection`-Klasse sowie eine Instanz von `CoachConnection` verwendet.

reset

Die Aufgabe der `reset`-Methode ist es, die Episode zu initialisieren. Dazu wird zunächst, falls vorhanden, der bestehende Server sowie Verbindungen zu diesem terminiert. Anschließend wird eine neue Instanz des Servers hochgefahren. Für die Durchführung der Episode wird pro Spieler eine Verbindung zum Server aufgebaut. Zusätzlich wird ein Coach, also ein Trainerprogramm mit dem Server verbunden. Der Coach sendet dann an den Server Kommandos um Spieler und Ball zu den designierten Startpositionen zu teleportieren. Je nach Experiment sind diese in jeder Episode gleich oder werden nach einem vorgegebenen Muster gewählt.

step

Die `step`-Methode ist das Herzstück des lernenden Agenten und bietet die Zustandsübergänge in dem zu lösenden Markov-Entscheidungsprozess. Sie nimmt als Eingabe einen diskreten Aktionswert, mithilfe dessen ein Steuerkommando für die Simulation erzeugt und an den Server weitergeleitet wird. Dazu wird die `step`-Methode der Klasse `Connection` aufgerufen, welche die Beobachtungen aus dem Folgezustand der Simulation zurückliefert, sobald sie verfügbar sind.

Aufgrund von Ausgangszustand, Aktion und Folgezustand wird die für das Experiment vorgesehene Belohnung errechnet. Aus dem Folgezustand geht außerdem hervor, ob sich die Simulation in einem terminalen Zustand befindet, zum Beispiel wegen Ablauf der Zeit für das Experiment, ungültigem Spielmodus oder anderen beobachteten Faktoren.

Das Tripel aus Folgezustand $s1$, Belohnungswert r und dem booleschen Indikator für Endzustände d wird anschließend zurückgegeben.

Belohnungen

Die Belohnungsfunktionen sind in eigene Klassen ausgelagert. Dabei wird ausgenutzt, dass Python es ermöglicht, Objekte einer Klasse direkt aufzurufen. Die `__call__`-Methode

der Belohnungsklassen wird so implementiert, dass sie das Tripel (s, a, s') entgegennehmen und den skalaren Belohnungswert r zurückliefern. Die Schnittstellendefinition ist in Quellcode-Ausschnitt 4.1 dargestellt.

```
class BaseReward(abc.ABC):
    def __init__(self, name='BASE_REWARD'):
        self.name = name

    @abc.abstractmethod
    def __call__(self, s0, a, s1):
        pass
```

Quellcode-Ausschnitt 4.1: Abstrakte Basisklasse für Belohnungsfunktionen.

Die Schnittstelle ist simpel gehalten: nur die Initialisierung und das Überschreiben der Objektaufrufmethode sind vorgegeben. Da Python abstrakte Klassen und Schnittstellen nicht nativ unterstützt, wird dafür das Modul `abc` (engl. abstract base class, abstrakte Basisklasse) aus der Python-Standardbibliothek verwendet.

4.3.2 Lernverfahren

Die `Model`-Schnittstelle bietet das Modell zur Approximation der Q-Funktion im Q-Lernen. Dabei wird die Methode *update_model* zum Optimieren der Q-Funktion genutzt. *predict_one* und *predict_batch* dienen dazu, die optimale Aktion für einen oder mehrere Zustände vorauszusagen. Es wurden zwei Klassen aus der Schnittstelle konkretisiert: das klassische Q-Lernen mit einer Zustandstabelle und das Q-Lernen mit tiefen neuronalen Netzwerken als Q-Funktions-Approximator.

Tabellarisches Q-Lernen

Die `RCQTable`-Klasse implementiert das Q-Lernen nach Watkins und Dayan [24]. Um dies zu ermöglichen, ist es nötig, eine Zuordnung von allen Zuständen und Aktionen zu deren Wert abspeichern zu können. Zwei Implementierungen für diese Zuordnung wären hierbei naheliegend. Zum einen könnte ein mehrdimensionales Feld angelegt werden, wobei jede Dimension des Zustandsraums mit einer Dimension dieses Feldes korrespondiert. Zum

anderen könnte ein assoziatives Datenfeld, zum Beispiel eine Hashtabelle, dazu genutzt werden, diese Übersetzung von Zuständen implizit zu übernehmen.

Dadurch, dass kein Hashwert für jeden Zustand berechnet werden muss, könnte die Geschwindigkeit des ersten Ansatzes schneller sein als die des Zweiten. Dennoch wurde als Datenstruktur ein assoziatives Datenfeld, konkret die Python-Klasse `defaultdict` aus dem Modul `collections` der Python-Standardbibliothek, gewählt. Zum einen ist die Handhabung dieser Datenstruktur deutlich simpler. Zum anderen wurde angenommen, dass diese auf modernen Prozessoren bei dem gegebenen Anwendungsfall keinen spürbaren Unterschied in der Performanz zu einem Feld aufweisen würde. Die `defaultdict`-Klasse ermöglicht es, zuvor unbekannt Zustandswerten einen Standardwert zuzuweisen, wodurch eine Operation wie $Q[s] = Q[s] + 1$ nicht zu einem Zugriffsfehler führt, ohne dass zuvor Q für alle $s \in S$ initialisiert wurde.

update_model

Diese Methode erhält ein Quadrupel (s, a, r, s'), bestehend aus Ausgangszustand, Aktion, Belohnungswert und Folgezustand, und wendet die Regel zum Optimieren der Q-Tabelle an:

```
def update_model(s0, a, r, s1):
    Q[s0][a] = Q[s0][a] + alpha * (r + gamma * np.amax(Q[s1]) - Q[s0][a])
```

Quellcode-Ausschnitt 4.2: Optimierung der Q-Tabelle.[2]

Dabei ist `np.amax` eine Funktion der Bibliothek NumPy, welche das maximale Element eines Feldes zurückgibt. Das heißt in diesem konkreten Fall, dass der Wert der schätzungsweise besten Aktion des Folgezustands gewählt wird [18, S. 131].

