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

In dieser Arbeit wird die Robustheit von symbolischen Algorithmen und Reinforcement-Learning-Verfahren miteinander verglichen. Dies geschieht am Beispiel des Brettspiels Vier Gewinnt. Dazu wurden zwei Agenten implementiert, die das Spiel selbstständig spielen. Beim ersten Agenten kommt der symbolische Algorithmus Monte Carlo Tree Search (MCTS) zum Einsatz. Dem zweiten Agenten liegt das RL-Verfahren Proximal Policy Optimization (PPO) zugrunde. Die Robustheit der Agenten wird quantifiziert, indem der Verlust der Gewinnrate gegen einen zufällig spielenden Agenten gemessen wird, während die zu untersuchenden Agenten zwei verschiedenen Szenarien mit ungünstigen Bedingungen ausgesetzt sind, die auf verschiedene Aspekte von Robustheit abzielen. Im ersten Szenario liegt Unsicherheit bezüglich Aktionen vor, was bedeutet, dass die Agenten keine vollständige Kontrolle darüber haben, in welche Spalte sie ihre Spielsteine platzieren. Im zweiten Szenario erhalten die Agenten fehlerhafte Informationen über das Spielfeld, somit liegt Unsicherheit bezüglich Beobachtungen vor.

Es wird gezeigt, dass unter den beiden implementierten Agenten der MCTS-Agent im Szenario Unsicherheiten bezüglich Aktionen robuster ist als der PPO-Agent. Für das Szenario Unsicherheiten bezüglich Beobachtungen wurde dies bis zu einem gewissen Ausmaß an Unsicherheiten ebenfalls beobachtet. Die Ergebnisse dieser Arbeit lassen sich jedoch nicht auf den Vergleich von Robustheit zwischen MCTS und PPO unabhängig von der konkreten Implementierung oder dem Anwendungsfall übertragen, da für einen fairen Vergleich ein wesentlich aufwändigeres Training des PPO-Agenten notwendig gewesen wäre. Dafür und um allgemeine Aussagen über den Vergleich der Robustheit zwischen symbolischen Algorithmen und Reinforcement Learning zu übertragen, sind weitere Untersuchungen erforderlich.

2. Abstract

This thesis investigates the relative robustness of symbolic algorithms versus reinforcement learning methods, using the board game Connect Four as a case study. Two agents were implemented that play the game autonomously. The first agent utilizes the symbolic algorithm Monte Carlo Tree Search (MCTS), while the second agent is based on the reinforcement learning method Proximal Policy Optimization (PPO). The robustness of the agents is quantified by measuring the decrease in win rate against an agent that plays randomly, while the agents under investigation are subjected to two different scenarios with unfavorable conditions that target various aspects of robustness. In the first scenario, there is uncertainty regarding actions, meaning that the agents do not have complete control over the column in which they place their pieces. In the second scenario, the agents receive erroneous information about the game board, thereby introducing uncertainty regarding observations.

It is shown that, among the two implemented agents, the MCTS agent is more robust than the PPO agent in the scenario involving uncertainty regarding actions. For the scenario with uncertainty regarding observations, this was also observed up to certain levels of uncertainty. However, the results of this study cannot be generalized to a comparison of robustness between MCTS and PPO, independent of the specific implementation or application case, as a significantly more extensive training of the PPO agent would have been necessary for a fair comparison. Further investigations are necessary to draw general conclusions regarding the comparison of robustness between symbolic algorithms and reinforcement learning.

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, 07.03.2025

Leo Herrmann

3. Einleitung

Zahlreiche Bereiche der Gesellschaft werden durch fortschreitende Automatisierung durchdrungen, so zum Beispiel die Fertigungsindustrie, das Gesundheitswesen oder der 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, ihre Funktionalität unter genau solchen veränderten Bedingungen erhalten bleibt. Robustheit ist daher besonders relevant für den Erfolg von Algorithmen und Modellen in der Praxis und wird in der Literatur ausgiebig untersucht [26] [27] [28].

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; [21], S. 12 f.; [17], S. 5f.). Robustheit wird in diesem Zusammenhang gewöhnlich nicht genannt, jedoch stellt sich auch hier die Frage, inwiefern bei der Entscheidung zwischen symbolischen und ML-basierten Lösungsansätzen nicht auch Robustheit ein relevantes Kriterium ist, und wenn 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 die Eignung von sowohl symbolischen Lösungsansätzen als auch Reinforcement Learning als Verfahren aus dem Bereich ML ersichtlich wird [30] [35] [38] [12] [43].

