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 regelbasierte Algorithmen und Machine Learning. 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.

Im Rahmen dieser Arbeit werden Robustheit eines symbolischen Algorithmus und eines Reinforcement Learning (RL) basierten Ansatzes zur Lösung des Brettspiels Vier Gewinnt untersucht. Bei Robustheit handelt es sich um eine Eigenschaften, die beschreibt, wie gut ein Algorithmus oder RL-Modell in der Praxis funktioniert, in der andere Bedingungen herrschen können als während der Entwicklung und Qualitätssicherung. Diese Kriterien sind besonders relevant für den Erfolg von Algorithmen und Modellen in der Praxis [19][20].

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 algorithmische als auch Reinforcement Learning basierte Lösungen eignen[22][25][2], [28], [11], [31].

Es wird Grundlagenforschung zu verbreiteten algorithmischen Ansätzen und Reinforcement Learning basierten Ansätzen betrieben. Anschließend wird die Robustheit von zwei Ansätzen aus den jeweiligen Bereichen am Beispiel von Vier Gewinnt empirisch untersucht. Dabei werden neue Erkenntnisse über Lösungsansätze von Vier Gewinnt herausgearbeitet, die sich auf vergleichbare Szenarien in der realen Welt übertragen lassen.

Die zentrale Fragestellung lautet: Inwiefern sind bei Vier Gewinnt symbolische Algorithmen oder Reinforcement Learning robuster? Das Ziel dieser Arbeit besteht darin, ein detailliertes Verständnis über verschiedene Aspekte von Robustheit der zu untersuchenden Ansätze zu bekommen.

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 [10].

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; [13], 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.; [23], 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.
- 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 wird, könnte diese Funktion 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, eine Policy zu finden, die jedem den Zuständen jeweils eine Aktion zuordnet, und dabei zur höchsten erwartbaren Belohnung führt ([23], 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, [23], 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 [21]. Für Schach wird die Spielbaumkomplexität auf 10^{31} geschätzt und die Aussichten auf eine starke Lösung liegen noch in weiter Ferne [24].

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 [30].

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 ([16], 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][29][31][28][25][22]. 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 ([12], S. 4; [14], S. 5 f., [15]).

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 (Ferguson Kapitel 4; Algorithms in a Nutshell, Kapitel 7.6 u. 7.8). Darauf folgt die Erklärung der Monte Carlo Tree Search, welche einen allgemeineren Ansatz darstellt ([23], S. 580; [27]).

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. Zurückgegeben 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 Bewertunt der Knoten muss auf Grundlage von Heuristiken erfolgen ([13], Kapitel 4; [16], 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 ([16], Kapitel 7.8; [13], 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 ([16], 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 ([16], 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 ([13], 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 ([23], 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 ([23], S. 161; [13], 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 ([23], 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 ([23], S. 161 ff.; [13], 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 Blattknoten erreicht wird, der in den folgenden Phasen untersucht werden soll. 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. Sie bewertet die Kinder eines Knotens n mit der Formel UCB1 und wählt den Knoten mit der höchsten Bewertung ([23], S. 163). Die Formel lautet wie folgt:

$$UCB1 = \frac{U(n)}{N(n)} + c * \sqrt{\frac{log(N(Parent(n)))}{N(n)}}$$

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 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 ([23], S. 163).

Im Zuge der **Expansion** wird vom zuvor ausgewählten Knoten ein zufälliger Zug aus-

geführt 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.

In der **Simulation**-Phase werden vom 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.

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.

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 ([23], S. 164).

Dadurch, dass Iterationen beliebig oft durchgeführt werden können, um die Vorhersagen 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 ([23], 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 ([17], 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 ([23], S. 789 f.; [26], 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 ([23], S. 850; [26], 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 ([23], S. 803 ff.). Es wurde sogar gezeigt, dass bei genügend Training die Leistung von Q-Learning-Agenten zu perfekten Verhalten konvergiert ([26], 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 ([23], S. 803, [26], 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 ([23], S. 809; [26], 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 ([23], S. 790; [26], 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 ([26], 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 ([23], 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], [28], [11], [31].

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 ([23], S. 790).

Bei strategiebasierten Methoden wird das Regelwerk nicht aus einer Nutzenfunktion abgeleitet, sondern das Regelwerk wird direkt erlernt ([23], S. 790). Gegenüber wertba-

sierten 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.; [23], 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 ([23], 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; [26], 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 ([23], 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.; [23], 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 ([23], 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 ([26], 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 ([13], Kapitel 10.3; [8], Kapitel 4). Daher existieren Verfahren, die die Lernrate dynamisch anpassen können, um den Optimierungsprozess zu beschleunigen ([1], S. 174).