Tiefe Q-Netzwerke

Um das tiefe optimierende Lernen zu ermöglichen, wurde die Klasse DQNModel implementiert. Sie fasst das neuronale Netzwerk mit dessen Operationen und die Hyperparameter

[2] Zur verbesserten Lesbarkeit wurde der Verweis zur aktuellen Instanz ausgelassen. Symbole, die nicht zu den Parametern gehören, sind implizit Instanzvariablen.

des Lernprozesses zusammen.

define_model

Diese Methode wird im Rahmen des Initialisierungsprozesses der Klasse aufgerufen und definiert das neuronale Netzwerk sowie die Operationen, die damit durchgeführt werden können.

```
def define_model(self, learning_rate):
    # Placeholders for state and value inputs
    state_input = tf.placeholder(shape=[None, n_states],
       dtype=tf.float32)
    current_q = tf.placeholder(shape=[None, n_actions], dtype=tf.float32)
    # Two fully connected hidden layers
    hidden1 = tf.layers.dense(state_input, 64, activation=tf.nn.relu)
    hidden2 = tf.layers.dense(hidden1, 64, activation=tf.nn.relu)
    # Fully connected output layer
    output = tf.layers.dense(hidden2, n_actions)
    # Optimization
    loss = tf.losses.mean_squared_error(current_q, output)
    optimizer = tf.train.AdamOptimizer(learning_rate).minimize(loss)
    # Weight initialization for all layers
    var_init = tf.global_variables_initializer()
```

Quellcode-Ausschnitt 4.3: Definition eines neuronalen Netzwerkmodells.

In der Eingabeschicht des neuronalen Netzwerks korrespondiert jedes Neuron mit einer festen Dimension des Zustandsraums, daher ist die Anzahl der Eingabeneuronen gleich der Zahl der Dimensionen des Zustandsraums. Darauf gefolgt sind zwei versteckte Schichten mit je 64 Neuronen. Die Ausgabeschicht hat so viele Knoten wie es erlaubte Aktionen in der aktuellen Problemstellung gibt. Die Fehlerwerte in der Q-Funktion sollen minimiert werden, wozu der Adam-Optimierer [33] verwendet wird.

update_model

Für das Lernverfahren DQN sind mehr Schritte nötig, um die Q-Funktion zu verbessern, als beim klassischen Q-Lernen. Die Implementierung des `update`-Algorithmus ist in Quellcode-Ausschnitt 4.4 gezeigt.

```
def update(transition):
    replay.add(transition)
    batch = replay.sample(batch_size)
    initial_states = [t[0] for t in batch]
    next_states = [t[3] if t[3] is not None else np.zeros(n_actions)
                   for t in batch]
    q_initial = predict_batch(initial_states)
    q_next = predict_batch(next_states)
    x = np.zeros((len(batch), n_states))
    y = np.zeros((len(batch), n_actions))
    for i, b in enumerate(batch):
        x[i], a, r, s1 = b
        y[i] = q_initial[i]
        if s1 is None:
            y[i][a] = r
        else:
            y[i][a] = r + gamma * np.amax(q_next[i])
    return train_batch(x, y)
```

Quellcode-Ausschnitt 4.4: Methode zur Optimierung der Q-Funktion mit DQN.

Zunächst wird der neue Zustandsübergang dem Speicher hinzugefügt. Aus diesem wird anschließend eine Stichprobe entnommen und in Ausgangs- und Folgezustände aufgeteilt. Diese Werte werden durch das neuronale Netzwerkmodell propagiert, um die aktuell approximierten Q-Werte der Zustände zu errechnen. Anschließend wird ein Trainingsstapel erzeugt, in welchem die errechneten Q-Werte um die erhaltene Belohnung korrigiert sind. Dieser Trainingsstapel aus `x` und `y` für Ausgangszustände und korrigierte Werte wird dann an die `train_batch`-Methode übergeben, welche die Rückpropagierung zum Trainieren des neuronalen Netzwerks durchführt.

Die Methoden `np.zeros` und `np.amax` sind der NumPy-Bibliothek [14] entnommen. `zeros` initialisiert ein mehrdimensionales Feld der angegebenen Form mit Nullwerten. `amax` liefert das maximale Element eines Feldes.

4.3.3 Klasse „Agent“

Die Klasse `Agent` bildet das Herzstück des Lern-Frameworks und verbindet Umgebung und Vorhersagemodell, um das Experiment durchzuführen. Nach der Initialisierung bietet die `run_episode`-Methode eine Schnittstelle zum Durchführen von Episoden.

Initialisierung

Der Konstruktor der Klasse nimmt die für das Experiment relevanten Parameter entgegen. Folgende Werte werden initialisiert:

model Modell der Wertfunktion, mit der die Aktionen des Agenten bestimmt werden

environment Instanz der Ausführungsumgebung, mit der interagiert werden soll

steps_per_episode Maximale Anzahl Zeitschritte pro Episode

epsilon Initiale Wahrscheinlichkeit, in einem Zeitschritt eine zufällige Aktion auszuwählen

epsilon_decay Faktor, um den der ϵ-Wert nach jeder Episode verringert wird

epsilon_min Mindestwahrscheinlichkeit für zufällige Aktionen

run_episode

Diese Methode lässt den Agenten eine Episode in der Lernumgebung durchführen. Der Ablauf einer Episode ist in Abbildung 4.2 dargestellt.

Zu Beginn einer Episode wird die Umgebung zurückgesetzt und der Agent erhält die initiale Beobachtung. Da der Agent mit der ϵ-gierigen Methode Aktionen auswählt, wird eine Zufallszahl gezogen, mit der bestimmt wird, ob in diesem Zeitschritt eine zufällige oder die nach der Wertfunktion erfolgversprechendste Aktion ausgeführt werden soll. Durch