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. Auf Grundlage der gewählten Verfahren wird jeweils ein Agent implementiert, der das Spiel selbstständig spielt. Die Robustheit der Agenten wird quantifiziert, indem der Verlust der Gewinnrate gegen einen zufällig spielenden Agenten gemessen wird, während die zu untersuchenden Agenten verschiedenen Szenarien mit ungünstigen Bedingungen ausgesetzt sind, die auf verschiedene Aspekte von Robustheit abzielen. Dadurch sollen neue Erkenntnisse darüber gewonnen werden, wie Robustheit als Kriterium für Entscheidungen zwischen symbolischen Algorithmen und Reinforcement Learning eingesetzt werden kann.

4. 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.

4.1. Vier Gewinnt

Vier Gewinnt ist ein Brettspiel, dessen Spielfeld aus 6 Zeilen und 7 Spalten besteht. Durch die beiden Spieler wird abwechselnd ein Spielstein in eine Spalte platziert, der in dieser Spalte bis zur untersten freien Position fällt. Es gewinnt der Spieler, durch den zuerst vier Spielsteine in einer horizontalen, vertikalen oder diagonalen Linie nebeneinander platziert wurden [11]. Abbildung 1 zeigt eine diagonale Gewinnposition für den Spieler, der die blauen Steine platziert.

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 den Spieler den besten weiteren Spielverlauf verspricht ([5], S. 96-100; [14], Kapitel 4.1).

Bei Zwei-Spieler-Nullsummenspielen verursacht der Sieg eines Spielers zwangsläufig die Niederlage des anderen Spielers. Die beiden Spieler haben damit entgegengesetzte Interessen ([5], S. 100; [4], S. 6). Das bedeutet, dass sich der Erfolg von verschiedenen

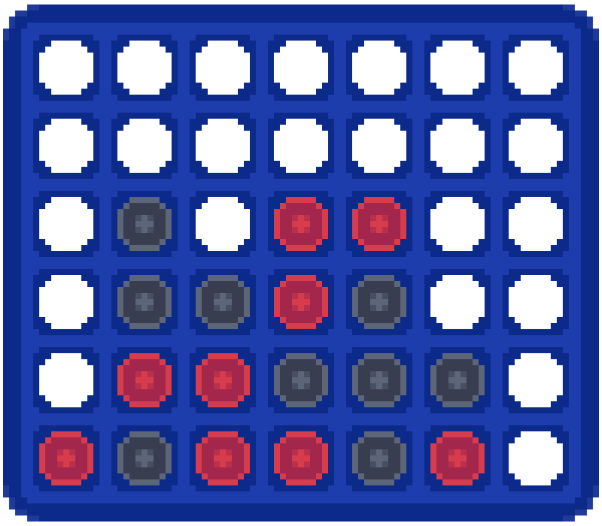


Abb. 1: Spielfeld von Vier Gewinnt mit einer diagonalen Gewinnposition für Blau. Grafikdesign aus [39].

Lösungsansätzen durch die durchschnittliche Gewinnrate im Spiel gegeneinander bewerten lässt. Bei Nullsummenspielen mit mehr als zwei Spielern, kann es passieren, dass eine (bewusst oder versehentlich) suboptimale Spielweise eines Spielers, dazu führt, dass ein zweiter Spieler davon profitiert, während ein dritter Spieler dadurch benachteiligt wird. Solche Wechselwirkungen sind bei Zwei-Personen-Nullsummenspielen ausgeschlossen ([5], S. 113 ff.; [32], S. 151 f.). Dadurch werden die Messergebnisse im Hauptteil besser vergleichbar.

4.1.1. 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 aufeinanderfolgenden und voneinander abhängigen Entscheidungsmöglichkeiten, 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. In bereits vollständig gefüllten Spalten können keine weiteren Spielsteine platziert werden.

- Übergangsmodell $P(s, a, s')$, das die Wahrscheinlichkeit beschreibt, von einem Zustand s durch die Aktion a zu einem weiteren Zustand s' zu gelangen. Das Übergangsmodell hängt im Fall von kombinatorischen Spielen von der Strategie der anderen Spieler ab. Bei der Modellierung als MDP besteht die Annahme, dass sich die Strategie des Gegenspielers im Laufe des Spiels nicht ändert. Das Übergangsmodell ist dem Spieler nicht bekannt.

