h_da hochschule darmstadt informatik

member of EUT+ EUROPEAN UNIVERSITY OF TECHNOLOGY

Hochschule Darmstadt

– Fachbereich Informatik –

Robustheit von symbolischen Algorithmen und Reinforcement Learning: Eine Fallstudie mit Vier Gewinnt

Abschlussarbeit zur Erlangung des akademischen Grades

Bachelor of Science (B. Sc.)

vorgelegt von

Leo Herrmann

Matrikelnummer: 1111455

Referentin: Prof. Dr. Elke Hergenröther
Korreferent: Adriatik Gashi

1 Kurzfassung

Inhaltsverzeichnis

Abbildungsverzeichnis

Eigenständigkeitserklärung

Ich versichere hiermit, dass ich die vorliegende Arbeit selbständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Die Zeichnungen oder Abbildungen in dieser Arbeit sind von mir selbst erstellt worden oder mit einem entsprechenden Quellennachweis versehen.

Darmstadt, 21.03.2023

Leo Herrmann

2 Einleitung

Fortschreitende Automatisierung durchdringt zahlreiche Bereiche der Gesellschaft, so zum Beispiel die Fertigungsindustrie, das Gesundheitswesen oder den Straßenverkehr. Zwei fundamentale Ansätze sind dabei symbolische Algorithmen und Machine Learning (ML). Die Einsatzbedingungen von Automatisierungssystemen unterscheiden sich häufig von den Bedingungen, unter denen sie entwickelt und getestet werden. Häufig müssen Systeme mit fehlerhaften oder veralteten Informationen arbeiten oder es treten Situationen ein, die bei der Konzipierung der Systeme nicht berücksichtigt werden können. Dabei sinkt die Leistungsfähigkeit dieser Systeme.

Bei Robustheit handelt es sich um eine Eigenschaft von Software, die beschreibt, wie gut sie unter genau solchen veränderten Bedingungen funktioniert. Robustheit ist daher besonders relevant für den Erfolg von Algorithmen und Modellen in der Praxis und wird in der Literatur ausgiebig untersucht [23][24][25].

Bei der Entscheidung zwischen symbolischen Algorithmen und ML-Verfahren sind entscheidende Faktoren in der Regel die Leistungsfähigkeit, Zuverlässigkeit und Komplexität der Lösung und die Nachvollziehbarkeit der Lösungsfindung ([12]; [14], Kapitel 1.1.2; [20], S. 12 f.; [17], S. 5f.). Hierbei stellt sich die Frage, inwiefern bei der Entscheidung zwischen symbolischen und ML-basierten Lösungsansätzen auch Robustheit ein relevantes Kriterium ist, und falls ja, welche Art von Verfahren robuster ist.

Spiele eignen sich zur Untersuchung von Algorithmen und Modellen, weil sie reale Probleme auf kontrollierbare Umgebungen abstrahieren und gleichzeitig reproduzierbare und vergleichbare Messungen ermöglichen. Die Untersuchungen dieser Arbeit erfolgen am Beispiel des Brettspiels Vier Gewinnt, da aus früheren Untersuchungen ersichtlich wird, dass sich dafür sowohl symbolische Lösungsansätze als auch Verfahren aus dem ML-Teilbereich Reinforcement Learning eignen [27][31][2][34][12][37].

Im Rahmen dieser Arbeit wird zunächst Grundlagenforschung zu verbreiteten symbolischen und Reinforcement Learning basierten Ansätzen betrieben. Dabei wird aus beiden Bereichen jeweils ein Verfahren ausgewählt, das sich zur Lösung von Vier Gewinnt und zur Untersuchung der Fragestellung eignet. Diese beiden Verfahren werden anschließend am Beispiel von Vier Gewinnt empirisch auf deren Robustheit untersucht. Dadurch sollen neue Erkenntnisse darüber gewonnen werden, wie Robustheit als Kriterium für Entscheidungen zwischen symbolischen Algorithmen und ML eingesetzt werden kann.

3 Grundlagen

In diesem Kapitel wird durch Literaturrecherche eine fundierte theoretische Basis geschaffen, auf die im weiteren Verlauf dieser Arbeit Bezug genommen wird. Zunächst wird Vier Gewinnt als zu lösendes Problem untersucht und eingeordnet. Anschließend folgt eine Auswahl von jeweils einem algorithmischen und einem Reinforcement Learning basiertem Lösungsansatz. Die Funktionsweise beider Lösungsmethoden wird erklärt. Außerdem werden bestehende Theorien und Definitionen zum Thema Robustheit zusammengetragen. Sie bilden die Grundlage für die Szenarien und Bewertungskriterien in den Experimenten des Hauptteils.

3.1 Vier Gewinnt

Vier Gewinnt ist ein Brettspiel, das aus einem 7 x 6 Spielfeld besteht. Die beiden Spieler werfen abwechselnd einen Spielstein in eine Spalte hinein, der in dieser Spalte bis zur untersten freien Position fällt. Es gewinnt der Spieler, der als erstes vier Spielsteine in einer horizontalen, vertikalen oder diagonalen Linie nebeneinander stehen hat [11].

Bei Vier Gewinnt handelt es sich um ein kombinatorisches Nullsummenspiel für zwei Spieler. Kombinatorische Spiele weisen „perfekte Information“ auf. Das bedeutet, dass alle Spieler zu jeder Zeit den gesamten Zustand des Spiels kennen. So ist es bei vielen Brettspielen der Fall. Kartenspiele hingegen besitzen diese Eigenschaft meistens nicht, weil jedem Spieler die Handkarten ihrer Gegenspieler unbekannt sind. Bei kombinatorischen Spielen sind außerdem keine Zufallselemente enthalten. Die einzige Herausforderung beim Spielen kombinatorischer Spiele besteht darin, unter einer Vielzahl von Entscheidungsoptionen diejenige auszuwählen, die für einen selbst den besten weiteren Spielverlauf verspricht ([6], S. 96-100; [14], Kapitel 4.1).

Bei Zwei-Spieler-Nullsummenspielen verursacht der Gewinn eines Spielers zwangsläufig einen Verlust des anderen Spielers. Die beiden Spieler haben also entgegengesetzte Interessen ([6], S. 100; [5], S. 6). Das bedeutet, dass sich der Erfolg von verschiedenen Lösungsansätzen durch die durchschnittliche Gewinnrate im Spiel gegeneinander bewerten lässt. Bei Nullsummenspielen mit mehr als zwei Personen, kann es passieren, dass, wenn ein Spieler (bewusst oder versehentlich) nicht optimal spielt, ein zweiter Spieler davon profitiert, während ein dritter Spieler dadurch benachteiligt wird. Solche Wechselwirkungen sind bei Zwei-Personen-Nullsummenspielen ausgeschlossen ([6], S. 113 ff.; [28], S. 151 f.). Das macht die Messergebnisse im Hauptteil besser vergleichbar.

3.1.1 Markov Decision Process

Vier Gewinnt lässt sich für beide Spieler jeweils als Markov Decision Process (MDP) modellieren. Dabei handelt es sich um ein Entscheidungsproblem, bei dem es darum geht, unter verschiedenen Entscheidungsmöglichkeiten, die aufeinander folgen und voneinander abhängig sind, die Beste zu wählen. Es hat folgende Bestandteile:

- Zustände S, die den legalen Spielfeldkonfigurationen entsprechen.
- Aktionen $A(s)$, die für bestimmte Zustände erlaubt sind. In Vier Gewinnt existieren sieben verschiedene Aktionen, eine Aktion für jede Spalte, in der ein Spielstein platziert werden kann. Im Laufe des Spiels ändert sich, welche Aktionen möglich sind. Sobald eine Spalte vollständig gefüllt ist, ist es nicht mehr möglich, einen weiteren Stein in dieser Spalte zu platzieren.
- Übergangsmodell $P(s, a, s')$, das die Wahrscheinlichkeit beschreibt, von Zustand s mit Aktion a zu s' zu gelangen. Das Übergangsmodell hängt im Fall von kombinatorischen Spielen von der Strategie der anderen Spieler ab. Bei der Modellierung als MDP wird angenommen, dass sich die Strategie des Gegenspielers im Laufe des Spiels nicht ändert. Das Übergangsmodell ist dem Spieler nicht bekannt.
- Belohnungsfunktion $R(s, a, s')$, die jedem Zustandsübergang eine Belohnung zuordnet. Bei einem Zwei-Spieler Nullsummenspiel wie Vier Gewinnt beträgt dessen Wert 0, solange kein Endzustand erreicht ist. Wenn ein Endzustand erreicht ist, könnte diese Funktion beispielsweise einen positiven Wert zurückliefern, wenn der untersuchte Spieler gewinnt, und einen negativen Wert, wenn der Gegenspieler gewinnt.

Das Ziel bei der Lösung eines MDPs besteht darin, ein Regelwerk zu finden, das jedem Zustand jeweils eine Aktion zuordnet, und dabei zur höchsten erwartbaren Belohnung führt ([28], S. 562 f.).

3.1.2 Komplexität

Nach Victor Allis lässt sich die Komplexität eines Spiels von strategiebasierten Zwei-Spieler-Nullsummenspielen durch ihre Zustandsraum- und Spielbaumkomplexität beschreiben. Die Zustandsraumkomplexität entspricht der Anzahl der verschiedenen möglichen Spielfeldkonfigurationen ab dem Start. Für ein Spiel kann dieser Wert oder zumindest dessen obere Schranke bestimmt werden, indem zunächst alle Konfigurationen

des Spielfelds gezählt, dann Einschränkungen wie Regeln und Symmetrie berücksichtigt werden, und die Anzahl der illegalen und redundanten Zustände von der Anzahl aller möglichen Konfigurationen abgezogen wird ([5], S. 158 f.).

Die Spielbaumkomplexität beschreibt die Anzahl der Blattknoten des Lösungsbaums. Der Spielbaum ist ein Baum, der die Zustände eines Spiels als Knoten und die Züge als Kanten darstellt ([6], S. 102, [28], S. 147). Der Lösungsbaum beschreibt die Teilmenge des Spielbaums, der benötigt wird, um die Gewinnaussichten bei optimaler Spielweise beider Spieler zu berechnen. Die Spielbaumkomplexität lässt sich durch die durchschnittliche Spiellänge und der Anzahl der Entscheidungsmöglichkeiten pro Zug (entweder konstant oder abhängig vom Spielfortschritt) approximieren. Da in den meisten Spielen ein Zustand über mehrere Wege erreicht werden kann, fällt die Spielbaumkomplexität meist wesentlich größer aus als die Zustandsraumkomplexität ([5], S. 159 ff.).

Die Spielbaumkomplexität ist maßgeblich für die praktische Berechenbarkeit einer starken Lösung. Für Tic Tac Toe wurde durch Allis eine obere Grenze für die Spielbaumkomplexität von 362880 ermittelt und eine starke Lösung lässt sich innerhalb von Sekundenbruchteilen berechnen [26]. Für Schach wird die Spielbaumkomplexität auf 10^{31} geschätzt und die Aussichten auf eine starke Lösung liegen noch in weiter Ferne [29].

Für Vier Gewinnt wurde eine durchschnittliche Spiellänge von 36 Zügen und eine durchschnittliche Anzahl von Entscheidungsmöglichkeiten (freie Spalten) von 4 ermittelt. Damit wurde die Spielbaumkomplexität auf $4^{36} \approx 10^{21}$ geschätzt ([5], S. 163).

3.1.3 Lösungsverfahren

Verschiedene Lösungsverfahren von Vier Gewinnt sind bereits ausgiebig untersucht. Das Spiel wurde 1988 von James Dow Allen und Victor Allis unabhängig voneinander mit wissensbasierten Methoden schwach gelöst, was bedeutet, dass für die Anfangsposition eine optimale Strategie ermittelt wurde. Im Fall von Vier Gewinnt kann der Spieler, der den ersten Zug macht, bei optimaler Spielweise immer gewinnen [3][4].

1993 wurde das Spiel von John Tromp auch durch einen Brute-Force Ansatz stark gelöst. Bei dieser Lösung kam Alpha-Beta-Pruning zum Einsatz, um bei einer Zustandsraumkomplexität von 4.531.985.219.092 die optimalen Zugfolgen für beide Spieler zu berechnen. Das hat damals etwa 40.000 CPU-Stunden gedauert [36].

Lösungen, die alle Möglichkeiten durchrechnen, um die optimale Entscheidung zu treffen, sind für den Einsatz in der Praxis aufgrund des hohen Rechenaufwands bei komplexeren Anwendungen auch heute noch selten praktikabel. Aus diesem Grund wird bevorzugt auf gute Heuristiken zurückgegriffen, die den Rechenaufwand minimieren,

aber dennoch gute Ergebnisse liefern ([19], Kapitel 7.6).

Untersuchungen haben gezeigt, dass sich sowohl regelbasierte Algorithmen als auch verschiedene RL-Ansätze eignen, um sogenannte Agents zu entwickeln, die das Spiel selbstständig spielen [2][35][37][34][31][27]. Wissensbasierte Methoden werden in dieser Arbeit nicht näher betrachtet, da sie stark an die jeweiligen Spielregeln gebunden und ihre Eigenschaften schwer zu verallgemeinern sind.

3.2 Symbolische Algorithmen

Bei symbolischen Algorithmen handelt es sich um Methoden zur Lösung von Problemen, indem Daten durch von Menschen interpretierbare Symbole repräsentiert werden und sie durch von Menschen explizit programmierte Regeln verarbeitet werden ([13], S. 4; [17], S. 5 f.; [18]).

In diesem Kapitel werden drei symbolische Algorithmen vorgestellt, die verwendet werden können, um in Spielbäumen erfolgversprechende Entscheidungen zu treffen. Die Algorithmen Minimax und dessen Optimierung Alpha-Beta-Pruning vorgestellt, die auf die Lösung von Spielbäumen ausgerichtet sind ([14], Kapitel 4; [19], Kapitel 7.6 u. 7.8). Darauf folgt die Erklärung der Monte Carlo Tree Search, welche einen allgemeineren Ansatz darstellt ([28], S. 580; [33]).

3.2.1 Minimax

Minimax ist ein Algorithmus, der aus Sicht eines Spielers ausgehend von einem beliebigen Ursprungsknoten im Spielbaum die darauf folgenden Knoten bewertet und den Kindknoten des Ursprungsknotens mit der besten Bewertung zurückgibt. Bei der Bewertung wird davon ausgegangen, dass der Gegner ebenfalls den Zug wählt, der für ihn am günstigsten ist. Der zu untersuchende Spieler versucht, die Bewertung zu maximieren, während der Gegenspieler versucht, sie zu minimieren.