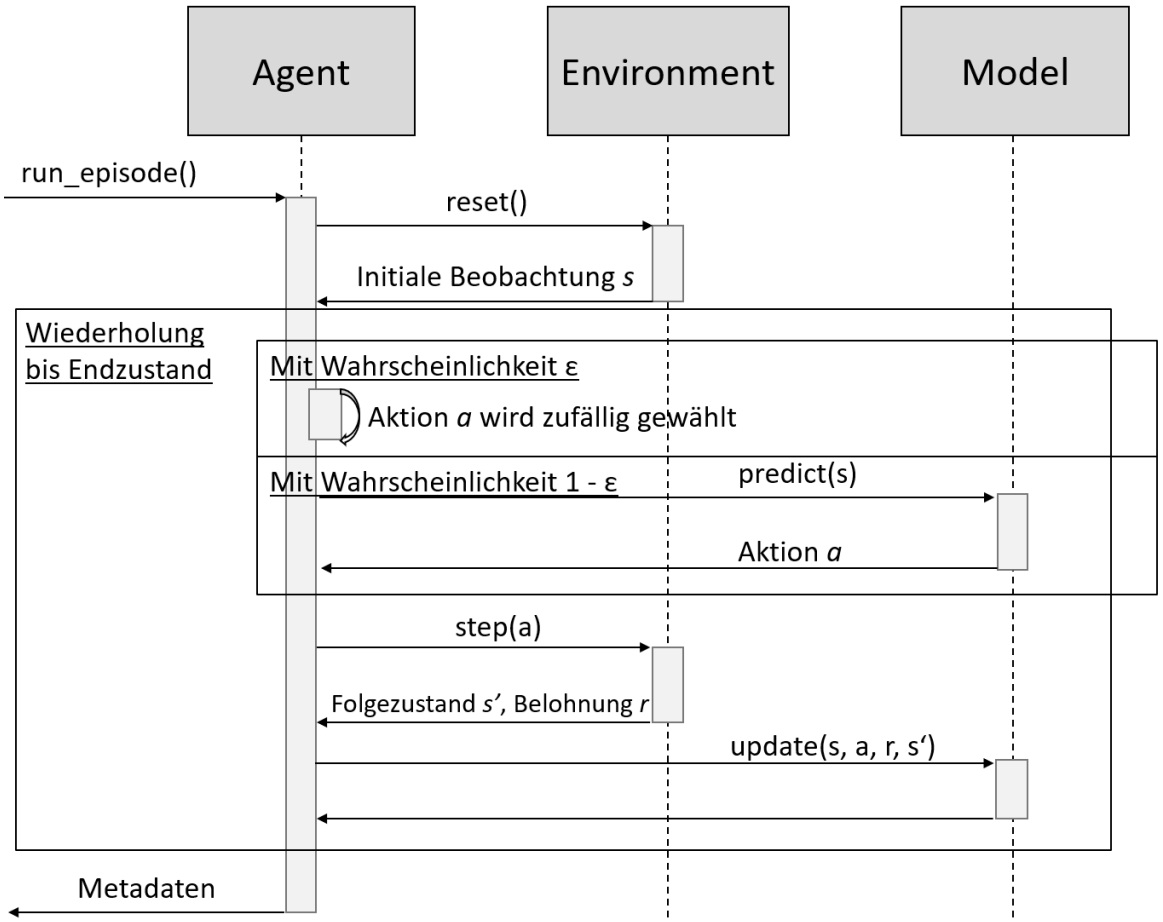

Abbildung 4.2: Ablauf einer Episode im optimierenden Lernen mit ϵ-gieriger Regelungsstrategie. Bildquelle: Eigenes Werk; [18]

Ausführen der Aktion erhält der Agent den Folgezustand und die Belohnung sowie die Information, ob der Folgezustand terminal ist. Das Quadrupel (s, a, r, s') wird dann dem Zustandsübergangsspeicher zugeführt. Anschließend wird noch das Q-Modell trainiert.

Dies wiederholt sich so lange, bis ein Endzustand erreicht ist.

Zu statistischen Auswertungszwecken werden der Anfangs- und Endzustand, die gesammelte Belohnung, die durchschnittlichen Trainingskosten und der ϵ-Wert der Episode zurückgegeben.

4.3.4 Klasse „ExperienceReplay“

Diese Klasse implementiert den in [25] vorgestellten Erfahrungsspeicher, der für DQN verwendet wird. Sie bietet eine Methode zum Speichern eines Zustandsübergangs, `add`, sowie die Methode `sample` zum Ziehen einer Stichprobe aus den gesammelten Erfahrungen.

Als Datenstruktur für den Speicher wurde `collections.deque`, die Implementierung eines Doppelstapels in der Python-Standardbibliothek, gewählt. Dabei wird ausgenutzt, dass sich bei der Instanziierung eine Kapazität festlegen lässt. Beim Einfügen in die `deque` wird das älteste Element verworfen, wenn die Kapazität erreicht ist.

sample

Aus dem Erfahrungsspeicher wird mithilfe der `sample`(n)-Methode eine diskret gleichverteilte Stichprobe mit n Übergängen ohne Zurücklegen gezogen. Dafür wird die Funktion `random.choice` der Python-Bibliothek NumPy [14] verwendet.

Sind noch nicht genügend Erfahrungswerte vorhanden, so werden alle vorhandenen Daten in zufällig angeordneter Weise zurückgegeben.

4.3.5 Serververbindungen

In Abschnitt 2.1.1 wurde bereits die Kommunikation zwischen RoboCup-Server und -Klienten vorgestellt. Um die Verbindung zum Server seitens des Lern-Frameworks zu ermöglichen, wurden die Klassen `Connection` und `CoachConnection` implementiert. Erstere kommuniziert asynchron bidirektional mit dem Server. Der Coach muss nur einzelne Befehle versenden können, die Antworten des Servers werden verworfen.

Klasse „Connection“

Diese Klasse nutzt das `asyncio`-Framework der Python-Standardbibliothek. Dadurch können Daten vom Server im Hintergrund empfangen werden, während andere Rechenvorgänge durch den Agenten durchgeführt werden. Die `Connection`-Instanz speichert den aktuellen Spielzustand, den sie durch den Simulationsserver per UDP erhält. Der Agent kann dann jederzeit diesen Zustand abrufen.

Es wurde eine Methode `step` implementiert, mit welcher eine Nachricht an den Server gesendet werden kann. Diese gibt den Zustand des Spieles erst dann zurück, wenn der Folgezustand verfügbar ist. Dadurch eignet sich die Methode gut für Schritte im optimierenden Lernen.

Klasse „CoachConnection“

Im Gegensatz zu der Spielerverbindung muss sich der Coach, zumindest in diesen simplem Experimenten, nicht um die Antworten des Servers kümmern. Deswegen ist bei dieser Klasse lediglich eine `send`-Methode implementiert, welche eine übergebene Nachricht per UNIX-UDP-Socket an den Simulationsserver sendet und dann terminiert.

4.3.6 Konfiguration

Das `learner`-Modul bietet durch die Unterstützung des `absl.flags`-Moduls einige Kommandozeilenparameter zum Konfigurieren des Lern-Frameworks. Einen Auszug aus den Definitionen der Parameter zeigt Quellcode-Ausschnitt 4.5.

```
flags.DEFINE_string('learning_style',
                    'dqn',
                    'Learning algorithm to use. (dqn|tabular)')
flags.DEFINE_integer('episodes',
                     5000,
                     'Number of episodes to run')
flags.DEFINE_float('learning_rate',
                   1e-3,
                   'Learning rate of neural network updates')
```

Quellcode-Ausschnitt 4.5: Ausschnitt der Definitionen der Kommandozeilenparameter des Lern-Frameworks.