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 ([23], 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 ([26], S. 200; [13], Kapitel 5.4.4; [9]).

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. Außerdem 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 ([26], 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 ([26], 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 ([26], 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 ([26], 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“([18], S. 64).

In zwei unabhängigen Studien wurde der Begriff 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[19][20].

Das Ziel beim robusten Reinforcement Learning besteht darin, ein Regelwerk zu finden, das auch unter ungünstigen Bedingungen, möglichst gute Entscheidungen trifft. Aus den Studien geht hervor, dass Robustheit in die Teile des modellierten MDPs aufgeteilt werden kann, das von den Unsicherheiten betroffen ist:

- Unsicherheit bezüglich Aktionen: Es wird eine andere Aktion ausgeführt, als die für die sich der Agent entschieden hat.
- Unsicherheit bezüglich Beobachtungen: Der Zustand, den der Agent beobachtet, entspricht nicht dem tatsächlichen Zustand der Umgebung.
- Unsicherheit bezüglich Dynamik der Umgebung: Die Übergangswahrscheinlichkeiten zwischen den Zuständen sind anders als erwartet.

Es existiert eine Reihe von Herangehensweisen, um RL-Verfahren besonders robust zu machen, diese werden im Rahmen dieser Arbeit jedoch nicht weiter betrachtet[19][20].

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 erklärt, wie eine Messumgebung aufgesetzt wurde, um die Robustheit der Lösungsverfahren empirisch zu bewerten. Diese Messumgebung ermöglicht es, zwei Agenten, die die zu untersuchenden Ansätze implementieren, das Spiel wiederholt gegeneinander spielen zu lassen. Die Spiele werden unter verschiedenen Szenarien durchgeführt, die jeweils auf ein in Kapitel 3.4 definierten Aspekt der Robustheit abzielen. Es werden die Gewinnraten gemessen, worüber Aussagen darüber getroffen werden können, welches der beiden Verfahren in den verschiedenen Szenarien stärker und damit robuster ist. Folgende Szenarien werden untersucht:

- 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 wesentlich 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 ([23], 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.

In den verschiedenen Szenarien gelten die veränderten Bedingungen nur für jeweils einen Agenten. Wenn beide Agenten gleichzeitig von den veränderten Bedingungen betroffen wären, könnten nur Aussagen über die relative Robustheit zueinander getroffen werden. Dadurch dass die veränderten Bedingungen nur für jeweils ein Verfahren auf einmal gelten, kann genau gesagt werden, welches Verfahren wie stark betroffen ist.

Die Messungen werden nicht nur unter Szenarien mit veränderten Bedingungen durchgeführt, sondern auch in einer neutralen Umgebung ohne veränderte Bedingungen, um eine Grundlage für die Analyse der Ergebnisse zu bilden.

Um einen fairen Vergleich zu gewährleisten, müssen die Agenten in der neutralen Umgebung gleich stark sein. Dazu werden die Parameter der Agenten (Anzahl der Simulationen beim MCTS Agenten, Anzahl der durchlaufenen Self-Play-Trainingsepisoden beim RL-Agenten) so eingestellt, dass sie in der neutralen Umgebung im Spiel gegeneinander jeweils eine Gewinnrate von etwa 50% aufweisen.

Außerdem ist es wichtig, dass die Agenten auf einem gewissen starken 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. Als Indikator für die Spielstärke der Agenten wird die durchschnittliche Spieldauer verwendet. Es wird angenommen, dass starke Agenten weniger Fehler machen und ihre Züge strategischer wählen, sodass komplexere Spielsituationen

entstehen und sich die Entscheidung des Spiels hinauszögert.

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

5 Realisierung

Als Grundlage für die Messumgebung dient das PettingZoo-Toolkit. Es abstrahiert Probleme in Umgebungen und stellt eine Schnittstelle für Agents bereit, die mit verschiedene Lösungsstrategien mit den Umgebungen interagieren. Eine Umgebung, die das Spiel Vier Gewinnt abstrahiert, ist Teil des PettingZoo Toolkits. Es kommen Reinforcement Learning Modelle zum Einsatz, die aus RL-Bibliotheken wie CleanRL oder Stable-Baselines bereitgestellt werden. Falls vorhanden, wird auf fertig implementierte Algorithmen zurückgegriffen.

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

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

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

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

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

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

[19] 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.

[20] 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.

[21] 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.

[22] 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.

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

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

[25] 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.

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

[27] 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.

[28] 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.

[29] 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.

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

[31] 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.