Zunächst werden die Blattknoten des Spielbaums bewertet. Je günstiger ein Spielfeldzustand für den zu untersuchenden Spieler ist, desto größer ist die Zahl, die diesem Zustand zugeordnet wird. In Abhängigkeit der zuvor bewerteten Knoten, werden nun deren Elternknoten bewertet. Ist im betrachteten Zustand der zu untersuchende Spieler am Zug, übernimmt dieser Zustand die Bewertung des Kindknotens mit der höchsten Bewertung. Umgekehrt ist es, wenn der Gegenspieler Spieler am Zug ist. Dann bekommt der zu untersuchte Knoten die Bewertung des Kindknotens mit der niedrigsten Bewertung. Dieser Vorgang wird wiederholt, bis der Ursprungsknoten erreicht ist. Zurückgege-

ben wird der Kindknoten des Ursprungsknotens, dem die größte Bewertung zugeordnet wurde.

Erfolgt die Bewertung anhand der Gewinnchancen, führt das dazu, dass die Wahl des Knotens mit der besten Bewertung auch die Gewinnchancen maximiert. Um die Gewinnchancen zu ermitteln, müssen jedoch alle Knoten des Spielbaums untersucht werden. Die Laufzeit des Algorithmus steigt linear zur Anzahl der zu untersuchenden Knoten und damit bei konstanter Anzahl von Möglichkeiten pro Zug exponentiell zur Suchtiefe. Den gesamten Spielbaum zu durchsuchen, ist daher nur für wenig komplexe Spiele praktikabel. Damit die Bewertung in akzeptabler Zeit erfolgen kann, muss für komplexere Spiele die Suchtiefe oder -breite begrenzt werden und die Bewertung der Knoten muss auf Grundlage von Heuristiken erfolgen ([14], Kapitel 4; [19], Kapitel 7.6).

3.2.2 Alpha-Beta-Pruning

Beim Alpha-Beta-Pruning handelt es sich um eine Optimierung des Minimax-Algorithmus. Dabei werden die Teilbäume übersprungen, die das Ergebnis nicht beeinflussen können, weil bereits absehbar ist, dass diese Teilbäume bei optimaler Spielweise beider Spieler nicht erreicht werden. Alpha-Beta-Pruning liefert dieselben Ergebnisse wie der Minimax-Algorithmus, aber untersucht dabei wesentlich weniger Knoten im Spielbaum.

Dazu werden während der Suche die Werte Alpha und Beta aufgezeichnet. Alpha entspricht der Mindestbewertung, die der zu untersuchende Spieler garantieren kann, wenn beide Spieler optimal spielen. Beta entspricht der Bewertung, die der Gegenspieler bei optimaler Spielweise maximal zulassen wird. Zu Beginn der Suche wird Alpha auf minus unendlich und Beta auf plus unendlich initialisiert.

Alpha wird aktualisiert, wenn für einen Knoten, bei dem der zu untersuchende Spieler am Zug ist, ein Kindknoten gefunden wurde, dessen Bewertung größer ist als das bisherige Alpha. Beta hingegen wird aktualisiert, wenn für einen Knoten, bei dem der Gegenspieler am Zug ist, ein Kindknoten gefunden wurde, dessen Bewertung kleiner ist als Beta.

Sobald bei einem Knoten Alpha größer oder gleich Beta ist, kann die Untersuchung dessen Kindknoten aus folgenden Gründen abgebrochen werden:

- Ist bei diesem Knoten der zu untersuchende Spieler am Zug, hatte der Gegenspieler in einem zuvor untersuchten Teilbaum bessere Chancen, und wird den aktuellen untersuchten Teilbaum nicht auswählen.
- Ist bei diesem Knoten der zu Gegenspieler am Zug, hatte der zu untersuchende

Spieler in eine zuvor untersuchten Teilbaum bessere Chancen, und wird den aktuell untersuchten Teilbaum nicht auswählen ([19], Kapitel 7.8; [14], Kapitel 4.5).

So kann im Vergleich zum Minimax-Algorithmus die Untersuchung von 80% bis 95% der Knoten übersprungen werden. Der Anteil der Knoten, die bei der Untersuchung übersprungen werden können, ist abhängig davon, wie schnell das Fenster zwischen Alpha und Beta verkleinert wird. Wenn die Reihenfolge, in der die Züge untersucht werden, geschickt gewählt wird, kann dies sogar zu einer Reduktion von über 99% führen ([19], Kapitel 7.8). In Schach ist dies beispielsweise möglich, indem Züge früher bewertet werden, je höherwertiger eine im Zug geworfene Figur ist.

Durch Alpha-Beta-Pruning kann der Spielbaum bei gleichbleibender Zeit wesentlich tiefer durchsucht werden, was beim Einsatz von Heuristiken als Bewertungsfunktion zu präziseren Ergebnissen führt. Die Laufzeit ist allerdings weiterhin exponentiell abhängig zur Suchtiefe. Den gesamten Spielbaum zu durchsuchen, um die Bewertung auf Grundlage von tatsächlichen Gewinnaussichten durchzuführen, bleibt bei komplexeren Spielen weiterhin unpraktikabel ([19], Kapitel 7.8).

Heuristische Bewertungsfunktionen sind in der Hinsicht problematisch, als dass sie spezifisch für die Regeln eines Spiels zugeschnitten sein müssen, bzw. dass es für bestimmte Anwendungsfälle keine guten Heuristiken gibt ([14], Kapitel 4.5). Das führt dazu, dass die Eigenschaften von Alpha-Beta-Pruning schwer auf verschiedene Anwendungsfälle übertragbar sind.

3.2.3 Monte Carlo Tree Search

Bei Monte Carlo Tree Search (MCTS) handelt es sich um einen heuristischen Algorithmus, der dazu dient, um in Bäumen, die aus sequentiellen Entscheidungen bestehen, einen möglichst vielversprechenden Pfad auszuwählen. MCTS kann als Lösung für MDPs betrachtet werden ([28], S. 580). Dazu werden wiederholt zufällig verschiedene Entscheidungen simuliert und deren potentieller Erfolg statistisch ausgewertet.

Der Vorteil gegenüber des Alpha-Beta-Prunings besteht darin, dass es bei MCTS nicht notwendig ist, innere Knoten, also Nicht-Blattknoten, zu bewerten (AIAMA, GOG, IEEE). Lediglich die Endzustände müssen bewertet werden können. Dies lässt sich im Gegensatz zur Bewertung von inneren Knoten relativ einfach umsetzen. Bei Spielen bedeutet das die Auswertung des Endergebnisses, also die Punktzahl oder den Gewinner ([28], S. 161; [14], Kapitel 4.5, [7]).

MCTS ist eine weit verbreitete Lösung für kombinatorische Spiele und hat sich insbesondere beim Spiel Go als erfolgreich bewiesen, das einen besonders breiten und tiefen

Spielbaum aufweist, woran frühere Verfahren daran gescheitert sind. Der Algorithmus wird auch abseits von Spielen in verschiedenen Variationen eingesetzt, so zum Beispiel in der Optimierung von Lieferketten und Zeitplanung von Prozessen ([28], S. 161; [7]).

Mit MCTS werden Entscheidungsmöglichkeiten statistisch ausgewertet, indem die vier Phasen Selection, Expansion, Simulation (auch Play-Out) und Backpropagation wiederholt durchlaufen werden. Dabei wird ein Baum verwaltet, der eine ähnliche Struktur wie der Spielbaum aufweist. Die Knoten beschreiben die Zustände der Umgebung und die Kanten Übergänge zwischen den Zuständen. Zu jedem Knoten im MCTS-Baum wird eine Statistik abgespeichert, die Informationen darüber enthält, wie oft der Knoten selbst oder dessen Kindknoten die Simulation-Phase durchlaufen haben, und was die Ergebnisse der Simulationen waren ([28], S. 161 ff.; [14], Kapitel 4.5).

Zu Beginn besteht der MCTS-Baum lediglich aus dem Ursprungsknoten.

In der Phase **Selection** wird zunächst der Ursprungsknoten aus dem MCTS-Baum betrachtet und es werden basierend auf den bisher gesammelten Statistiken so lange Folgeknoten gewählt, bis ein Knoten erreicht wird, der um mindestens einen Folgeknoten erweitert werden kann. Dieser Knoten wird in den noch kommenden Phasen untersucht. Dabei besteht jeweils die Möglichkeit, einen Knoten zu wählen, der vielversprechend erscheint, um genauere Informationen darüber zu erhalten (Exploitation), oder einen Knoten zu wählen, der noch nicht so oft untersucht wurde, um ggf. neue Bereiche im Spielbaum zu erkunden, die bessere Chancen versprechen (Exploration). Es gibt verschiedene Auswahlstrategien, wobei UCT (Upper Confidence Bound applied for Trees) die am weitesten verbreitete ist. Dabei wird die UCB1 Formel eingesetzt, die ursprünglich zur Lösung von Bandit-Problemen (vgl. [28], S. 581 ff.) konzipiert wurde, um die Kinder eines Knotens n zu bewerten. Anschließend wird der Knoten mit der höchsten Bewertung gewählt ([22]; [7]; [28], S. 163). Die Formel lautet wie folgt:

$$UCT = \frac{U(n)}{N(n)} + c_{UCT} * \sqrt{\frac{log(N(Parent(n)))}{N(n)}}$$

Dabei ist $U(n)$ der summierte Wert der Ergebnisse der bisher durchgeführten Simulationen ab Knoten n. In Vier Gewinnt entspricht das der Anzahl, wie oft der zu untersuchende Spieler in den bisher durchgeführten Simulationen gewonnen hat. $N(n)$ entspricht der Anzahl der Simulationen, die bisher ab Knoten n durchgeführt wurden. Damit stellt der Teil links vom plus den Exploitation-Teil dar. Er wächst mit den Erfolgsaussichten des untersuchten Knotens. $N(Parent(n))$ ist Anzahl wie oft Elternknoten von n simuliert wurde, $N(n)$ hingegen die Anzahl wie oft der zu betrachtete Knoten selbst

simuliert wurde. Der Teil hinter dem Plus wird größer, je seltener ein Knoten simuliert wurde und fördert damit die Exploration von bisher selten untersuchten Knoten. Bei c_{UCT} handelt es sich um einen Parameter, über den die Exploitation- und Exploration-Teile der Formel ausbalanciert werden können. Als Richtwert wird hier $\sqrt{2}$ empfohlen ([28], S. 163).

Im Zuge der **Expansion** wird vom zuvor ausgewählten Knoten ein zufälliger Zug ausgeführt und der neue Zustand wird als Kindknoten hinzugefügt. Je nach Spielfeldzustand handelt es sich beim neuen Zug um einen Zug des zu untersuchenden Spielers oder des Gegenspielers. Es können auch mehrere oder alle möglichen Züge auf einmal ausgeführt und als Kindknoten hinzugefügt werden, wenn die durchschnittliche Anzahl der Kindknoten und die verfügbaren Rechenressourcen dies zulassen ([28], S. 162; [7]).

In der **Simulation**-Phase werden ab den zuvor hinzugefügten Knoten so oft zufällige Entscheidungen hintereinander simuliert, bis ein Endzustand erreicht wurde. Es ist dabei zu beachten, dass dabei die getroffenen Entscheidungen nicht in den MCTS-Baum aufgenommen werden ([28], S. 162). Besteht eine Simulation aus besonders vielen Zügen, kann es sinnvoll sein, sie nach einer bestimmten Anzahl von Zügen abzubrechen, und den letzten Zustand heuristisch oder mit einem neutralen Ergebnis zu bewerten([28], S. 164). Es existiert eine Variante, die sogenannte Heavy-Playouts einsetzt, was bedeutet, dass die Entscheidungen in der Simulation nicht rein zufällig getroffen werden, sondern unter Zuhilfenahme von Wissen über das konkret zu lösende Problem. Dadurch werden die Simulationen realistischer [7].

Als letztes erfolgt die Phase **Backpropagation**. Das Ergebnis der Simulation wird für den untersuchten Knoten im MCTS-Baum abgespeichert und die Statistik der Knoten, die vom Ursprungsknoten zum untersuchten Knoten geführt haben, wird aktualisiert. Zurückgegeben wird der Folgeknoten des Ursprungsknoten, der am häufigsten besucht wurde ([28], S. 162).

Mit jeder Wiederholung der vier Phasen wird die Statistik über die Erfolgschancen der Entscheidungsmöglichkeiten akkurater. Nach einer bestimmten Anzahl von Wiederholungen wird basierend auf den gesammelten Statistiken eine Entscheidung getroffen.

Da die Phasen Expansion und Simulation basierend auf Zufall erfolgen, ist MCTS nicht deterministisch und liefert keine perfekten Vorhersagen. Außerdem besteht vor allem bei wenigen Iterationen besteht die Gefahr, dass wenn es wenige Züge gibt, die den Verlauf des Spiels wesentlich beeinflussen, diese Züge unentdeckt bleiben, und die Statistik inakkurat wird ([28], S. 164).

Dadurch, dass Iterationen beliebig oft durchgeführt werden können, um die Vorhersa-

gen zu verbessern, ist die Laufzeit von MCTS schwer mit der von den zuvor genannten Methoden vergleichbar. Untersuchungen zeigen, dass bei kleinen Problemen und der Verfügbarkeit von präzisen Heuristiken Alpha Beta mit begrenzter Suchtiefe schneller und besser arbeitet. MCTS schneidet im Verleich besser ab, je tiefer und stärker verzweigt die zu lösenden Entscheidungsbäume sind ([28], S. 163 f.). Es wurde außerdem gezeigt, dass die Ergebnisse von MCTS unter Verwendung von UCT als Selection-Strategie bei unbegrenzten Ressourcen zu Minimax konvergiert [7].

Es existieren verschiedene Variationen und Verbesserungen für die Strategien in den Phasen Selection und Expansion, die unter bestimmten Umständen für bessere Vorhersagen sorgen. Dazu gehören auch welche, die Machine Learning einsetzen, um in der Selection-Phase fundiertere Entscheidungen zu treffen [7]. Diese werden im Rahmen dieser Arbeit nicht betrachtet. MCTS soll hier klar von Machine Learning Verfahren abgegrenzt sein. Es wird davon ausgegangen, dass die Verbesserungen für die Untersuchung der Robustheit nicht relevant sind, und dass sich die Beobachtungen auch auf Varianten von MCTS übertragen lassen.