- Belohnungsfunktion $R(s, a, s')$, wodurch jedem Zustandsübergang eine Belohnung zugeordnet wird. 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. Durch die Anwendung des Regelwerks soll die höchste erwartbaren Belohnung erzielt werden ([32], S. 562 f.).

4.1.2. 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 Zwei-Spieler-Nullsummenspiele kann dieser Wert oder zumindest dessen obere Schranke bestimmt werden. Dazu werden zunächst alle Konfigurationen des Spielfelds gezählt, dann Einschränkungen wie Regeln und Symmetrie berücksichtigt, und die Anzahl der illegalen und redundanten Zustände von der Anzahl aller möglichen Konfigurationen abgezogen ([4], 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 ([5], S. 102; [32], 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 ([4], 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 [29]. Für Schach wird die Spielbaumkomplexität auf 10^{31} geschätzt und die Aussichten auf eine starke Lösung liegen noch in weiter Ferne [33].

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 ([4], S. 163).

4.1.3. Lösungsverfahren

Für Vier Gewinnt wurden bereits verschiedene Lösungsverfahren 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 mit Anzugsrecht bei optimaler Spielweise immer gewinnen [2] [3]. Der Begriff Anzugsrecht bezeichnet das Recht eines Spielers, den ersten Zug durchführen zu dürfen, und in dieser Arbeit in Anlehnung an ([5]) verwendet.

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 [41].

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 [40] [43] [38] [35] [30]. Wissensbasierte Methoden werden in dieser Arbeit nicht näher betrachtet, da sie stark an die jeweiligen Spielregeln gebunden und ihre Eigenschaften schwer zu verallgemeinern sind.

4.2. 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. Zunächst werden die Algorithmen zur Lösung von Spielbäumen Minimax und dessen Optimierung Alpha-Beta-Pruning vorgestellt ([14], Kapitel 4; [19], Kapitel 7.6 u. 7.8). Darauf folgt die Erklärung der Monte Carlo Tree Search, welche einen allgemeineren Ansatz darstellt ([32], S. 580; [37]).

4.2.1. 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 der Elternknoten 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 Bewertung der Knoten muss auf Grundlage von Heuristiken erfolgen ([14], Kapitel 4; [19], Kapitel 7.6).

4.2.2. Alpha-Beta-Pruning

Beim Alpha-Beta-Pruning handelt es sich um eine Optimierung des Minimax-Algorithmus. Dabei werden die Teilbäume übersprungen, die bei optimaler Spielweise beider Spieler nicht erreicht werden, und damit das Ergebnis nicht beeinflussen können. 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 im Rahmen dieser Arbeit 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.

4.2.3. 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 ([32], 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 ([32]; [14]; [7]). 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, wie zum Beispiel den Gewinner oder eine Punktzahl ([32], 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 gescheitert sind. Der Algorithmus wird auch abseits von Spielen in verschiedenen Variationen eingesetzt, so zum Beispiel in der Optimierung von Lieferketten und der Zeitplanung von Prozessen ([32], S. 161; [7]).

Mit MCTS werden Entscheidungsmöglichkeiten statistisch ausgewertet, indem die in Abbildung 2 dargestellten 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 ([32], S. 161 ff.; [14], Kapitel 4.5).

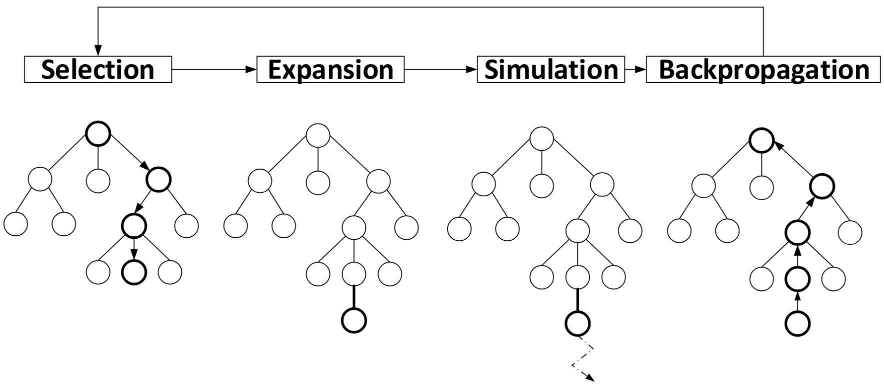


Abb. 2: Die vier Phasen des MCTS-Algorithmus [37].