Dadurch lässt sich das Skript zum Beispiel wie folgt aufrufen: `python3 learner.py --learning_style=tabular --episodes 50 --learning_rate=0.01`. Damit würde ein Experiment mit tabellarischem Q-Lernen und einer Lernrate von 0,01 über 50 Episoden gestartet werden.

Das Auswählen der Problemstellungen, die in Kapitel 3 vorgestellt wurden, wird über den Parameter `problem` ermöglicht. Dazu ist in der Hauptdatei des Lern-Frameworks das assoziative Datenfeld `PROBLEMS` definiert, welches die Namen den Problemkonfigurationen zuordnet. Diese Definition ist ausschnittsweise in Quellcode-Ausschnitt 4.6 gezeigt.

```
PROBLEMS = {
    ...
    'run_to_point_turn_dash': {
        'reward': rewards.ShapedRunToPoint(point=(47., 0.)),
        'observations': ['self'],
        'actions': [p_cmd.Dash(100, 0), p_cmd.Turn(90), p_cmd.Turn(-90)]
    },
    ...
}
```

Quellcode-Ausschnitt 4.6: Ausschnitt aus den Problemkonfigurationen des Lern-Frameworks, korrespondierend zur Problemstellung in Abschnitt 3.3.

Belohnungsfunktion, benötigte Zustandsdimensionen und verfügbare Aktionen werden hier spezifiziert. Die assoziative Datenstruktur lässt sich einfach um weitere Einträge erweitern, sollten neue Problemstellungen definiert werden.

Das in dem Quellcode-Ausschnitt genutzte Modul `p_cmd` ist eine selbst entwickelte Abstraktion für die Kommandos des *rcssserver*.

5 Ergebnisse

In diesem Kapitel werden die Ergebnisse der Arbeit vorgestellt. Zunächst wird die Performanz des Lern-Frameworks kurz diskutiert. Anschließend werden die Resultate verschiedener Experimente präsentiert, die mit dessen Hilfe durchgeführt wurden.

5.1 Lern-Framework

Das Lern-Framework wurde nach erfolgreicher Implementierung in einigen Versuchen eingesetzt und erwies sich als zuverlässig für den Anwendungszweck. Bisher funktioniert die Software gut für Versuche mit einzelnen Agenten. Möchte man weitere Agenten hinzufügen, müsste man `Environment`-Klasse und Serverbindungen entkoppeln.

Eine Analyse während eines Experiments hat ergeben, dass etwa die Hälfte der Zeit für das Training des DQN-Modells (`update_model`-Methode der Klasse `DQNModel`) aufgewendet wird. Ein Viertel der gesamten Ausführungszeit benötigt das Stoppen der Ausführungsumgebung. Die direkte Kommunikation mit dem Server nimmt nur etwa 14% der Zeit ein[1].

5.2 Ergebnisse der Versuche

In Kapitel 3 wurden mehrere Problemstellungen vorgestellt, die es durch optimierendes Lernen zu lösen galt. Die Implementierung der Lernverfahren wurde im vorherigen Kapitel vorgestellt. Im Folgenden werden nun die auf dieser Grundlage durchgeführten Experimente vorgestellt und ihre Ergebnisse diskutiert.

Alle Experimente wurden auf einem Rechner mit einem Intel Core i7-8086k Hauptprozessor mit 12 logischen Kernen und einer Taktrate von 4,0 GHz ausgeführt. Die Optimierung der neuronalen Netzwerkmodelle mittels TensorFlow wurde durch einen Nvidia GeForce RTX 2080 Grafikprozessor beschleunigt.

[1] Die Rohdaten der Analyse können dem Aufrufgraphen unter dem Dateinamen *python-profiling.png* entnommen werden, welcher sich auf dem beigelegten Datenträger befindet.

5.2.1 Erzeugung der Lernkurven

Die in den folgenden Abschnitten gezeigten Grafiken sollen den Lernfortschritt des Agenten visualisieren. Da nicht alle Versuche mit dem gleichen Ziel ausgeführt wurden, werden hier kurz die diversen Funktionen vorgestellt, mit denen die Lernkurven erzeugt wurden.

Bei Arbeiten mit optimierendem Lernen wird häufig der pro Episode akkumulierte Belohnungswert visualisiert. Dies ist bei den durchgeführten Versuchen unpraktisch, da die Belohnungsfunktionen den Agenten bereits für den Weg zum Ziel so stark belohnen, dass alleine aus dem Belohnungssignal nicht abgelesen werden kann, ob das Ziel in der Episode erreicht wurde.

Alle Lernkurven wurden mit Matplotlib [15] in Python erstellt.

Laufexperimente: Geglätteter normierter Positionsfehler im Endzustand

Bei diesen Graphen wird der euklidische Abstand zwischen Zielpunkt und Spieler im Endzustand jeder Episode abgebildet. Aus jeweils zehn Episoden wird der Durchschnitt gebildet, sodass die Grafik den groben Lernerfolg abbilden kann.

Sei P_i^{final} die Spielerposition im Endzustand von Episode i und $P_{i_{\text{ziel}}}$ die Zielposition in Episode i. Dann wird der Graph mit folgender Funktion erzeugt:

$$f(x) = \frac{1}{10} \sum_{i=x-9}^{x} |P_i^{\text{final}} - P_i^{\text{ziel}}| \text{ mit } x \in \{10, 20, ..., N\}.$$

Torschussexperimente: Anzahl der Tore

Für die Experimente, bei denen ein Torschuss erzielt werden sollte, wurden in der Lernkurve die in fünfzig Episoden erzielten Tore abgebildet. Da Episoden bei einem gefallenen Tor terminieren, ist der Maximalwert hier 50.

Sei $T_i = 1$ wenn in Episode i ein Tor fiel und $T_i = 0$ wenn nicht, dann wird die Lernkurve bei diesen Experimenten wie folgt erzeugt:

$$f(x) = \sum_{i=x-49}^{x} T_i \text{ mit } x \in \{50, 100, ..., N\}$$

5.2.2 Omnidirektionaler Lauf mit tabellarischem Q-Lernen

Bei den Experimenten mit Problemstellung 1 (Abschnitt 3.1) handelt es sich um Testläufe für die Lernumgebung. Das Ziel sollte sein, zu zeigen, dass die Schnittstellen der Umgebung korrekt implementiert sind und ein erfolgreicher Lernprozess möglich ist. Ein Experiment von 500 Episoden mit je 200 Simulationsschritten mit Lernrate $\alpha = 0{,}001$ und Diskontierungsfaktor $\gamma = 0{,}8$ wurde durchgeführt. Dabei wurde der Spieler zu Beginn jeder Episode an einen zufälligen Punkt der linken Spielfeldhälfte gesetzt und hatte das Ziel, den fixen Punkt $(-47, 0)$, die Mitte der eigenen Torraumlinie, zu erreichen. Dieser Versuch resultierte in der in Abbildung 5.1 gezeigten Lernkurve.