3.3 Reinforcement Learning

RL ist ein Teilgebiet von Machine Learning. Beim Machine Learning geht es darum, Vorhersagen oder Entscheidungen zu treffen, indem ein Lösungsmodell eingesetzt wird, das automatisiert durch Beispieldaten generiert (trainiert) wurde. Im Gegensatz zu symbolischen Algorithmen muss das Verhalten des Lösungsmodells nicht explizit durch Menschen definiert werden. Machine Learning eignet sich daher für Probleme, für die es besonders schwierig ist, explizite Lösungsstrategien zu definieren ([20], S. 12). Das Ziel beim RL besteht darin, für eine Umgebung, in der sich aufeinanderfolgende Entscheidungen gegenseitig beeinflussen, ein Regelwerk zu generieren, das den möglichen Zuständen der Umgebung die erfolgsversprechendsten Entscheidungen zuordnet. Beim RL wird das Lösungsmodell trainiert, indem es mit der Umgebung interagiert, und die Rückmeldung der Umgebung verarbeitet, um sein Regelwerk zu verbessern. Durch RL zu lösende Probleme werden häufig durch MDPs modelliert ([28], S. 789 f.; [32], S. 1 f.). Reinforcement Learning ist nicht nur zur Lösung von Spielen verbreitet, sondern findet auch in Bereichen der Robotik Anwendung bis hin zur Personalisierung von Inhalten auf Webseiten ([28], S. 850; [32], S. 450).

3.3.1 Taxonomie

Es existieren viele verschiedene Arten von RL-Verfahren. Dieses Kapitel beleuchtet weit verbreitete Kategorien, deren Eigenschaften und wie gut sich Verfahren aus diesen Kategorien zur Lösung von Vier Gewinnt und zur Beantwortung der Fragestellung eignen.

Tabellenbasierte vs. approximierende Verfahren Manche RL-Verfahren verwenden Tabellen, um die Grundlage für das Regelwerk abzubilden, andere Verfahren approximieren diese Tabellen. Bei tabellenbasierten Verfahren wie Q-Learning oder SARSA wird jedem Paar aus Zuständen und Aktionen ein Wert zugeordnet, der beschreibt, wie gut es ist, im jeweiligen Zustand die jeweilige Aktion zu wählen. Diese Verfahren eigenen sich für relativ kleine Zustandsräume mit einer Größe von bis zu 10^6 Zuständen ([28], S. 803 ff.). Es wurde sogar gezeigt, dass bei genügend Training die Leistung von Q-Learning-Agenten zu perfekten Verhalten konvergiert ([32], S. 140). Vier Gewinnt hat allerdings eine wesentlich höhere Zustandskomplexität von 10^{14} [5]. Um für jedes Paar aus Zuständen und Aktionen auch nur einen Bit zu speichern, wären $\frac{7}{8}\,\text{Byte} \cdot 10^{14} = 87.5\,\text{Terabyte}$ Speicher erforderlich, und ein akkurates Modell zu trainieren würde zu viel Zeit in Anspruch nehmen ([28], S. 803, [32], S. 195). In solchen Fällen muss die Tabelle approximiert werden. Dazu haben sich tiefe neuronale Netzwerke (DNNs) als etablierte Lösung herausgestellt. Wenn bei RL DNNs zum Einsatz kommen, spricht man von Deep RL ([28], S. 809; [32], S. 236).

Modellbasierte vs. modellfreie Verfahren Bei RL wird zwischen modellbasierten und modellfreien Ansätzen unterschieden. Dabei bezieht sich der Begriff „Modell“ nicht auf das Lösungsmodell, das bei beiden Ansätzen trainiert wird, sondern auf ein Modell der Umgebung, dass bei beim Training und der Nutzung von modellbasierten Methoden eingesetzt wird, um Vorhersagen über die Auswirkungen von Entscheidungen zu treffen. Modellfreie Methoden hingegen kommen ohne ein solches Modell aus. Der Agent lernt alleine durch die Interaktion mit der Umgebung und die dadurch erhaltene Rückmeldung ([28], S. 790; [32], S. 7). Es ist anzumerken, dass alle in Kapitel 3.2 vorgestellten symbolischen Algorithmen ähnlich wie modellbasierte RL-Verfahren auf Modelle zurückgreifen, um Vorhersagen über das Verhalten der Umgebung zu treffen.

Daher sind modellfreie Methoden einfacher in der Implementierung und gut geeignet für Szenarien, die aufgrund ihrer Komplexität schwierig zu modellieren sind ([32], S. 12). Aufgrund der Fähigkeit, Vorhersagen über die Umgebung treffen zu können, weisen modellbasierte Methoden eine höhere Sample Complexity auf, was bedeutet, dass beim

Training weniger Versuche benötigt werden, um ein effektives Regelwerk zu erlernen. Das ist besonders vorteilhaft, wenn Versuche teuer sind und nicht es eine Herausforderung darstellt, genügend Daten zu erheben, so zum Beispiel beim Training in der realen Welt ([28], S. 687, S. 818, S. 959 f.).

Aufgrund des niedrigeren Implementierungsaufwands und des im Fall von Vier Gewinnt günstigen Trainings, richtet sich der Fokus der Arbeit auf modellfreie Methoden. Außerdem wurde in verschiedenen Untersuchungen modellfreie Methoden erfolgreich zur Implementierung von Agents für Vier Gewinnt eingesetzt [2], [34], [12], [37].

Wertbasierte vs. strategiebasierte Verfahren Modellfreie RL-Verfahren lassen sich in wertbasierte und strategiebasierte Varianten einteilen. Bei wertbasierten Verfahren wird eine Nutzenfunktion gelernt, die jedes Zustands-Aktionspaar bewertet. Bei der Anwendung eines trainierten wertbasierten Modells kommt eine Regelwerk zum Einsatz, das entsprechend der erlernten Nutzenfunktion für jeden Zustand immer die Aktion mit der besten Bewertung wählt ([28], S. 790).

Bei strategiebasierten Methoden wird das Regelwerk nicht aus einer Nutzenfunktion abgeleitet, sondern das Regelwerk wird direkt erlernt ([28], S. 790). Gegenüber wertbasierten Methoden hat das den Vorteil, dass dadurch auch Regelwerke modelliert erlernt werden können, die Entscheidungen basierend auf Wahrscheinlichkeiten treffen ([1], S. 195). Ein klassischer Anwendungsfall ist das Spiel Schere-Stein-Papier, bei dem das optimale Regelwerk darin besteht, alle Aktionen (Schere, Stein, Papier) zufällig mit derselben Wahrscheinlichkeit zu wählen. Ein solches wahrscheinlichkeitsbasiertes Regelwerk kann durch wertbasierte Methoden nicht abgebildet werden, da sie stets den einen laut Werte-Funktion vermeintlich besten Zustand wählen. Ein weiterer Vorteil von strategiebasierten Methoden ist, dass sie kontinuierlichen Aktionsräume abbilden können und wertbasierte Methoden nicht ([1], S. 196).

Im Fall von Vier Gewinnt sind beide Vorteile von strategiebasierten Verfahren nicht relevant, da Vier Gewinnt einen diskreten Aktionsraum besitzt, und zufälliges Handeln keinen strategischen Vorteil bringt.

Single-Agent vs. Multi-Agent Reinforcement Learning Vier Gewinnt kann als Problem des Gebiets Multi-Agent RL (MARL) betrachtet werden. MARL ist ein Teilgebiet des RL, in denen mehrere RL-Agenten in derselben Umgebung miteinander interagieren ([1], S. 2). Die Agenten können in der Umgebung ein kompetitives oder kooperatives Verhältnis oder eine Mischung beider Verhältnisse zueinander haben ([1], S. 9). In

Zwei-Spieler Nullsummenspielen wie Vier Gewinnt arbeiten die Agenten rein kompetitiv. Ein entscheidender Unterschied von kompetitiven MARL-Problemen zu Single-Agent-RL-Problemen (SARL) besteht darin, dass in SARL-Problemen die Umgebung eines trainierenden Agents statisch ist, was bedeutet, dass sich die Übergangsfunktion des zugrundeliegenden MDPs nicht ändert. Beim Training in einer Multi-Agent-Umgebung lernen mehrere Agenten gleichzeitig, damit ändert sich die Übergangsfunktion und die Umgebung ist nicht statisch. Die Agents müssen sich im Trainingsprozess an die sich ändernde Umgebung anpassen können ([1], S. 12).

Es gibt MARL-Methoden, die auf eine sich ändernde Umgebung optimiert sind. Dazu gehören Beispielsweise Methoden, die dem Konzept „Centralized Training Decentralized Execution“ (CTDE) zuzuordnen sind. CTDE bedeutet, dass Agenten während des Trainings aus den Erfahrungen voneinander lernen, aber ihre Entscheidungen trotzdem selbstständig treffen können ([1], S. 231). Da solche koordinierenden Ansätze zusätzliche Komplexität einführen, wird in dieser Arbeit der Fokus auf Independent Learning des Bereichs „Decentralized Training Decentralized Execution“ gerichtet. Beim Independent Learning interagieren die Agenten zwar im Training miteinander, erlernen ihr Regelwerk jedoch unabhängig voneinander. Bei Nullsummenspielen geschieht Independent Learning üblicherweise im Zusammenhang mit Self-Play, was bedeutet, dass alle Agenten das selbe Lernverfahren einsetzen. Im Training lernen die Agenten, gegenseitig ihre Schwächen auszunutzen und diese Schwächen zu beheben. Es ist aber auch möglich, Mixed-Play anzuwenden, also im Training Agenten mit verschiedenen Lernverfahren gegeneinander antreten zu lassen. Über Independent Learning lassen sich auch SARL-Methoden auf MARL-Probleme anwenden. Dabei ist zu berücksichtigen, dass SARL-Modelle in nichtstationären Umgebungen ein weniger stabiles Lernverhalten aufweisen als bei stationären Umgebungen, dennoch werden sie in der Praxis häufig erfolgreich für MARL-Probleme eingesetzt ([1], S. 221 f.).

Außerdem ist anzumerken, dass sich Off-Policy-Verfahren weniger für MARL eigenen als On-Policy-Verfahren, weil Off-Policy-Verfahren Entscheidungen basierend auf Erfahrungen treffen, die mehrere Lernvorgänge in der Vergangenheit liegen, in der der Gegenspieler noch eine inzwischen veraltete Strategie hatte. Agents mit On-Policy Algorithmen hingegen lernen nur von anhand des letzten Lernvorgangs und damit der aktuellsten Strategie der anderen Agenten. Das kann zu stabilerem Lernverhalten führen ([1], S. 224 f.).

3.3.2 Künstliche neuronale Netzwerke

Bei künstlichen neuronalen Netzwerken (KNN) handelt es sich um eine weit verbreitet Methode des Machine Learning zur Approximation von komplexen, nicht-linearen Funktionen ([1], S. 164 f.). Wie in Kapitel 3.3.1 erwähnt, werden neuronale Netzwerke im Zusammenhang mit Deep Reinforcement Learning eingesetzt, um eine Approximierung für die optimale Nutzenfunktion oder das optimale Regelwerk zu finden.

Aufbau Den Hauptbestandteil von künstlichen neuronalen Netzwerken bilden die Neuronen. Sie bestehen aus folgenden Komponenten:

- Eingabewerte x_1 bis x_n
- Gewichte für jeden Eingabewert w_1 bis w_n
- Bias b
- Nicht-lineare Aktivierungsfunktion g
- Ausgabewert, der berechnet wird, indem die Gewichtete Summe aus den Eingabewerten mit dem Bias addiert wird, und dann in der Aktivierungsfunktion verrechnet wird ([1], S. 166 f.).

In künstlichen neuronalen Netzwerken sind diese Neuronen schichtweise miteinander verbunden. In KNNs mit der Feedforward-Eigenschaft, auf die sich diese Arbeit beschränkt, nimmt jedes Neuron Ausgaben der Neuronen der vorangegangenen Schicht als Eingaben entgegen. Es gibt keine Rückkopplungen oder zyklischen Verbindungen.

Eine Ausnahme bilden die Neuronen der ersten Schicht, der sogenannten Eingabeschicht. Darin nimmt jedes Neuron als Eingabewert einen Parameter der zu approximierenden Funktion entgegen. Auf die Eingabeschicht folgen beliebig viele versteckte Schichten. Die Neuronen in den versteckten Schichten verarbeiten die Eingaben entsprechend nach Gewichten, Bias und Aktivierungsfunktion und leiten deren Ausgaben an die Neuronen in der nächsten Schicht weiter. Das passiert solange, bis die Ausgabeschicht erreicht wurde. Sie enthält so viele Neuronen, wie Ausgabewerte berechnet werden sollen ([1], S. 165 f.; [28], S. 751 f.).

Die Gewichte und Biase der Neuronen sind zunächst zufällig initialisiert und werden im Zuge des Trainings auf Grundlage von Beispieldaten optimiert, sodass das Netzwerk die Zielfunktion so gut wie möglich approximiert ([1], S. 169).

Der Aufbau eines KNNs lässt sich unter anderem über die Anzahl der versteckten Schichten, der darin enthaltenen Neuronen, der Art, wie sie miteinander verbunden sind, und den eingesetzten Aktivierungsfunktionen variieren ([28], S. 759).

Durch größere Netzwerke können komplexere Probleme gelöst werden, bei zu großen Netzwerken besteht jedoch die Gefahr des Overfitting, was bedeutet, dass das Netzwerk schlecht mit Eingaben umgehen kann, die es im Training nicht gesehen hat ([1], S. 166; [32], S. 225). Es wurde außerdem gezeigt, dass KNNs bei gleicher Anzahl von Gewichten und Biases bessere Ergebnisse erzielen, wenn sie tiefer statt breiter sind, also mehr haben anstatt mehr Neuronen pro Schicht haben ([28], S. 769).

Als Aktivierungsfunktion sind derzeit Rectified Linear Unit (ReLU) und Variationen davon verbreitet. ReLU gibt für Eingabewerte < 0 0 und ansonsten den Eingabewert zurück ([1], S. 167 f.; [28], S. 759).

Der optimale Aufbau eines KNNs hängt vom zu lösenden Problem ab. Es gibt Werkzeuge, die beim Finden eines guten Aufbaus unterstützen, dabei erfolgt dieser Prozess in der Praxis häufig auch durch Experimente und unter Zuhilfenahme von menschlicher Erfahrung und Intuition ([28], S. 759).

Training Während des Trainings werden die zunächst zufällig initialisierten Parameter θ (Gewichte und Biase der Neuronen) so optimiert, dass das KNN die Zielfunktion möglichst gut approximiert ([1], S. 169).