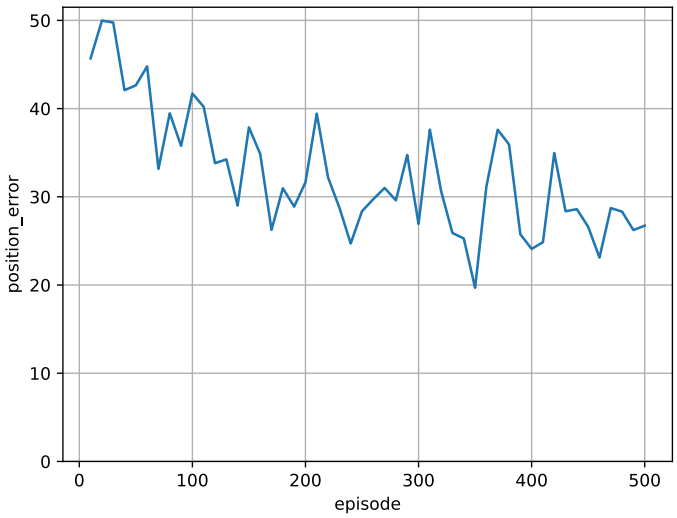

Abbildung 5.1: Lernfortschritt im Experiment mit Q-Lernen. Die X-Achse zeigt die Anzahl der Episoden, die Y-Achse den Positionsfehler am Ende der Episode für je zehn Episoden gemittelt. Bildquelle: Eigenes Werk

In den 500 Episoden konnte nicht erlernt werden, die Zielposition von beliebigen Startpositionen aus anzusteuern. Da jedoch ein deutlicher Fortschritt im Verlauf der Episoden

sichtbar ist, kann von einem erfolgreichen Testlauf für die Umgebung gesprochen werden.

5.2.3 Omnidirektionaler Lauf mit DQN

Der vorherige Abschnitt zeigt, dass das Q-Lernen prinzipiell für die Problemstellung des omnidirektionalen Laufs im Roboterfußball geeignet ist. In dieser Arbeit soll jedoch primär der Algorithmus DQN [23] in der RoboCup-Simulation evaluiert werden.

Hier wird nun der Lernverlauf eines Experiments vorgestellt, bei dem für das fast identische Problem (vgl. Abschnitt 3.1 und 3.2) das DQN-Lernverfahren verwendet wurde. Es wurden 1000 Episoden mit je 200 Zeitschritten durchgeführt. Das tiefe Q-Netzwerk wurde mit einer Lernrate $\alpha = 0{,}001$, einem Diskontierungsfaktor $\gamma = 0{,}9$ und 1024 Datensätzen pro Schritt optimiert. Der ϵ-Faktor in Episode i betrug $0{,}99^{i-1}$, mindestens jedoch 0,05.

Abbildung 5.2 zeigt den Verlauf des Positionsfehlers in diesem Experiment, bei dem die gleichen Startbedingungen und der gleiche Zielpunkt wie im vorherigen Experiment gelten.

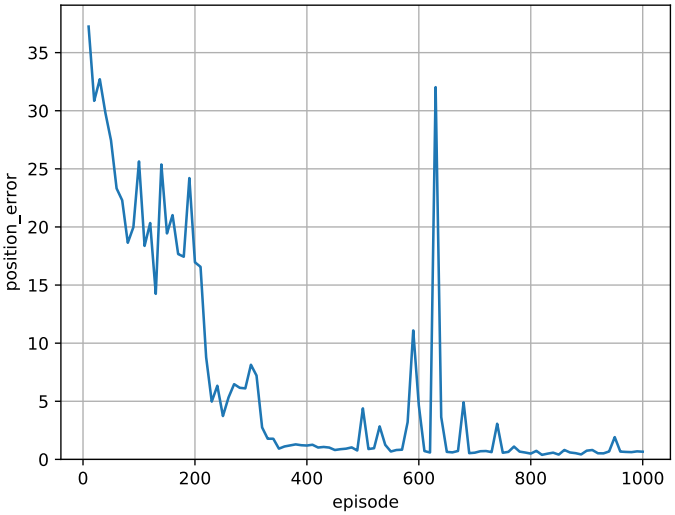

Abbildung 5.2: Lernfortschritt im ersten Experiment mit DQN. Die X-Achse zeigt die Anzahl der Episoden, die Y-Achse den Positionsfehler am Ende der Episode für je zehn Episoden gemittelt. Bildquelle: Eigenes Werk

In dieser Abbildung ist zu erkennen, dass der Agent nach etwas über dreihundert Episoden bereits den Zielpunkt zuverlässig erreichen kann. Danach wird die Strategie instabil,

wodurch hohe Fehlerwerte gemessen werden. Gegen Ende des Experiments gelingt es dem Agenten jedoch wieder, die Zielposition in den meisten Fällen zu erreichen.

Abbildung 5.3 zeigt eine anspruchsvollere Variation des Versuchs, bei dem auf die gleiche Art gelernt wurde, der Zielpunkt jedoch die gegnerische Torraumlinie anstelle der eigenen war und der Spieler zu Beginn jeder Episode an einen zufälligen Punkt im gesamten Spielfeld gesetzt wurde.