Dazu muss bestimmt werden können, wie gut das neuronale Netzwerke seine Aufgabe löst. Als Indikator dafür dient der Verlust. Sind die Ausgabewerte bekannt, die das KNN für bestimmte Eingaben liefern soll, so wie es im ML-Teilbereich Supervised Learning der Fall ist, kann der Verlust eines KNNs durch den Mean Squared Error, also die durchschnittliche quadrierte Differenz zwischen berechneten und tatsächlichen Werten angegeben werden ([1], S. 170). Wie Verlust bei RL-Verfahren berechnet wird, hängt vom konkret eingesetzten Verfahren ab ([32], S. 225).

Der Verlust kann als Funktion $L(\theta)$ betrachtet werden, die von den Parametern des KNNs abhängt. Das KNN löst seine Aufgabe dann gut, wenn das Minimum dieser Verlustfunktion erreicht wurde. Um das Minimum zu finden, wird der Gradient („multidimensionale Ableitung “) der Verlustfunktion $\Delta_\theta L(\theta)$ betrachtet, der die Steigung dieser Verlustfunktion beschreibt. Wird der Gradient für einen Satz von Parametern berechnet, kann daraus gefolgert werden, in welche Richtung und in welchem Verhältnis die Parameter zueinander verändert werden müssen, um den Verlust zu reduzieren ([1], S. 171).

Auf dieser Tatsache beruht das Gradientenverfahren. Nach dem Gradientenverfahren, werden die Parameter θ wiederholt wie folgt angepasst, bis ein Minimum erreicht wurde:

$$\theta \leftarrow \theta - \alpha * \Delta_\theta L(\theta)$$

α bezeichnet hierbei die Lernrate. Wird sie zu klein gewählt, erfolgt die Annäherung an das Minimum sehr langsam. Wenn sie zu groß gewählt wird, kann es passieren, dass erst gar kein Minimum gefunden wird ([14], Kapitel 10.3; [8], Kapitel 4). Daher existieren Verfahren, die die Lernrate dynamisch anpassen können, um den Optimierungsprozess zu beschleunigen ([1], S. 174). Bei der Lernrate handelt es sich um einen Hyperparameter. Hyperparameter beeinflussen den Lernprozess und müssen vor dem Training gezielt initialisiert werden. Ähnlich wie die Gestaltung der Architektur erfolgt die Initialisierung der Hyperparameter häufig händisch, wobei automatisierte Werkzeuge dabei unterstützen können [15] [30].

Um den Gradienten der Verlustfunktion für einen bestimmten Satz von Parametern zu berechnen, wird bei neuronalen Netzwerken das Verfahren Backpropagation angewendet. Auf Grundlage des Verlustes von jedem einzelnen im Training verfügbaren Datenpunkt wird rekursiv von der Output-Schicht bis hin zur Input-Schicht bestimmt, wie die Parameter im Verhältnis zueinander geändert werden müssen, um den Verlust zu reduzieren. Der Gradient entspricht dem Durchschnitt dieser Änderungen ([28], S. 766 f., [1], S. 174f.).

Das Standard-Gradientenverfahren berechnet den Verlust auf Grundlage aller beim Training verfügbaren Daten. Dies ist mit hohem Rechenaufwand verbunden. Aus diesem Grund existieren das stochastische Gradientenverfahren und das Mini-Batch-Gradientenverfahren, die den Verlust nicht basierend auf allen verfügbaren Daten, sondern nur auf Grundlage eines Datenpunktes bzw. einer Teilmenge der Daten berechnen ([1], S. 172).

Es ist anzumerken, dass über das Gradientenverfahren in den meisten Fällen nicht das globale Minimum der Verlustfunktion gefunden werden kann, sondern nur ein lokales Minimum. Dies reicht in den meisten Fällen jedoch aus, da die lokalen Minima von Verlustfunktionen gerade bei größeren KNNs größtenteils ähnlich niedrige Werte aufweisen, und die Wahrscheinlichkeit, ein lokales Minimum mit einem wesentlichen höheren Wert zu finden, sehr niedrig ist ([32], S. 200; [14], Kapitel 5.4.4; [10]).

3.3.3 Advantage Actor-Critic

Advanced Actor Critic (A2C) ist ein weit verbreitetes Verfahren, das sich nach den vorangegangenen Kapiteln zur Lösung von Vier Gewinnt eignet. Es ist ein Single-Agent-Verfahren, das modellfrei und off-policy arbeitet, und parametisierte Funktionen (wie z.B. KNNs) einsetzt, um das optimale Verhalten zu approximieren. Darüber hinaus handelt es sich um ein Actor-Critic-Verfahren, welche strategiebasierte Ansätze in der Actor-Komponente mit wertebasierten Ansätzen in der Critic-Komponente kombinieren ([1], S. 202 ff.).

Strategiebaierte Actor-Komponente Bei Actor-Critic-Verfahren kommen in der Actor-Komponente stets Policy-Gradient-Verfahren zum Einsatz. Policy Gradient Verfahren sind eine Unterart von strategiebasierten RL-Verfahren. Sie setzen voraus, dass das Regelwerk als parametrisierte Funktion abgebildet ist, so wie es beispielsweise bei KNNs der Fall ist. Denn dann gilt das Policy Gradient Theorem, das Aussagen über den Gradienten der Leistungsfähigkeit eines parametrisierten Regelwerks trifft ([32], S. 324; [1], S. 195)

Das Policy-Gradient-Verfahren, das die Grundlage des Actors in A2C bildet, ist das REINFORCE-Verfahren ([1], S. 203). Der Agent startet dabei an einem Startzustand der Trainingsumgebung und trifft dabei solange Entscheidungen anhand des Regelwerks π, bis ein Endzustand erreicht wurde. Damit ist eine Episode abgeschlossen. Dabei speichert er die Historie der in der Episode besuchten Zustände S, durchgeführten Aktionen A und erhaltenen Belohnungen R. Für jeden Schritt t in der Historie werden die Parameter ϕ des Regelwerks pi im Rahmen des Gradientenverfahrens nach folgendem Ausdruck angepasst:

$$\phi \leftarrow \phi + \alpha\gamma^t * G_t * \frac{\Delta\pi(A_t|S_t,\phi)}{\pi(A_t|S_t,\phi)}$$

Anders als in Abschnitt 3.3.2 wird hier allerdings nicht der Verlust reduziert, sondern es wird die Wahrscheinlichkeit erhöht, bei Zustand S_t die Aktion A_t auszuführen. Das geschieht proportional zur diskontierten Belohnung G_t, sodass die Wahrscheinlichkeit für Züge mit größerer Belohnung stärker erhöht wird. Der Gradient $\Delta\pi(A_t|S_t,\phi_t)$ zeigt dabei an, in welche Richtung und in welchem Verhältnis die Parameter zueinander verschoben werden müssen, um die Wahrscheinlichkeit, den Zug A_t bei Zustand S_t auszuführen, zu maximieren. Der Gradient wird durch die Wahrscheinlichkeit $\pi(A_t|S_t,\phi_t)$, den Zug auszuführen, dividiert, um dem Effekt entgegenzuwirken, dass Züge mit einer höheren

Wahrscheinlichkeit häufiger gewählt werden, und damit die Gewichte in Richtung der Züge mit höherer Wahrscheinlichkeit überproportional verschoben werden würden.

γ ist der Diskontierungsfaktor, für den gilt $0 <= \gamma <= 1$. Je kleiner, desto mehr beeinflussen frühere Züge die Parameter als spätere Züge.

Die diskontierten Belohnung G_t errechnet sich dabei aus einer gewichteten Summe der in der Episode erhaltenen Belohnungen:

$$G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} * R_k$$

Hier kommt erneut der Diskontierungsfaktor γ zum Einsatz. Je größer der Diskontierungsfaktor, desto stärker werden Belohnungen gewichtet, die weiter in der Zukunft liegen. Je kleiner, desto „kurzsichtiger" ist der Agent ([32], S. 55).

Die Wahrscheinlichkeit, im Zustand S_t den Zug A_t auszuführen, ergibt sich über das Regelwerk selbst. Der Gradient davon wird über den Backpropagation-Algorithmus ausgerechnet ([32], S. 326 f.).

Die Idee, dass die Erhöhung der Wahrscheinlichkeit der Züge, auch die Leistungsfähigkeit des Regelwerks erhöhen wird, ist aus dem Policy Gradient Theorem hergeleitet ([32], S. 326 f.).

REINFORCE weist häufig langsames und instabiles Training auf. Das liegt daran, dass die Parameter auf Grundlage von den Entscheidungen einer gesamten Episode angepasst werden, die einer Wahrscheinlichkeitsverteilung unterliegen und damit eine hohe Varianz aufweisen (MARL, S. 200). Um dieser hohen Varianz entgegenzuwirken, wird bei A2C ein wertebasierter Critic eingesetzt.

Wertebasierte Critic-Komponente Bei der Critic-Komponente handelt es sich um eine parametrisierte Funktion $V(s, \theta)$, die unter Berücksichtigung der Parameter θ Schätzungen über den Wert eines gegebenen Zustands s liefert (MARL, S. 202 ff.)

Bei A2C wird während des Trainings nach jeder durchgeführten Aktion ein Advantage-Wert berechnet. Dabei handelt es sich um die Differenz zwischen dem geschätzten Wert des Zustands, in dem sich der Agent befunden hat, bevor er die Aktion durchgeführt hat, und einem neuen Schätzwert, der sich aus der Summe der durch die Aktion erhaltene Belohnung und dem Schätzwert des über die Aktion erreichten neuen Zustands zusammensetzt:

$$Adv(s_t, a_t) = r_t + \gamma V(s_{t+1}, \theta) - V(s_t, \theta)$$

Ein positiver Advantage-Wert bedeutet, dass der durch die Aktion erreichte neue Zustand höherwertiger ist als durch die Wertefunktion angenommen. Ein negativer Advantage-Wert bedeutet, dass er weniger wert ist.

Dementsprechend werden die Parameter der Wertefunktion unter Verwendung des Gradientenverfahrens aktualisiert. Der Verlust wird hierbei als Quadrat des Advantage-Wertes definiert (MARL, S. 205):

$$L(\theta) = (r_t + \gamma V(s_{t+1}, \theta) - V(s_t, \theta))^2$$

Es ist anzumerken, dass hierbei Endzustände einer gesonderten Betrachtung bedürfen (vgl. MARL, S. 205). Aus Gründen der Übersichtlichkeit wird in dieser Arbeit darauf verzichtet.

Actor- und Critic-Komponenten im Zusammenspiel Das Zusammenspiel zwischen Actor- und Critic-Komponente gestaltet sich bei A2C so, dass die Parameter der auf REINFORCE basierten Actor-Komponente nicht am Ende einer Episode proportional zur in der Episode erhaltenen Belohnung aktualisiert werden, sondern stattdessen werden die Parameter bei jedem Schritt im Training proportional zum über die Critic-Komponente berechneten Advantage-Wert aktualisiert (MARL, S. 205):

$$\theta \leftarrow \theta + \alpha Adv(s_t, a_t) \frac{\Delta\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$

Dadurch, dass die Parameter der Funktionen in Abhängigkeit von den Erfahrungen aus einem Trainingsschritt und nicht einer gesamten Episode aktualisiert werden, weisen die Parameteraktualisierungen weniger Varianz auf. Kombiniert mit der dadurch häufigeren Anzahl an Parameteraktualisierungen führt dies in vielen Fällen zu effizienterem Training (MARL, S. 202).

Es existiert eine optimierte Variante von A2C, die sich Proximal Policy Optimization (PPO) nennt. Sie enthält unter anderem Mechanismen, die große Sprünge in den Veränderungen des Regelwerks verhindert, was zu einem noch effizienterem Trainingsverhalten führt (MARL, S. 206 ff.). Aufgrund der einfacheren Funktionsweise von A2C wird im Rahmen der Arbeit auf die Vorzüge von PPO verzichtet.

3.4 Robustheit

Der Begriff Robustheit wird durch das IEEE Standard Glossary of Software Engineering Terminology definiert als „Der Grad, zu dem ein System oder eine Komponente in der

Lage ist, unter fehlerhaften Eingaben oder belastenden Umgebungsbedingungen korrekt zu funktionieren“([21], S. 64).

In der Studie von Moos u.a. wird eine Übersicht zu verschiedenen Herangehensweisen zur Verbesserung der Robustheit von RL-Verfahren geschaffen. Der Begriff Robustheit wird insofern konkret auf RL übertragen, als dass es darum geht, wie gut RL-Verfahren funktionieren, wenn das Verhalten der Umgebung teilweise unbekannt ist. Das ist insbesondere relevant, weil RL-Modelle zunehmend in der realen physischen Welt angewendet werden, während sie weiterhin aus Kosten- oder Zeitgründen in simulierten Umgebungen trainiert werden. Bei der Anwendung dieser Modelle in der realen Welt kommt es dazu, dass deren Leistungsfähigkeit sinkt, da reale Einsatzszenarien Eigenschaften besitzen, die in der Simulation nicht vollständig abgebildet werden. So zum Beispiel in der Navigation basierend auf Kamerabildern, bei der zeitweise das Sichtfeld blockiert sein kann, oder bei der Kollaboration von Robotern und Menschen, bei der die Roboter in der Lage sein müssen, auf die vielschichtigen Absichten des Menschen reagieren zu können[24][25].

Das Ziel beim robusten Reinforcement Learning besteht darin, ein Regelwerk zu finden, das auch unter ungünstigen Bedingungen, möglichst gute Entscheidungen trifft. In der Studie wird jede untersuchte Herangehensweise einer von vier Kategorien zugeordnet, je nach dem in welcher Hinsicht dadurch erhöhte Robustheit erzielt werden soll. Die Kategorien betreffen jeweils unterschiedliche Aspekte des des modellierten MDPs.

Die erste Kategorie lautet Robustheit gegenüber Unsicherheit bezüglich Aktionen. Die hier zugeordneten Methoden zur Erhöhung der Robustheit zielen darauf ab, das Verhalten eines Agenten zu verbessern, wenn er in Szenarien eingesetzt wird, in denen er sich nicht darauf verlassen kann, dass die Aktionen, für die er sich entscheidet, auch tatsächlich durchgeführt werden. So ist es beispielsweise der Fall, wenn ein RL-Agent einen Roboterarm steuert, auf den plötzliche Stöße wirken (S. 301). Ein solches Szenario kann in Vier Gewinnt abgebildet werden, indem beispielsweise mit einer bestimmten Wahrscheinlichkeit nicht die Aktion durchgeführt wird, für die sich die Agenten entscheiden, sondern eine zufällige Aktion.

Bei der Kategorie Robustheit gegenüber Unsicherheit bezüglich Beobachtungen besteht das Ziel darin, das Verhalten eines Agenten zu optimieren, wenn sich der Agent nicht darauf verlassen kann, dass der beobachtete Zustand auch der tatsächliche Zustand ist. Das ist zum Beispiel der Fall, wenn Entscheidungen basierend auf Sensordaten getroffen werden, die fehlerhaft oder verrauscht sind. Auch das lässt sich in Vier Gewinnt abbilden, indem die Agenten nicht den tatsächlichen Zustand, sondern eine manipulierte Version als Entscheidungsgrundlage erhalten.

Bei den Herangehensweisen der Kategorien Robustheit gegenüber Störungen und Robustheit gegenüber Unsicherheiten bezüglich der Übergangsfunktion soll das Verhalten von Agenten unter leicht verschiedenen physikalischen Parametern wie Masse, Reibung, oder Gravitation verbessert und die Toleranz gegenüber Modellierungsfehler im Training erhöht werden. Es ist nicht möglich, diese Arten von Robustheit bei Vier Gewinnt zu untersuchen, da es sich bei dabei um ein theoretisches und kein physikalisches Problem handelt, und keine vergleichbaren Parameter variiert werden können.

Es ist anzumerken, dass die Kategorien nicht streng voneinander getrennt sind. Es liegt nahe, ein Szenario, bei dem ein Agent anstelle der durch ihn bestimmten Aktion eine andere Aktion durchführt, der Kategorie Robustheit gegenüber Unsicherheit bezüglich Aktionen zuzuordnen. Es kann aber auch der Kategorie Unsicherheit bezüglich der Übergangsfunktion zugeordnet werden, mit der Begründung, dass das Szenario auch so betrachtet werden kann, dass der Agent die gewählte Aktion durchführt, aber dadurch in einen anderen Zustand gerät als erwartet.

Ein Konzept, das beim robusten Reinforcement Learning häufig zum Einsatz kommt, besteht darin, das zu lösende Single-Agent-Problem als Multi-Agent-Problem zu betrachten, bei dem während des Trainings des RL-Agenten, dessen Ziel die Lösung des ursprünglichen Problems ist, auch einen Kontrahenten trainiert wird, der durch bestimmte Störfaktoren, z.B. die Änderung von physikalischen Parametern der Umgebung, die Leistungsfähigkeit des ersten Agenten senken soll. Dadurch arbeiten auf Robustheit optimierte Verfahren häufig pessimistischer als nicht auf Robustheit optimierte Verfahren, sodass es je nach konkretem Verfahren vorkommen kann, dass das Verfahren unter optimalen Bedingungen schlechtere Ergebnisse erzielt als das nicht auf Robustheit optimierte Verfahren. Wenn die Trainingsumgebung die Anwendungsumgebung gut abbildet, können nicht-robuste Verfahren bessere Ergebnisse erzielen, als welche, die auf Robustheit optimiert sind (S. 293).

Da sich die verschiedenen Arten der Unsicherheit auf Aspekte des MDPs beziehen, ergibt sich die Möglichkeit, in dieser Hinsicht nicht nur RL-Verfahren, sondern auch MDP-lösende symbolische Algorithmen auf Robustheit zu untersuchen.

4 Konzept

In diesem Kapitel wird erläutert, wie im Rahmen dieser Arbeit die Fragestellung der Arbeit beantwortet werden soll, inwiefern symbolische Algorithmen oder RL-Verfahren robuster sind.

Dies geschieht exemplarisch am Beispiel von Vier Gewinnt, da dieses Spiel eine kontrollierbare Umgebung darstellt, die reproduzierbare und vergleichbare Messungen ermöglicht. Außerdem wurde in verschiedenen Studien gezeigt, dass sich sowohl symbolische als auch RL-basierte Lösungsverfahren zur Lösung dieses Spiels eigenen. Als symbolischer Algorithmus wird hier MCTS eingesetzt, da dieses Verfahren unabhängig von problemspezifischen Heuristiken angewandt werden kann, und im Gegensatz zum Minimax-Algorithmus und Alpha-Beta-Pruning auch bei komplexeren Problemen mit einer akzeptablen Laufzeit betrieben werden kann. RL-Verfahren werden in dieser Arbeit durch PPO vertreten. Dabei handelt es sich um eine Optimierung des Verfahrens A2C. Genau wie A2C handelt es sich um ein approximierendes, modellfreies on-policy Verfahren, welche sich wie in Kapitel 3.3 herausgearbeitet, zur Lösung von Vier Gewinnt am besten eigenen, ohne dabei auf spezielle Verfahren für Multi-Agent-Probleme zurückzugreifen. Im Vergleich zu A2C erleichtert PPO durch seine geringere Sensibilität gegenüber Änderungen an Hyperparametern die Implementierung.

Die Robustheit der Verfahren wird bewertet, indem Agenten, die diese Verfahren implementieren, dazu veranlasst werden, das Spiel wiederholt gegen einen zufällig spielenden Agenten zu spielen. Dabei werden die Agenten unter bestimmten Szenarien beeinträchtigt, welche auf die in Kapitel **??** genannten Aspekte von Robustheit abzielen und es wird gemessen und verglichen, wie stark sich die Beeinträchtigungen auf die Gewinnrate auswirken. Es wäre auch denkbar, den MCTS-Agenten und den PPO-Agenten im Spiel gegeneinander zu untersuchen, während einer der beiden Agenten oder beide Agenten den Beeinträchtigungen ausgesetzt sind. Dies hat allerdings den Nachteil, dass im Anschluss nicht klar herausgearbeitet werden kann, wie sehr ein Agent durch ein Szenario beeinträchtigt wird oder seine geschwächtes Leistungsvermögen durch den Gegenspieler ausgenutzt wird. Um die Vergleichbarkeit und Unabhängigkeit der Ergebnisse zu gewährleisten, spielen die beiden Agenten daher gegen einen zufällig spielenden Agenten.

Der Verlust der Gewinnrate eines Agenten in einem Szenario mit Beeinträchtigung wird dabei relativ zur Gewinnrate im neutralen Szenario ohne Beeinträchtigungen betrachtet, um einen Ausgleich dafür zu schaffen, dass die Agenten im neutralen Szenario nicht dieselbe Gewinnrate gegen den zufällig spielenden Agenten erzielen und die Gewinnrate eines Agenten, der im neutralen Szenario eine niedrigere Gewinnrate erzielt, auch weniger weit fallen kann. Die Betrachtung erfolgt außerdem relativ zur Gewinnrate von 50 %, da ein Agent ab dem Punkt, ab dem er eine Gewinnrate von nur 50 % gegen einen zufällig spielenden Agenten erzielt, keinen strategischen Vorteil mehr besitzt ist.

Daraus ergibt sich folgende Formel zur Bewertung der Robustheit:

$$l = (r_n - r_s)/(r_n - 0.5)$$

Dabei bezeichnet r_n die Gewinnrate des untersuchten Agenten im Spiel gegen einen zufällig spielenden Agenten im neutralen Szenario und r_s die Gewinnrate des untersuchten Agenten im betrachteten Szenario mit Beeinträchtigung. Erzielt ein untersuchter Agent im neutralen und betrachteten Szenario dieselbe Gewinnrate, wird l 0. Erzielt ein untersuchter Agent im betrachteten Szenario eine Gewinnrate von 50 %, so ist l 1. Dazwischen verhält sich l linear zu r_s. Somit ist ein untersuchtes Verfahren im betrachteten Szenario robuster, je kleiner der Wert von l ist. Je größer r_n ist, desto präziser lässt sich l berechnen.

Folgende Szenarien werden betrachtet:

- Unsicherheit bezüglich Aktionen: Die Aktionen der Agenten bestehen darin, dass sie den nächsten Spielstein in eine freie Spalte des Spielfelds hineinwerfen, die sie auf Grundlage des beobachteten Spielfeldzustands auswählen. Dieses Szenario führt Unsicherheit bezüglich Aktionen ein, indem für einen Agenten der Spielstein mit einer bestimmten Wahrscheinlichkeit nicht in die ausgewählte Spalte, sondern in eine zufällige freie Spalte fällt.

- Unsicherheit bezüglich Beobachtungen: Die Agenten wählen zwischen möglichen Aktionen basierend auf deren Beobachtungen des Spielfelds. In diesem Szenario erhalten die Agenten fehlerhafte Informationen über das Spielfeld. Jedes Feld besitzt dabei eine bestimmte Wahrscheinlichkeit mit der nicht dessen tatsächlicher Zustand (leer, besetzt durch Spieler 1, besetzt durch Spieler 2) erkannt wird, sondern ein zufälliger Zustand.

  Es ist anzumerken, dass hierbei aus dem MDP ein Partially Observable MDP (POMDP) wird, also ein MDP dessen Umgebung nur teilweise oder fehlerhaft beobachtbar ist. Der betroffene Agent kann nicht mit Sicherheit gesagt werden, in welchem Zustand er sich gerade befindet. Zusätzlich zum MDP enthält das POMDP ein Observation Modell O(s, o), das die Wahrscheinlichkeit beschreibt, eine Beobachtung o im Zustand s zu machen. POMDPs sind komplizierter gezielt zu lösen, in der realen Welt jedoch wesentlich häufiger anzutreffen. Es gibt Optimierungen von MCTS und bestimmte RL-Methoden, über die POMDPs gezielt gelöst werden können, zum Beispiel indem für eine Entscheidung nicht nur der aktuelle Zustand, sondern die Historie der Zustände betrachtet wird ([28], S. 588 ff.).

Da es in dieser Arbeit darum geht, zu untersuchen, inwiefern grundsätzliche Eigenschaften von symbolischen Algorithmen und Reinforcement Learning Robustheit beeinflussen, werden diese gezielten Lösungen zur Vereinfachung nicht betrachtet. Beide Agenten gehen davon aus, dass es sich bei dem fehlerhaften Bild um den tatsächlichen Zustand des Spiels handelt, auch wenn der beobachtete Zustand gemäß der Spielregeln nicht erreicht werden könnte.

- Unsicherheit bezüglich Dynamik der Umgebung: Die Agenten erwarten ein bestimmtes Verhalten von der Umgebung, das durch die Spielregeln abgebildet wird. Der MCTS-Agent führt Simulationen anhand dieser Erwartungen aus und der RL-Agent wird auf Grundlage dieser Erwartungen trainiert. In diesem Szenario werden die Erwartungen an das Verhalten der Umgebung gebrochen, indem ein bestimmter Spieler mit einer bestimmten Wahrscheinlichkeit zwei Züge hintereinander durchführt.

Da bei Vier Gewinnt, wie in Kapitel 3.1 erwähnt, der Spieler, der den ersten Stein platziert, einen Vorteil hat, wechselt bei den Messungen, sofern nicht anders gekennzeichnet, zu Beginn jedes Spiels das Recht, den ersten Zug zu durchzuführen.

Es ist zu beachten, dass die Agenten auf einem ähnlichen und gewissermaßen starkem Level spielen. Denn bei Agenten, die ohnehin nicht stark spielen, wird es schwierig sein, in den verschiedenen Szenarien zur Untersuchung der Robustheit eine aussagekräftige Änderung in den Gewinnraten zu messen. Außerdem ist anzunehmen, dass ein stärkerer Agent Entscheidungen trifft, die längerfristig zum Erfolg führen, sodass er auch die Beeinträchtigungen in den verschiedenen Szenarien besser wegstecken kann als ein schwächerer Agent. Als Indikator für die Spielstärke der Agenten wird die durchschnittliche Gewinnrate und Spieldauer im Spiel gegen zufällig spielende Agenten bei abwechselndem Anzugsrecht und im Spiel gegen sich selbst bei konstantem Anzugsrecht verwendet. Im Spiel gegen zufällig spielende Agenten sollten starke Agenten hohe Gewinnraten bei kurzer Spieldauer erzielen. Im Spiel gegen sich selbst sollte der erste Spieler aufgrund des Vorteils des Anzugsrechts eine hohe Gewinnrate erzielen und auch die Spieldauer wird sich verlängern, da der Gegenspieler auch besser verteidigt, sodass komplexere Spielsituationen entstehen und sich die Entscheidung des Spiels hinauszögert. Zur Beurteilung der Spielstärke der Agenten erfolgt außerdem eine kurze qualitative Analyse im Spiel gegen einen Menschen.

5 Realisierung

5.1 Messumgebung

Als Grundlage für die Messumgebung dient die Open Source Python-Bibliothek PettingZoo, die zum Entwickeln und Testen von MARL-Systemen konzipiert wurde. Sie stellt eine einheitliche Schnittstelle zu Umgebungen bereit, in der Agenten miteinander interagieren können.

Die Umgebungen definieren den Rahmen, in dem die Agenten miteinander interagieren. Sie weisen ein bestimmtes Verhalten auf und definieren unter anderem, unter welchen Bedingungen welche Aktionen möglich sind, Belohnungen verteilt werden und in welcher Form die Agenten Informationen über den Zustand der Umgebung erhalten. Es existieren eine Reihe von vorgefertigten Umgebungen, darunter welche, die kooperative Probleme zum Benchmarking von MARL-Systemen oder auch rundenbasierte Spiele wie Vier Gewinnt abbilden.

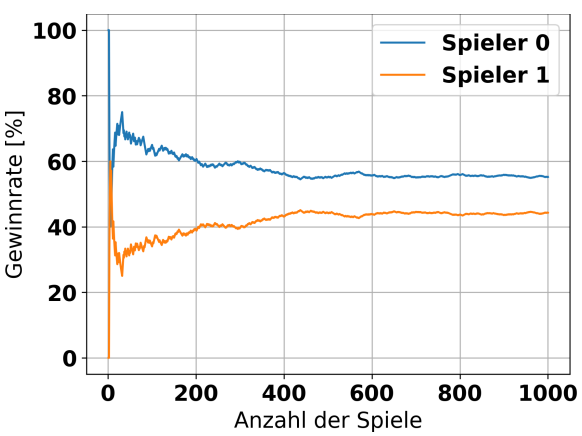
Die durch PettingZoo bereitgestellte Schnittstelle ermöglicht es, aus Sicht eines Agenten den aktuellen Zustand der Umgebung zu beobachten, die im aktuellen Zustand möglichen Aktionen und erhaltenen Belohnungen zu ermitteln und eine Aktion auszuwählen, die in der Umgebung durchgeführt werden soll.