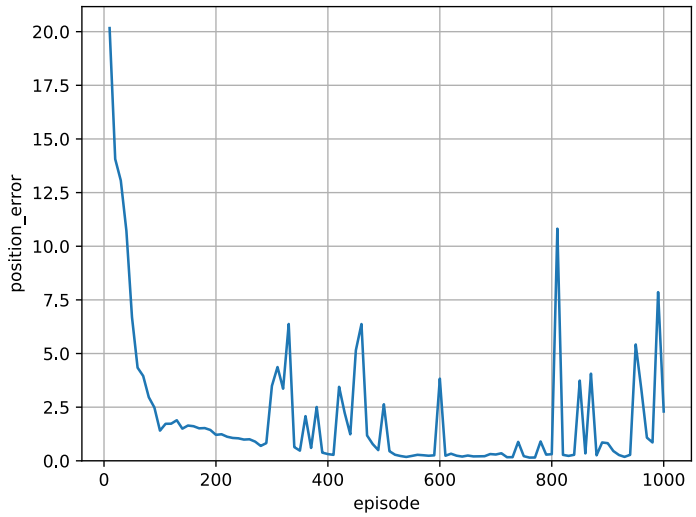

Abbildung 5.3: Lernfortschritt eines DQN-Experiments mit dem Problem des omnidirektionalen Laufs mit fester Zielposition in der gegnerischen Spielfeldhälfte. Entlang der X-Achse verlaufen die Episoden, die Y-Achse zeigt den durchschnittlichen Positionsfehler am Ende der Episode für je zehn Episoden gemittelt. Bildquelle: Eigenes Werk

Dieser Versuch lief ebenfalls erfolgreich, sodass nach 500 Episoden der Zielpunkt in den meisten Fällen erreicht wurde. Eine manuelle Auswertung von Stichproben der Segmente mit starken Schwankungen hat ergeben, dass auch zwischen 300 und 500 sowie 800 und 1000 Episoden in den meisten Fällen der Zielpunkt direkt angesteuert wird, jedoch in einzelnen Episoden eine vollkommen falsche Richtung eingeschlagen wird, wodurch die Kurve verzogen wird.

5.2.4 Vorwärtslauf und Drehung mit DQN

Bei diesen Versuchen sollte das zuvor untersuchte Problem noch weiter modifiziert werden. Während der Agent durch die Natur der Belohnungsfunktion bei allen verfügbaren Aktionen der vorherigen Problemstellungen einen Belohnungswert ungleich null erhielt, erhält er hier bei zwei von drei Aktionen, den Drehungen, keine direkte Belohnung.

Abbildung 5.4 zeigt die Lernkurve eines Versuchs mit $\epsilon = 0{,}995^{i-1}$ in Episode i und ansonsten den gleichen Hyperparametern wie im ersten Versuch von Abschnitt 5.2.3 mit der Problemstellung von Abschnitt 3.3. Das Experiment lief hier über 5000 Episoden.

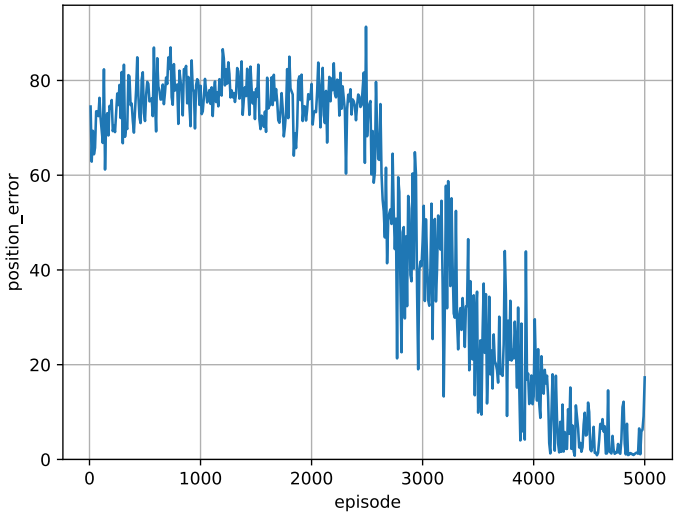

Abbildung 5.4: Lernfortschritt im dritten Experiment mit DQN. Die X-Achse zeigt die Anzahl der Episoden, die Y-Achse den durchschnittlichen Positionsfehler im Endzustand der letzten zehn Episoden. Bildquelle: Eigenes Werk

In dieser Lernkurve ist gut zu sehen, dass der Agent zunächst viele Erfahrungen sammeln musste, um das Problem lösen zu können. Nach 2500 Episoden zeichnet sich eine stetige Verbesserung der Regelungsstrategie ab. Nach über 4000 Episoden erreicht er die Zielposition in den meisten Fällen.

5.2.5 Torschuss mit DQN

Zuletzt wird noch der Fortschritt eines Experiments vorgestellt, in dem der Agent mit dem Ball interagieren muss, um das Ziel zu erreichen, diesen in das gegnerische Tor zu schießen.

Dazu wurde der Ball zufällig in der gegnerischen Hälfte platziert und der Spieler an einem zufälligen Punkt in der Nähe des Balls. Die Episoden endeten nach maximal 500 Zeitschritten. Bis auf $\epsilon = 0.995^{i-1}$ in Episode i wurden alle Hyperparameter des Lernverfahrens von den vorherigen Experimenten übernommen. Für die Belohnungsfunktion wurden die Gewichtungsparameter $w_1 = 100$ für den Punktestand, $w_2 = 0{,}25$ für die Annäherung des Spielers an den Ball und $w_3 = 1$ für die Annäherung des Balls an das Tor gewählt.

Die Ergebnisse des Versuchs sind in Abbildung 5.5 gezeigt.

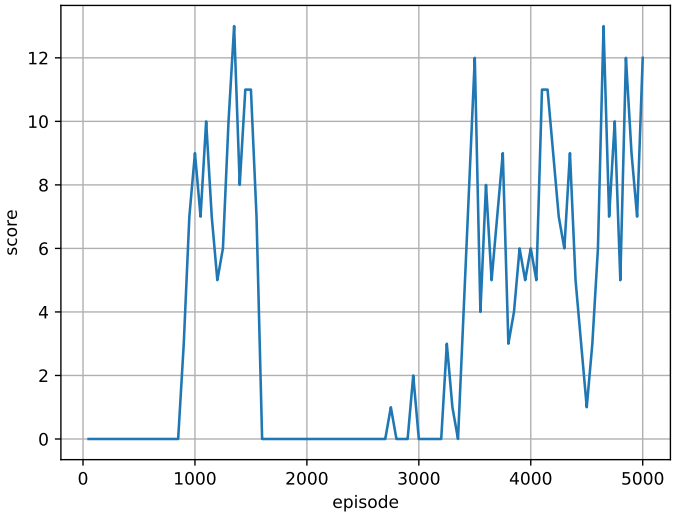

Abbildung 5.5: Lernfortschritt eines Experiments, bei dem der Agent Tore schießen sollte. Die X-Achse zeigt die Anzahl der Episoden, die Y-Achse die Summe von Toren über je 50 Episoden. Bildquelle: Eigenes Werk

Hier sieht man, dass der Agent es unter diesen Bedingungen nach fast 5000 Episoden des Lernens in etwa 20% der Fälle schafft, zum Ball zu laufen, sich korrekt zu positionieren und den Ball ins Tor zu schießen.