Für alle Agenten der Umgebung kann benutzerdefinierte Logik eingebunden werden, die bestimmt, wie sie ihre Aktionen wählen. Vorgesehen sind dabei RL-Modelle, es können jedoch auch symbolische Algorithmen eingesetzt werden, darunter auch welche, die ihre Entscheidungen rein zufällig oder unter Einbezug von menschlichen Eingaben treffen [16].

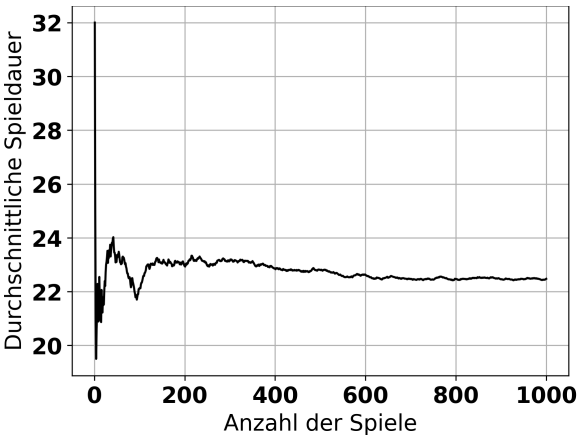
Die Implementierung der Messumgebung baut auf der offiziellen Implementierung der Vier-Gewinnt-Umgebung von PettingZoo auf. Die Messumgebung tut dabei nichts anderes, als wiederholt zwei Agenten mit bestimmten Lösungsansätzen das Spiel spielen zu lassen und dabei die Gewinnraten und Spieldauer zu messen. Die Open-Source-Eigenschaft ermöglicht es, den Quellcode zu modifizieren. Davon wird im weiteren Verlauf der Arbeit Gebrauch gemacht, unter anderem, um die verschiedenen Szenarien zur Untersuchung von Robustheit abzubilden.

Im Zuge der Realisierung der Messumgebung ist aufgefallen, dass wenn zwei Agenten (Spieler 0 und Spieler 1) alle Aktionen im Spiel mit derselben Wahrscheinlichkeit rein zufällig wählen, nach 1000 Spielen Spieler 0 mit 55,20 % gegenüber Spieler 1 mit 44,30 % eine wesentlich höhere Gewinnrate erzielt.

Das lässt sich dadurch erklären, dass die Vier-Gewinnt-Umgebung so implementiert



(a) Gewinnrate.



(b) Durchschnittliche Spieldauer.

Abb. 1: Gewinnrate und durchschnittliche Spieldauer bei konstanter Spielerreihenfolge.

ist, dass Spieler 0 immer der Spieler ist, der den ersten Stein setzen darf. Er ist damit seinem Gegenspieler immer einen Spielzug voraus, was die Wahrscheinlichkeit erhöht, als erstes Vier Steine in eine Reihe zu bekommen. An dieser Stelle sei nochmals zu erwähnen, dass bei Vier Gewinnt der erste Spieler bei optimaler Spielweise stets gewinnen kann. Um ausgeglichene Messungen zu gewährleisten, muss daher sichergestellt werden, dass sich im Rahmen der Messungen die beiden Spieler mit dem ersten Zug abwechseln.

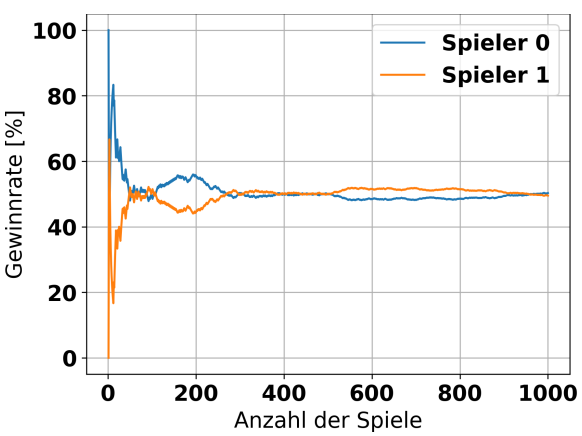
Die Vier-Gewinnt-Umgebung von PettingZoo wurde daher erweitert, um einen Parameter entgegenzunehmen und zu verarbeiten, der bestimmt, welcher Spieler anfangen soll. Die Messumgebung wechselt den Wert des Parameters nach jedem Spiel durch. Nach dieser Änderung weisen die Spiele wesentlich ausgeglichenere Ergebnisse auf. Spieler 0 gewinnt 50,30 % und Spieler 1 49,50 % der Spiele. Die durchschnittliche Spieldauer bleibt dabei mit 22,48 Zügen vor der Änderung gegenüber 22,26 Zügen nach der Änderung nahezu unverändert.

5.2 MCTS-Agent

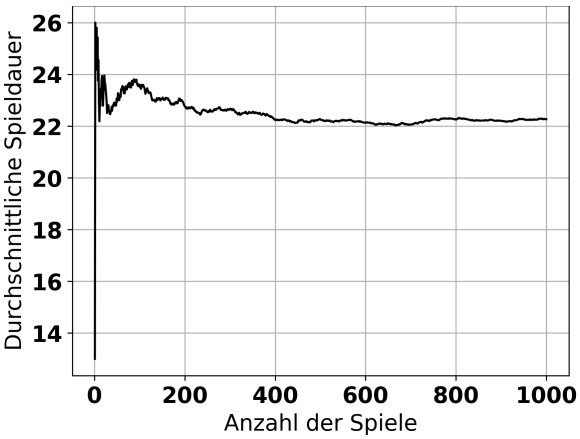
Bei der Implementierung des MCTS-Agenten, diente „Deep Learning and the Game of Go“ ([14], Kapitel 4.5) als Orientierung. Wie im genannten Werk besteht die im Rahmen dieser Arbeit entstandenen Implementierung aus zwei Klassen. Einer Klasse `MctsNode`, die einen Knoten im MCTS-Baum abbildet, und einer weiteren Klasse `MctsAgent`, die den Agenten repräsentiert und die wesentliche Logik des Algorithmus beinhaltet.

Die Klasse `MctsNode` besitzt dabei unter anderem folgende Attribute:

- `parent: MctsNode` und `children: list[MctsNode]`: Sie verwalten Beziehungen



(a) Gewinnrate.



(b) Durchschnittliche Spieldauer.

Abb. 2: Gewinnrate und durchschnittliche Spieldauer bei abwechselnder Spielerreihenfolge.

zu anderen Instanzen der Klasse `MctsNode`, sodass sie zusammen den MCTS-Baum abbilden.

- `visitation_count: int` und `total_reward: int`: Diese Werte werden zur Berechnung der UCT-Werte in der Selection-Phase benötigt und in der Backpropagation-Phase aktualisiert.

- `state: numpy.ndarray(6, 7, 2)`: Dabei handelt es sich um den Spielfeldzustand, den der Knoten repräsentiert. Das Format ist dabei dasselbe wie das, mit dem PettingZoo Beobachtungen über das Spielfeld zur Verfügung stellt. Dieses Attribut dient als Ausgangspunkt für die zufälligen Simulationen.

Die zentrale Methode der Klasse `MctsAgent` ist die Methode `determine_action(self, state: numpy.ndarray(6, 7, 2)) -> int`. Sie nimmt den Zustand des Spielfelds entgegen, führt den MCTS-Algorithmus durch und gibt eine Zahl zurück, die die Aktion widerspiegelt, die auf Grundlage des Algorithmus gewählt werden soll. Zunächst wird darin ein Objekt der Klasse `MctsNode` initialisiert, dessen `state`-Attribut der beobachtete Zustand `state` zugewiesen wird. Dieses Objekt stellt den Wurzelknoten des MCTS-Baums dar. Anschließend werden n mal folgende Methoden wiederholt, wobei n die Anzahl der pro Entscheidung durchzuführenden Simulationen ist, die über den Konstruktor der Klasse konfiguriert werden kann:

- `select(root_node: MctsNode) -> MctsNode`: Vom zuvor definierten Wurzelknoten werden per UCT-Formel solange Kinder ausgewählt, bis ein Knoten erreicht

wurde, der ein Endzustand ist oder nicht vollständig expandiert ist, also weniger Kindknoten als Aktionen hat, die von dem dem Knoten entsprechenden Zustand möglich sind. Die UCT-Konstante kann dabei über den Konstruktor konfiguriert werden. Wenn es sich bei dem erreichten Knoten um einen Endzustand handelt, wird dieser Knoten zurückgegeben. Ansonsten, wird von diesem Knoten ein zufälliger legaler Spielzug ausgeführt und ein neuer Knoten, der den dadurch erreichten Zustand abbildet, wird als Kindknoten hinzugefügt. Dies entspricht dem Expansion-Schritt des MCTS-Algorithmus. Zurückgegeben wird dann der neu hinzugefügte Knoten.

- `simulate(selected_node: MctsNode) -> int`: Diese Methode nimmt den in der `select`-Methode ausgewählten Knoten entgegen, das Spiel wird ab dem Zustand, den der Knoten repräsentiert, zu Ende gespielt, und das Ergebnis des Spiels wird zurückgegeben.

- `backpropagate(selected_node: MctsNode, result: int) -> None`: Von ausgewählten Knoten wird das Attribut `visitation_count` erhöht und das Ergebnis der Simulation zu `total_rewards` addiert. Dieser Vorgang wird jeweils für alle Elternknoten durchgeführt, bis der Wurzelknoten erreicht wurde.

Nach n Wiederholungen wird eine Zahl zurückgegeben, die die Aktion repräsentiert, die vom Wurzelknoten zum direkten Kindknoten führt, dessen `visitation_count`-Attribut den höchsten Wert hat.

Für den Algorithmus wird ein Abbild für die Dynamik des Spiels benötigt, das unter anderem die Spielregeln, Gewinnbedingungen oder mögliche Aktionen in Abhängigkeit des aktuellen Zustands enthält. Dafür kommt in dieser Implementierung die Vier-Gewinnt-Umgebung von PettingZoo zum Einsatz. Diese enthält ein genau solches Abbild, und dadurch wird Aufwand in der Implementierung gespart. Es hat sich herausgestellt, dass an der Umgebung zwei Modifikationen notwendig sind, da die PettingZoo-Umgebungen in erster Linie zum Training von RL-Agenten konzipiert sind, und nicht um als Modell für symbolische Algorithmen oder modellbasierte RL-Verfahren zu dienen.

Zunächst wird keine Funktionalität unterstützt, um eine PettingZoo-Umgebung mit einem bestimmten Zustand zu initialisieren, was jedoch in der Klasse `MctsAgent` beispielsweise vor der Durchführung von Simulationen notwendig ist. Die Methode `reset(self, seed: int, options: dict) -> None` der Vier-Gewinnt-Umgebung wurde daher erweitert, um im `options`-Parameter nach dem Schlüssel ‘‘`state`’’ zu suchen, unter dem der gewünschte Zustand abgelegt werden kann, und ggf. entsprechend verarbeitet wird.

Der Zustand wird dabei in derselben Form erwartet, wie PettingZoo seine Beobachtungen liefert.

Eine weitere Herausforderung bestand darin, dass PettingZoo-Umgebungen Beobachtungen stets perspektivisch aus Sicht des Agenten liefern, der aktuell am Zug ist. Im Fall von Vier Gewinnt bestehen die Beobachtungen aus einem Array, das das Spielfeld repräsentiert. Dieses Array enthält sechs weitere Arrays, die jeweils eine Reihe des Spielfelds abbilden, wobei jedes dieser Arrays sieben Felder enthält, die einem Feld in der jeweiligen Reihe entsprechen. Jedes dieser Felder ist ein Array bestehend aus zwei Elementen, die jeweils die Werte 0 und 1 annehmen können. Wenn das erste Element 1 ist, bedeutet das, dass der Spieler, der aktuell am Zug ist, einen Stein an der entsprechenden Position platziert hat. Wenn das zweite Feld 1 ist, bedeutet das, dass der Gegenspieler einen Stein platziert hat. 0 bedeutet, dass der entsprechende Spieler an der Position keinen Stein platziert hat [16]. Diese perspektivischen Beobachtungen erleichtern die Implementierung von RL-Agenten. Da die Vier-Gewinnt-Umgebung von PettingZoo jedoch nicht mit perspektivischen Beobachtungen, sondern mit einem globalen Zustand arbeitet, mussten Mechanismen implementiert werden, um diesen aus den perspektivischen Beobachtungen zu erzeugen. Dazu speichert die Klasse MctsNode, welcher Spieler als Nächstes am Zug ist, und die reset-Methode der Umgebung berücksichtigt einen entsprechenden Schlüssel im options-Parameter.

Aufgrund der zeitlichen Beschränkung dieser Arbeit wird auf Experimente bezüglich Optimierungen verzichtet. Daher wird als Auswahlstrategie in der Selection-Phase nach den Empfehlungen UCT mit $c = \sqrt{2}$ eingesetzt. Was die Expansion-Phase betrifft, wird sich ebenfalls an den Standard gehalten und nur einen und nicht mehrere Knoten hinzugefügt. In der Simulation-Phase werden Light-Playouts und keine Heavy-Playouts eingesetzt, da Light-Playouts ohne Wissen über das konkret zu lösende Problem auskommen, sodass die Ergebnisse dieser Arbeit so weit möglich auf verschiedene Probleme angewandt werden können.

5.2.1 Zeitlicher Aufwand von Entscheidungen des MCTS-Agenten

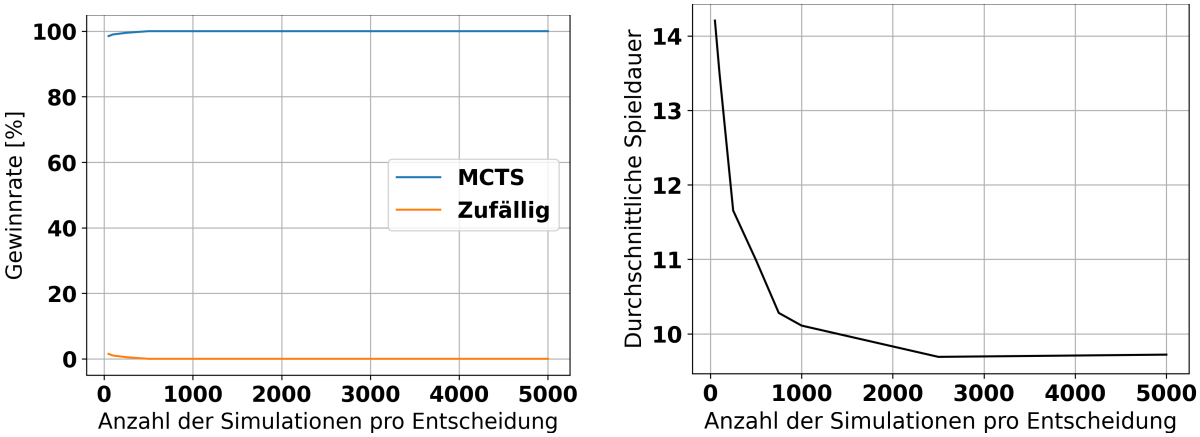
Die in dieser Arbeit durchgeführten Messungen fanden auf einem Computer mit einer Intel Core i7 8650U CPU statt. Ein MCTS-Agent, der pro Entscheidung 5.000 Simulationen durchführt, benötigt dabei für jeden Zug etwa eine zehn Sekunden. In einem Spiel mit einer Länge von 20 Zügen rechnet der MCTS-Agent also 200 Sekunden. Für 200 Spiele, die im Rahmen dieser zeitlich begrenzten Arbeit an vielen Stellen als ausreichend für aussagekräftige Messungen betrachtet werden, werden für jeden beteiligten

MCTS-Agenten mehr als elf Stunden benötigt. Die Rechenzeit verhält sich proportional zur Anzahl der durchgeführten Simulationen.

Es liegt nahe, den MCTS-Agenten durch Parallelisierung zu beschleunigen. Dadurch lässt sich die Rechenzeit proportional (ggf. sogar überproportional) zur auf der Maschine verfügbaren CPU-Ressourcen verkürzen, ohne dass die Ergebnisse dadurch beeinträchtigt werden (vgl. [9]). Dies ist vor allem bei Echtzeitanwendungen sinnvoll. Ein solcher Anspruch wird in dieser Arbeit jedoch nicht gestellt. Um die Komplexität des Algorithmus möglichst niedrig zu halten, wird auf die Parallelisierung von MCTS verzichtet. Stattdessen werden die Messungen parallelisiert und die Ergebnisse zusammengeführt.

5.2.2 Quantitative Untersuchung

Um einen Eindruck davon zu bekommen, wie sich die Spielstärke des implementierten MCTS-Agenten in Abhängigkeit von der Anzahl der für jede Entscheidung durchgeführten Simulationen zu bekommen, wurde eine qualitative Analyse des MCTS-Agenten durchgeführt, in der 200 Spiele gegen einen zufällig spielenden Agenten mit abwechselndem Anzugsrecht durchgeführt wurden und 200 Spiele gegen sich selbst mit konstantem Anzugsrecht. Diese Messungen wurden mit 50, 100, 250, 500, 750, 1000, 2500 und 5000 Simulationen pro Entscheidung durchgeführt.



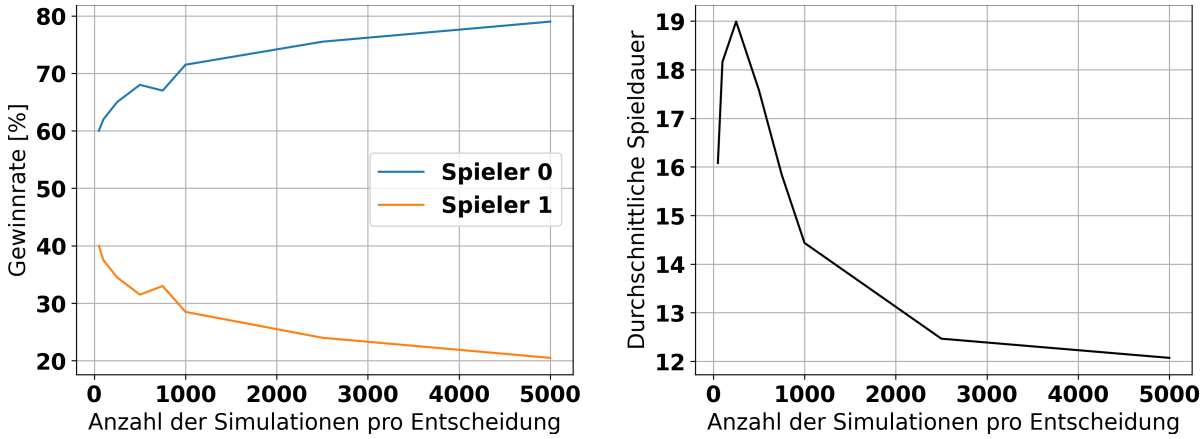
(a) Gewinnrate.

(b) Durchschnittliche Spieldauer.

Abb. 3: Gewinnrate und durchschnittliche Spieldauer in Abhängigkeit von der Anzahl der Simulationen pro Entscheidung beim Spiel eines MCTS-Agenten gegen einen zufällig spielenden Agenten.

Die Gewinnrate, die der MCTS-Agent gegen den zufällig spielenden Agenten erzielt, beträgt bei 50 Simulationen pro Entscheidung bereits 98,5 % und steigt bis 1000 Simulationen auf 100 %, welche der Agent auch bei 2500 und 5000 Simulationen hält. Die

mittlere Spieldauer startet bei 50 Simulationen mit 14.21 Zügen und flacht bei 2500 Simulationen mit 9,69 ab. Dass die Spieldauer bei 5000 Simulationen auf 9,72 steigt, dabei handelt es sich vermutlich um eine stochastisch bedingte Messungenauigkeit. Die kürzeste mögliche durchschnittliche Spieldauer beträgt dabei 7.5, denn damit der anziehende Spieler gewinnt kann, müssen mindestens sieben Steine platziert sein, bzw. acht Steine, damit der nachziehende Spieler gewinnen kann. Sie ist auch bei perfekter Spielweise des MCTS-Agenten schwer zu erreichen, da ein Spiel länger dauert, sobald sein Gegenspieler eine durch den MCTS-Agenten gebildete Kette blockiert. Aus den Messungen geht hervor, dass der MCTS-Agent einem zufällig spielenden Agenten bereits bei 50 Simulationen pro Entscheidung weit überlegen ist. Auch wenn die Gewinnrate ab 1000 Simulationen 100 % beträgt, kann über die kürzer werdende Spieldauer eine Steigerung der Leistung beobachtet werden. Der MCTS-Agent entscheidet mit steigender Anzahl von Simulationen pro Entscheidung das Spiel schneller für sich.



(a) Gewinnrate.

(b) Durchschnittliche Spieldauer.

Abb. 4: Gewinnrate und durchschnittliche Spieldauer in Abhängigkeit von der Anzahl der Simulationen pro Entscheidung beim Spiel von zwei MCTS-Agenten gegeneinander.

Im Spiel gegen sich selbst erzielt der MCTS-Agent mit Anzugsrecht bei 50 Simulationen eine Gewinnrate von 60,0 %. Sie steigt von dort auf 79,0 % bei 5000 Simulationen pro Entscheidung. Bei 750 Simulationen sinkt die Gewinnrate auf 67,0 % von 68,0 % bei 500 Simulationen. Diese Beobachtung ist vermutlich stochastisch bedingt. Die durchschnittliche Spieldauer steigt zunächst rapide von 16,08 bei 50 Simulationen auf 18,99 Spieldauer bei 250 Simulationen und sinkt von dort auf 12,07 bei 5000 Simulationen. Aus der steigenden Gewinnrate und sinkenden Spieldauer geht hervor, dass der MCTS-Agent den Vorteil, das Anzugsrecht zu haben, unso konsequenter ausnutzen kann, je mehr Simulationen pro Entscheidung durchgeführt werden. Im Vergleich zu den Messungen aus

Kapitel 5.1, in denen zwei zufällig entscheidende Agenten mit konstantem Anzugsrecht spielen, wird deutlich, dass dies bereits bei 50 Simulationen der Fall ist. Da der erste Spieler das Spiel theoretisch immer gewinnen kann, sollte die Gewinnrate mit steigender Simulationsanzahl zu 100 % konvergieren.

5.2.3 Qualitative Untersuchung

Neben der quantitativen Analyse gegen zufällig spielende Agenten und gegen sich selbst wurde auch eine kurze qualitative Analyse im Spiel gegen einen menschlichen Spieler durchgeführt, um einen Eindruck über das strategische Spielverhalten des MCTS-Agenten zu gewinnen, das aus den quantitativen Analysen nicht hervorgeht. Der menschliche Spieler ist bei den Untersuchungen stets der Spieler, der den ersten Stein setzen darf. Er kann damit theoretisch jedes Spiel gewinnen und hat so mehr Kontrolle über den Spielverlauf, was die Analyse vereinfacht.

Ist der MCTS-Agent konfiguriert, um 500 Simulationen pro Entscheidung durchzuführen, lässt sich bereits beobachten, dass, wenn er bereits drei Steine in einer Kette platziert hat, stets den vierten Stein platziert, um zu gewinnen, sofern dies möglich ist. Wenn hingegen sein Gegenspieler drei Steine in einer Reihe hat, versäumt der Agent häufig, den vierten Stein zu blockieren, sodass der Gegenspieler das Spiel für sich entscheiden kann. Wird der MCTS-Agent nicht unter Druck gesetzt, wirken seine Züge häufig ziellos. Wenn er Angriffe vorbereitet, sind seine Züge leicht durchschaubar, was es als Mensch recht einfach macht, sie zu verteidigen. Es ist auch ohne strategische Kenntnisse des Spiels leicht möglich, gegen den MCTS-Agenten zu gewinnen. Ein Vorgehen das dabei häufig funktioniert, besteht darin, darauf abzuzielen, einen Stein in der vierten Reihe der mittleren Spalte platziert zu bekommen, und von dort aus eine diagonale Kette nach links unten oder rechts unten zu bilden. Manchmal geht dieser Plan nicht auf, jedoch entwickeln sich dadurch im Laufe des Spiels schnell andere offensichtliche Gewinnchancen.

Bei 2.500 Simulationen pro Entscheidung platziert der MCTS-Agent seinen ersten Stein meistens in der mittleren Spalte, sofern er frei ist, was laut Allis auch der stärkste Anfangszug ist [4]. Hat der Gegenspieler drei Steine in einer Kette positioniert, so wird diese Angriffsposition meistens durch den MCTS-Agenten verteidigt. Es macht sich auch bemerkbar, dass der Agent wesentlich aggressiver spielt. Platziert der Gegenspieler beispielsweise seine Steine zu Beginn des Spiels so, dass er keine Gefahr darstellt und die Bildung von Viererketten erst zum letztmöglichen Zeitpunkt verteidigt, baut der Agent Druck auf, sodass der Gegenspieler im restlichen Verlauf des Spiels häufig

dazu gezwungen ist, einen bestimmten Zug zu wählen, um nicht im nächsten Zug zu verlieren. Verteilt der Gegenspieler seine ersten drei Steine auf die beiden gegenüberliegenden Ränder, erzeugt der Agent häufig eine Zwickmühle, in dem er seine ersten drei Steine jeweils in die unterste Reihe der drei mittleren Spalten platziert, sodass er im nächsten Zug gewinnen kann, indem er seinen Stein in der zweiten oder vorletzten Spalte platziert. Dieses Verhalten konnte bei 500 Simulationen nicht beobachtet werden. Auch hier gelingt die oben genannte Strategie, sich als Ziel zu setzen, eine Diagonale von einer unteren Ecke zur Position in der mittleren Spalte in der vierten Reihe zu bilden. Allerdings scheint der MCTS-Agent besser voraus zu planen, denn die Spiele dauern länger und sein Gegenspieler muss häufiger verteidigen.

Bei 5.000 Simulationen pro Zug setzt der MCTS-Agenten seinen ersten Stein, wenn möglich, stets in die mittlere Spalte. Es ist eine verstärkte Bildung von Zwickmühlen zu bemerken, sofern sie durch den Gegenspieler nicht frühzeitig erkannt und verhindert werden. Durch seine noch aggressivere Spielweise, ist es nur unter besonderer Anstrengung möglich, als menschlicher Spieler ohne Kenntnisse über die optimale Spielweise gegen den MCTS-Agenten zu gewinnen.

Meistens geschieht dies durch die Ausnutzung einer Schwachstelle, die der MCTS-Agent unabhängig von der Anzahl der Simulationen pro Entscheidung (auch bei 10.000 Simulationen) aufzuweisen scheint: Hat der Gegenspieler drei Steine in einer Kette platziert und es droht, dass er im nächsten Zug den vierten Stein platziert und damit das Spiel gewinnt, so blockiert der MCTS-Agent die Platzierung nicht in jedem Fall. Besonders deutlich wird dies, wenn der Gegenspieler drei Steine in einer Spalte aufeinander liegen hat, während der MCTS-Agent zwei Steine in einer anderen Spalte aufeinander liegen hat, und das Spielfeld ansonsten leer ist. Anstatt den Gegenspieler zu blockieren, setzt er häufig den dritten Stein in seine Spalte.

Um diese Schwachstelle weiter zu untersuchen, wurde der MCTS-Agent konfiguriert, um 10.000 Simulationen pro Entscheidung durchzuführen und es wurde mit verschiedenen Werten für c_{UCT} gemessen, wie oft sich der MCTS-Agent in dieser Situation dafür entscheidet, den unmittelbaren Gewinn des Gegenspielers zu verhindern und wie oft er es bevorzugt, stattdessen den dritten Stein in seine Spalte zu platzieren und damit den Gewinn des Gegenspielers zuzulassen. Bei 200 Wiederholungen trifft der Agent stets eine der beiden Entscheidungen und entscheidet sich nie dafür, den Stein in einer anderen Spalte zu platzieren. Beim empfohlenen Wert von $c_{UCT} = \sqrt{2}$ entscheidet er sich mit einer Wahrscheinlichkeit von 20,0 %, den Zug zu blockieren, bei $c_{UCT} = 1.7$ nur 5,5 % und bei $c_{UCT} = 1.1$ sind es 44,0 %. Je schwächer der Exploitation-Teil der UCT-Formel ge-

wichtet wird, desto wahrscheinlicher wird es, dass der Agent den unmittelbaren Gewinn des Gegenspielers verhindert.

Eine mögliche Erklärung für dieses Verhalten besteht darin, dass in den vom MCTS-Agenten durchgeführten Simulationen alle Züge rein zufällig gewählt werden. So kann es passieren, dass für die Züge, die den Gegenspieler nicht blockieren, selten oder gar nicht der Fall eintritt, dass er unmittelbar darauf das Spiel für sich entscheidet. Gleichzeitig besitzt der Zug, der den dritten Stein in die Kette des MCTS-Agenten platziert, eine erhöhte Gewinnwahrscheinlichkeit. So steigt der UCT-Wert des entsprechenden Knotens im MCTS-Baum über den Exploitation-Teil der UCT-Formel, was dazu führt, dass dieser Zug in den Simulationen häufiger untersucht wird und am Ende mit einer höheren Wahrscheinlichkeit gewählt wird.