6 Ausblick

In dieser Arbeit wurde der Nachweis der Machbarkeit von Ende-zu-Ende-Lernen mit DQN im simulierten Roboterfußball durch verschiedene Experimente erbracht. Das Ziel, eine komplette Mannschaft von elf Spielern in dieser Weise zu trainieren, ist jedoch noch nicht erreicht. Welche Schritte als nächstes unternommen werden könnten, wird in diesem Kapitel diskutiert.

6.1 Lern-Framework

Die potenziellen Möglichkeiten der entwickelten Software sind mit den in dieser Arbeit gezeigten Experimenten bei weitem nicht erschöpft. Aus diesem Grund wäre es denkbar, die Software weiterzuentwickeln und frei zugänglich zu machen, damit auch andere Forschungsgruppen sich mithilfe einer simplen Schnittstelle mit dem simulierten Roboterfußball beschäftigen können.

Würde man annehmen, es könnte mit dem Lern-Framework eine Regelungsstrategie gelernt werden, mit der die Spieler als Mannschaft gegen eine andere Mannschaft spielen könnten, so wäre es im aktuellen Zustand der Software nicht möglich, im Turnierwettkampf damit anzutreten. Die Spiele der 2D-Simulationsliga des RoboCup werden immer im asynchronen Modus ausgetragen. Die Unterstützung für diesen ist im Lern-Framework aktuell nicht vorhanden und müsste implementiert und getestet werden. Zwar unterstützen viele Teams den synchronen Modus technisch gesehen, jedoch sind einige von ihnen darauf ausgelegt, Berechnungen bis zum Ablaufen des Simulationsschritts durchzuführen, wodurch ihnen im synchronen Modus Nachteile entstehen könnten.

Für die Unterstützung des asynchronen Modus würde es sich anbieten, die Methode zum Senden von Nachrichten über die `Connection`-Klasse vom Erhalt der Zustandsnachricht zu entkoppeln. Vorbild hierfür könnte die Implementierung aus Agent2D [8] sein.

6.2 Problemstellungen

Das finale Ziel soll es sein, den simulierten Roboterfußball mithilfe von Ende-zu-Ende-Lernen zu meistern. Dazu müssen die Problemstellungen Schritt für Schritt an die Wettkampfbedingungen des RoboCup angepasst werden. Versuche mit mehreren Agenten

können der nächste logische Schritt sein. Für diese ist jedoch unter Umständen eine Anpassung der Belohnungsfunktionen notwendig. Des Weiteren wird im Wettkampf nicht der *fullstate*-Modus des Servers aktiviert, weswegen die Agenten anhand von Sensorinformationen trainiert werden müssen.

6.2.1 Fokus auf Multiagentenexperimente

In dieser Arbeit wurden, um den Umfang etwas einzuschränken, nur Versuche mit einzelnen Agenten durchgeführt. Damit das System um weitere Agenten erweitert werden kann, müssen nur wenige Anpassungen vorgenommen werden. Die `Environment`-Klasse müsste so verändert werden, dass mehrere Verbindungen gleichzeitig mit dem Server möglich sind.

Dadurch könnte untersucht werden, wie gut das verwendete Lernverfahren im Vergleich zu bewährten Agenten wie FRA-UNIted [6] oder Agent2D [8] abschneidet.

6.2.2 Belohnungsfunktionen

In allen Problemstellungen wurde die Belohnungsfunktion so gewählt, dass der Agent durch sie an das Ziel herangeführt wird. Diese Belohnungsfunktionen sind suboptimal, weil mit ihnen nicht gut generalisiert werden kann [18, vgl. Kapitel 17.4].

Die genutzte Belohnungsfunktion, mit der erfolgreich das Schießen von Toren gelernt werden konnten, gibt sowohl für die Annäherung des Spielers an den Ball als auch für die Annäherung des Balls an das Tor einen positiven Belohnungswert. In Versuchen mit einem einzelnen Agenten funktioniert dies gut, da die Annäherung an den Ball die einzige Möglichkeit für den einzelnen Agenten ist, ein Tor zu schießen. Im Zusammenspiel mehrerer Agenten wäre es jedoch wünschenswert, wenn nicht alle dem Ball hinterherrennen würden, sondern sich einzelne auch näher an das Tor begeben, den Ball entgegennehmen und weiterschießen würden.

6.2.3 Zustände

In dieser Arbeit wurde ausschließlich mit vollständigen Beobachtungen der Simulation (*fullstate*-Modus des Servers) gearbeitet. In einer Weiterentwicklung könnte die Möglichkeit geboten werden, mit partiellen Beobachtungen, also den simulierten Sensorinformationen des Simulationsservers, umzugehen. Dort würde sich anbieten, Lernverfahren zu verwenden, welche die Historie der Zustände berücksichtigen [18, vgl. Kapitel 17.3].

6.3 Fazit

Die RoboCup-Fußballsimulation bietet eine Herausforderung für Algorithmen des maschinellen Lernens. Der DQN-Algorithmus wurde im Rahmen dieser Bachelorarbeit erfolgreich verwendet, um einfache Problemstellungen im RoboCup-Umfeld zu lösen. Hier kann noch vieles optimiert werden, unter anderem wurden die Hyperparameter des Lernverfahrens und die Länge der Experimente empirisch gewählt.

Dennoch konnte erreicht werden, dass ein Agent in überschaubarer Zeit selbstständig und ohne externes Weltmodell erlernen konnte, Tore im simulierten Roboterfußball zu schießen.

Quellen

[1] Minora Asada. „A Report on RoboCup 2017 [Competitions]“. In: *IEEE Robotics and Automation Magazine* 24 (Dez. 2017), S. 21–23. DOI: 10.1109/MRA.2017.2757720.

[2] Noda Itsuki. „Soccer Server: a simulator of RoboCup“. In: *In JSAI AI-Symposium 95: Special Session on RoboCup*. 1995.

[3] Hiroaki Kitano u. a. „RoboCup: A Challenge Problem for AI“. In: *1.* 18.1 (März 1997), S. 73. ISSN: 2371-9621. DOI: 10.1609/aimag.v18i1.1276.

[4] Mao Chen u. a. *RoboCup Soccer Server – for Soccer Server Version 7.07 and later*. URL: https://rcsoccersim.github.io/rcssserver-manual-20030211.pdf.

[5] Thomas Gabel und Martin Riedmiller. „Brainstormers 2D - Team Description 2009“. In: *Supplementary material to RoboCup 2009* (Juni 2009).