Bei einem geringeren Wert für c_{UCT} werden Knoten, die selten untersucht wurden, häufiger untersucht als Knoten, bei denen aus den bisherigen Simulationen eine höhere Gewinnwahrscheinlichkeit hervorgegangen ist, sodass für nicht-blockierende Züge häufiger der Fall eintritt, dass der Gegenspieler das Spiel im nächsten Zug gewinnt, wodurch sich die Gewinnrate und damit der Exploitation-Teil aller Züge, die nicht blockieren, sinkt, der blockierende Zug häufiger simuliert und damit letztendlich auch gewählt wird.

Eine mögliche Lösung zum Beheben dieser Schwachstelle besteht darin, während der Simulationen nicht alle Züge rein zufällig zu wählen, sondern die Wahrscheinlichkeit zu erhöhen, dass der Gegenspieler den vierten Stein in eine bereits existierende Dreierkette platziert, sofern dies möglich ist, sodass in den Simulationen nicht-blockierende Züge eine niedrigere Gewinnrate aufweisen und mit einer niedrigeren Wahrscheinlichkeit gewählt werden.

Da das Ziel dieser Arbeit nicht darin besteht, optimale Agenten zu entwickeln, und die Ergebnisse möglichst unabhängig von Optimierungen auf Vier Gewinnt sein sollen, wird diese Funktionalität nicht eingebaut. Aufgrund der zeitlichen Beschränkung dieser Arbeit können mögliche Nebenwirkungen von niedrigen Werten für c_{UCT} nicht untersucht werden. Aus diesem Grund werden die anstehenden Messungen weiterhin mit dem Standardwert von $c_{UCT} = \sqrt{2}$ durchgeführt.

5.3 PPO-Agent

5.4 Szenarien zur Untersuchung von Robustheit

Um die im Konzept genannten Szenarien zu implementieren, wurde die Messumgebung um eine Klasse `DistortionGenerator` erweitert. Über dessen Konstruktor können zwei

Attribute gesetzt werden, eines um die Wahrscheinlichkeit festzulegen, mit der eine Aktion verfälscht werden soll, und ein Weiteres um die Anzahl von falsch zu beobachtenden Feldern zu konfigurieren.

Sie besitzt eine Methode `distort_action(orinal_action: int, action_mask: list[int]) -> int`, die eine von einem Agenten gewählte Aktion und die im aktuellen Zustand möglichen Aktionen entgegennimmt, und mit der im Konstruktor gegebenen Wahrscheinlichkeit eine zufällige Aktion und ansonsten die ursprüngliche Aktion zurückgibt. Sie wird immer dann ausgeführt, nachdem die Messumgebung für einen Agenten einen Zug berechnet hat und es wird die Aktion ausgeführt, die durch diese Funktion zurückgegeben wird.

Die Methode `distort_state(state: numpy.ndarray(6, 7, 2)) -> numpy.ndarray(6, 7, 2)` nimmt den aktuellen Zustand des Spielfelds entgegen. Sie verwaltet zwei Listen, die jeweils Koordinaten von Spielfeldern enthalten. Eine Liste für Felder, bei denen Steiner platziert, und eine weitere Liste für Felder, bei denen Steine entfernt werden können, sodass jeweils illegale Zustände entstehen. In die erste Liste werden die Felder hinzugefügt, dessen unteren Nachbarn frei sind, und in die zweite Liste werden die Felder hinzugefügt, dessen obere Nachbarn besetzt sind. Wie in Kapitel 4 erwähnt, wird dabei beachtet, dass aus vollen Spalten keine Spielsteine entfernt werden und keine Spalten mit nur einem freien Feld gefüllt werden, sodass sich die im beobachteten Zustand möglichen Aktionen nicht von den im tatsächlichen Zustand möglichen Aktionen unterscheiden. Die Methode wird nach jedem Spielzug durchgeführt, sobald der neue Zustand des Spielfelds bekannt ist. Das Ergebnis wird an die Agenten weitergegeben, die auf dessen Grundlage den nächsten Zug wählen.

Eine Frage, die sich bei der Implementierung des Szenarios Unsicherheit bezüglich Beobachtungen stellt, ist wie der MCTS-Agent auf Grundlage des illegalen Spielfeldzustands Simulationen durchführt. Die PettingZoo-Umgebung, die in dieser Implementierung eingesetzt wird, handhabt den Fall, dass ein Stein in eine Spalte platziert wird, in der sich ein Stein über einem freien Feld befindet, so, dass der zu platzierende Stein im untersten freien Feld landet. Eine alternative Herangehensweise wäre, den zu platzierenden Stein immer auf das freie Feld über das höchste besetzte Feld zu setzen. Es wird angenommen, dass die beiden Herangehensweisen keinen wesentlichen Unterschied in der Leistungsfähigkeit des MCTS-Agenten verursachen, daher wird das Verhalten so belassen, wie es in der PettingZoo-Umgebung umgesetzt ist.

Es war jedoch eine Anpassung an das Action Masking der PettingZoo-Umgebung notwendig, also der Art und Weise, wie in berechnet wird, welche Aktionen möglich sind.

Diese Berechnung ist so implementiert, dass Spalten als nicht bespielbar gekennzeichnet werden, sobald das oberste Feld der Spalte belegt ist. Das bedeutet, wenn fälschlicherweise beobachtet wird, dass das oberste Feld einer Spalte besetzt ist, wird sie als nicht bespieltbar gekennzeichnet, auch wenn darunter freie Felder beobachtet werden. Das Action Masking wurde daher so angepasst, dass Spalten erst dann als nicht bespielbar gekennzeichnet werden, wenn alle Felder in der Spalte besetzt sind.

6 Ergebnisdiskussion

7 Zusammenfassung und Ausblick

8 Literaturverzeichnis

[1] Stefano V. Albrecht, Filippos Christianos, Lukas Schäfer. *Multi-Agent Reinforcement Learning: Foundations and Modern Approaches*. MIT Press, 2024. URL: https://www.marl-book.com.

[2] E. Alderton, E. Wopat, J. Koffman. *Reinforcement Learning for Connect Four*. Techn. Ber. Stanford University, Stanford, California 94305, USA, 2019.

[3] James Dow Allen. *The complete book of Connect 4: history, strategy, puzzles*. New York, NY : Puzzle Wright Press, 2010.

[4] Victor Allis. „A Knowledge-Based Approach of Connect-Four“. In: *J. Int. Comput. Games Assoc.* 11 (1988), S. 165. URL: https://api.semanticscholar.org/CorpusID:24540039.

[5] Victor Allis. „Searching for solutions in games and artificial intelligence“. In: 1994. URL: https://api.semanticscholar.org/CorpusID:60886521.

[6] Jörg Bewersdorff. *Glück, Logik und Bluff: Mathematik im Spiel - Methoden, Ergebnisse und Grenzen*. 7. Aufl. Springer Spektrum Wiesbaden, 8. Mai 2018. ISBN: 978-3-658-21764-8. DOI: 10.1007/978-3-658-21765-5.

[7] Cameron B. Browne u. a. „A Survey of Monte Carlo Tree Search Methods“. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (2012), S. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[8] N. Buduma, N. Buduma, J. Papa. *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. O'Reilly Media, 2022. ISBN: 9781492082132.

[9] Guillaume M. J. B. Chaslot, Mark H. M. Winands, H. Jaap van den Herik. „Parallel Monte-Carlo Tree Search“. In: *Computers and Games*. Hrsg. von H. Jaap van den

Herik u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 60–71. ISBN: 978-3-540-87608-3.

[10] Anna Choromanska u. a. *The Loss Surfaces of Multilayer Networks*. 2015. arXiv: 1412.0233 [cs.LG]. URL: https://arxiv.org/abs/1412.0233.

[11] Milton Bradley Company. *Connect Four*. https://www.unco.edu/hewit/pdf/giant-map/connect-4-instructions.pdf. [Letzer Zugriff am 17. Dezember 2024]. 1990.

[12] Mayank Dabas, Nishthavan Dahiya, Pratish Pushparaj. „Solving Connect 4 Using Artificial Intelligence“. In: *International Conference on Innovative Computing and Communications*. Hrsg. von Ashish Khanna u. a. Singapore: Springer Singapore, 2022, S. 727–735. ISBN: 978-981-16-2594-7.

[13] P. Fergus, C. Chalmers. *Applied Deep Learning: Tools, Techniques, and Implementation*. Computational Intelligence Methods and Applications. Springer International Publishing, 2022. ISBN: 9783031044199. URL: https://books.google.de/books?id=eJv5zgEACAAJ.

[14] Kevin Ferguson, Max Pumperla. *Deep Learning and the Game of Go*. Manning Publications, January 2019.

[15] Matthias Feurer, Frank Hutter. „Hyperparameter Optimization“. In: *Automated Machine Learning: Methods, Systems, Challenges*. Hrsg. von Frank Hutter, Lars Kotthoff, Joaquin Vanschoren. Cham: Springer International Publishing, 2019, S. 3–33. ISBN: 978-3-030-05318-5. DOI: 10.1007/978-3-030-05318-5_1. URL: https://doi.org/10.1007/978-3-030-05318-5_1.

[16] The Farama Foundation. *PettingZoo Documentation*. https://pettingzoo.farama.org/. (Besucht am 26. 01. 2025).

[17] Alfred Früh, Dario Haux. *Foundations of Artificial Intelligence and Machine Learning*. Bd. 29. Weizenbaum Series. Berlin: Weizenbaum Institute for the Networked

Society - The German Internet Institute, 2022, S. 25. DOI: https://doi.org/10.34669/WI.WS/29.

[18] Marta Garnelo, Murray Shanahan. „Reconciling deep learning with symbolic artificial intelligence: representing objects and relations“. In: *Current Opinion in Behavioral Sciences* 29 (2019). Artificial Intelligence, S. 17–23. ISSN: 2352-1546. DOI: https://doi.org/10.1016/j.cobeha.2018.12.010. URL: https://www.sciencedirect.com/science/article/pii/S2352154618301943.

[19] George T. Heineman, Gary Pollice, Stanley Selkow. *Algorithms in a Nutshell*. O'Reilly Media, Inc., October 2008.

[20] B.G. Humm. *Applied Artificial Intelligence: An Engineering Approach*. Independently Published, 2020. ISBN: 9798635591154.

[21] „IEEE Standard Glossary of Software Engineering Terminology“. In: *IEEE Std 610.12-1990* (1990), S. 1–84. DOI: 10.1109/IEEESTD.1990.101064.

[22] Levente Kocsis, Csaba Szepesvári. „Bandit Based Monte-Carlo Planning“. In: *Machine Learning: ECML 2006*. Hrsg. von Johannes Fürnkranz, Tobias Scheffer, Myra Spiliopoulou. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006, S. 282–293. ISBN: 978-3-540-46056-5.

[23] Zoltán Micskei u. a. „Robustness Testing Techniques and Tools“. In: *Resilience Assessment and Evaluation of Computing Systems*. Hrsg. von Katinka Wolter u. a. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 323–339. DOI: 10.1007/978-3-642-29032-9_16. URL: https://doi.org/10.1007/978-3-642-29032-9_16.

[24] Janosch Moos u. a. „Robust Reinforcement Learning: A Review of Foundations and Recent Advances“. In: *Machine Learning and Knowledge Extraction* 4.1 (2022), S. 276–315. ISSN: 2504-4990. DOI: 10.3390/make4010013. URL: https://www.mdpi.com/2504-4990/4/1/13.

[25] Tianwei Ni, Benjamin Eysenbach, Ruslan Salakhutdinov. „Recurrent Model-Free RL is a Strong Baseline for Many POMDPs“. In: *CoRR* abs/2110.05038 (2021). arXiv: 2110.05038. URL: https://arxiv.org/abs/2110.05038.

[26] Aditya Jyoti Paul. „Randomized fast no-loss expert system to play tic tac toe like a human“. In: *CoRR* abs/2009.11225 (2020). arXiv: 2009.11225. URL: https://arxiv.org/abs/2009.11225.

[27] Yiran Qiu, Zihong Wang, Duo Xu. „Comparison of Four AI Algorithms in Connect Four“. In: *MEMAT 2022; 2nd International Conference on Mechanical Engineering, Intelligent Manufacturing and Automation Technology*. 2022, S. 1–5.

[28] S.J. Russell, S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Pearson series in artificial intelligence. Pearson, 2020. ISBN: 9780134610993.

[29] Jonathan Schaeffer u. a. „Checkers Is Solved“. In: *Science* 317 (Okt. 2007), S. 1518–1522. DOI: 10.1126/science.1144079.

[30] N. Shawki u. a. „On Automating Hyperparameter Optimization for Deep Learning Applications“. In: *2021 IEEE Signal Processing in Medicine and Biology Symposium (SPMB)*. 2021, S. 1–7. DOI: 10.1109/SPMB52430.2021.9672266.

[31] Kavita Sheoran u. a. „Solving Connect 4 Using Optimized Minimax and Monte Carlo Tree Search“. In: *Advances and Applications in Mathematical Sciences* 21.6 (2022), S. 3303–3313.

[32] Richard S. Sutton, Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: A Bradford Book, 2018. ISBN: 0262039249.

[33] Maciej Swiechowski u. a. „Monte Carlo Tree Search: A Review of Recent Modifications and Applications“. In: *CoRR* abs/2103.04931 (2021). arXiv: 2103.04931. URL: https://arxiv.org/abs/2103.04931.

[34] Henry Taylor, Leonardo Stella. *An Evolutionary Framework for Connect-4 as Test-Bed for Comparison of Advanced Minimax, Q-Learning and MCTS*. 2024. arXiv: 2405.16595 [cs.AI]. URL: https://arxiv.org/abs/2405.16595.

[35] Markus Thill, Patrick Koch, Wolfgang Konen. *Reinforcement Learning with N-tuples on the Game Connect-4*. Techn. Ber. Department of Computer Science, Cologne University of Applied Sciences, 51643 Gummersbach, Germany, 2012.

[36] John Tromp. *John's Connect Four Playground*. https://en.wikipedia.org/w/index.php?title=Wine&oldid=1262619132. [Letzer Zugriff am 13. Dezember 2024].

[37] Stephan Wäldchen, Felix Huber, Sebastian Pokutta. *Training Characteristic Functions with Reinforcement Learning: XAI-methods play Connect Four*. 2022. arXiv: 2202.11797 [cs.LG]. URL: https://arxiv.org/abs/2202.11797.