[6] Thomas Gabel, Philipp Klöppner und Eicke Godehardt. „FRA-UNIted - Team Description 2018“. In: *Supplementary material to RoboCup 2018: Robot Soccer World Cup XXII* (Juni 2018).

[7] M. Riedmiller und H. Braun. „A direct adaptive method for faster backpropagation learning: the RPROP algorithm“. In: *IEEE International Conference on Neural Networks*. März 1993, 586–591 vol.1. DOI: 10.1109/ICNN.1993.298623.

[8] Hidehisa Akiyama und Tomoharu Nakashima. „HELIOS Base: An Open Source Package for the RoboCup Soccer 2D Simulation“. In: *SpringerLink* (Juni 2013), S. 528–535. DOI: 10.1007/978-3-662-44468-9_46.

[9] Guido van Rossum und Fred L. Drake. *The Python Language Reference Manual*. Network Theory Ltd., 2011. ISBN: 9781906966140.

[10] Caleb Hattingh. *Using Asyncio in Python 3*. Place of publication not identified: O'Reilly Media, Inc, 2018. ISBN: 9781491999691.

[11] Martin Abadi u. a. „TensorFlow: A system for large-scale machine learning“. In: *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI*

16). 2016, S. 265–283. URL: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf.

[12] John Nickolls u. a. „Scalable Parallel Programming with CUDA“. In: *Queue* 6.2 (März 2008), S. 40–53. ISSN: 1542-7730. DOI: 10.1145/1365490.1365500. URL: http://doi.acm.org/10.1145/1365490.1365500.

[13] S. Shi u. a. „Benchmarking State-of-the-Art Deep Learning Software Tools“. In: *2016 7th International Conference on Cloud Computing and Big Data (CCBD)*. Nov. 2016, S. 99–104. DOI: 10.1109/CCBD.2016.029.

[14] Travis E. Oliphant. *Guide to NumPy*. 2nd. USA: CreateSpace Independent Publishing Platform, 2015. ISBN: 9781517300074.

[15] J. D. Hunter. „Matplotlib: A 2D graphics environment“. In: *Computing in Science & Engineering* 9.3 (2007), S. 90–95. DOI: 10.1109/MCSE.2007.55.

[16] Greg Brockman u. a. „OpenAI Gym“. In: *arXiv* (Juni 2016). eprint: 1606.01540. URL: https://arxiv.org/abs/1606.01540.

[17] *abseil / abseil.io*. [Online; Aufgerufen 24. Okt. 2019]. Sep. 2019. URL: https://abseil.io.

[18] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018. URL: http://incompleteideas.net/book/the-book-2nd.html.

[19] Ian Goodfellow, Yoshua Bengio und Aaron Courville. *Deep Learning*. http://www.deeplearningbook.org. MIT Press, 2016.

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

[21] Gerald Tesauro. „Temporal Difference Learning and TD-Gammon“. In: *Commun. ACM* 38.3 (März 1995), S. 58–68. ISSN: 0001-0782. DOI: 10.1145/203330.203343. URL: http://doi.acm.org/10.1145/203330.203343.

[22] Martin Riedmiller. „Neural Fitted Q Iteration – First Experiences with a Data Efficient Neural Reinforcement Learning Method“. In: *Machine Learning: ECML 2005*. Hrsg. von João Gama u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, S. 317–328. ISBN: 978-3-540-31692-3.

[23] Volodymyr Mnih u. a. „Playing Atari with Deep Reinforcement Learning“. In: *arXiv* (Jan. 2013). eprint: 1312.5602. URL: https://arxiv.org/abs/1312.5602.

[24] Christopher J. C. H. Watkins und Peter Dayan. „Q-learning“. In: *Mach. Learn.* 8.3-4 (Mai 1992), S. 279–292. ISSN: 0885-6125. DOI: 10.1007/BF00992698.

[25] Long-Ji Lin. „Reinforcement Learning for Robots Using Neural Networks“. UMI Order No. GAX93-22750. Diss. Pittsburgh, PA, USA, 1992.

[26] T. Gabel und M. Riedmiller. „On Experiences in a Complex and Competitive Gaming Domain: Reinforcement Learning Meets RoboCup“. In: *Proceedings of the IEEE Symposium on Computational Intelligence and Games (CIG 2007)*. Honolulu, USA, 2007.

[27] Matthew Hausknecht und Peter Stone. „Deep Reinforcement Learning in Parameterized Action Space“. In: *arXiv* (Nov. 2015). eprint: 1511.04143. URL: https://arxiv.org/abs/1511.04143.

[28] Timothy P. Lillicrap u. a. *Continuous control with deep reinforcement learning*. 2015. arXiv: 1509.02971 [cs.LG].

[29] Shivaram Kalyanakrishnan, Yaxin Liu und Peter Stone. „Half Field Offense in RoboCup Soccer: A Multiagent Reinforcement Learning Case Study“. In: Juni 2006, S. 72–85. DOI: 10.1007/978-3-540-74024-7_7.

[30] Matthew Hausknecht u. a. „Half Field Offense: An Environment for Multiagent Learning and Ad Hoc Teamwork“. In: *AAMAS Adaptive Learning Agents (ALA) Workshop*. Singapore, Mai 2016.

[31] Max Jaderberg u. a. „Human-level performance in 3D multiplayer games with population-based reinforcement learning“. In: *Science* 364.6443 (Mai 2019), S. 859–865. ISSN: 1095-9203. DOI: 10.1126/science.aau6249. URL: http://dx.doi.org/10.1126/science.aau6249.

[32] Thomas Gabel u. a. „Communication in Soccer Simulation: On the Use of Wiretapping Opponent Teams“. In: *SpringerLink* (Juni 2018), S. 3–15. DOI: 10.1007/978-3-030-27544-0_1.

[33] Diederik P. Kingma und Jimmy Ba. „Adam: A Method for Stochastic Optimization“. In: *arXiv* (Dez. 2014). eprint: 1412.6980. URL: https://arxiv.org/abs/1412.6980.

Eidesstattliche Erklärung

Ich, Philipp Klöppner, Matrikel-Nr. 1183638, versichere hiermit, dass ich meine Bachelorarbeit mit dem Titel *Untersuchungen zum Ende-zu-Ende-Lernen im simulierten Roboterfußball mit tiefem optimierendem Lernen* selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe, wobei ich alle wörtlichen und sinngemäßen Zitate als solche gekennzeichnet habe. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Frankfurt am Main, 25. Oktober 2019

Philipp Klöppner