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 neue Bereiche im Spielbaum mit ggf. besseren Chancen zu erkunden (Exploration). Es gibt verschiedene Auswahlstrategien, wobei UCT (Upper Confidence Bound applied for Trees) die am weitesten verbreitete ist. Dabei wird die UCB1-Formel (1) eingesetzt, die ursprünglich zur Lösung von Bandit-Problemen (vgl. [32], S. 581 ff.) konzipiert wurde, um die Kinder eines Knotens n zu bewerten. Anschließend wird der Knoten mit der höchsten Bewertung gewählt ([24]; [7]; [32], S. 163).

$$UCT = \frac{U(n)}{N(n)} + c_{UCT} \cdot \sqrt{\frac{log(N(Parent(n)))}{N(n)}} \tag{1}$$

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 Term $\frac{U(n)}{N(n)}$ den Exploitation-Teil dar. Er wächst mit den Erfolgsaussichten des untersuchten Knotens. $N(Parent(n))$ ist die Anzahl der Simulationen des Elternknotens von n. Der Term $\sqrt{\frac{log(N(Parent(n)))}{N(n)}}$ 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 ([32], 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 ([32], 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 die getroffenen Entscheidungen nicht in den MCTS-Baum aufgenommen werden ([32], S. 162). Besteht eine Simulation aus besonders vielen Zügen, kann es sinnvoll sein, sie nach einer bestimmten Anzahl von Zügen abzubrechen. Der letzte erreichte Zustand kann dann heuristisch oder mit einem neutralen Ergebnis bewertet werden ([32], 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 ([32], 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 die Gefahr, dass wenige kritische Züge unentdeckt bleiben, und die Statistik inakkurat wird ([32], S. 164).

Die Laufzeit von MCTS ist schwer mit den zuvor genannten Methoden vergleichbar, da die Iterationen beliebig oft durchgeführt werden können, um die Vorhersagen zu verbessern. Durch Untersuchungen wurde gezeigt, dass bei kleinen Problemen und der Verfügbarkeit von präzisen Heuristiken Alpha Beta mit begrenzter Suchtiefe schneller und besser arbeitet. MCTS schneidet im Vergleich besser ab, je tiefer und stärker verzweigt die zu lösenden Entscheidungsbäume sind ([32], S. 163 f.). Es wurde außerdem gezeigt, dass die Ergebnisse von MCTS unter Verwendung der 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.

4.3. 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 ([21], 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 erfolgversprechendsten 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 ([32], S. 789 f.; [36], 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 ([32], S. 850; [36], S. 450).

4.3.1. 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 ([32], S. 803 ff.). Es wurde sogar gezeigt, dass bei genügend Training die Leistung von Q-Learning-Agenten zu perfekten Verhalten konvergiert ([36], S. 140). Vier Gewinnt hat allerdings eine wesentlich höhere Zustandskomplexität von 10^{14} [4]. 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 ([32], S. 803; [36], 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 ([32], S. 809; [36], 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, das 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 ([32], S. 790; [36], S. 7). Es ist anzumerken, dass alle in Kapitel 4.2 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 ([36], S. 12). Aufgrund der Fähigkeit, Vorhersagen über die Umgebung treffen zu können, weisen modellbasierte Methoden eine höhere Sample Complexity auf. Dies bedeutet, dass beim

Training weniger Versuche benötigt werden, um ein effektives Regelwerk zu erlernen. Das ist besonders vorteilhaft, wenn Versuche teuer sind und es eine Herausforderung darstellt, genügend Daten zu erheben, so zum Beispiel beim Training in der realen Welt ([32], 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 [38] [12] [43].

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 ein Regelwerk zum Einsatz, das entsprechend der erlernten Nutzenfunktion für jeden Zustand immer die Aktion mit der besten Bewertung wählt ([32], S. 790).

Bei strategiebasierten Methoden wird das Regelwerk nicht aus einer Nutzenfunktion abgeleitet, sondern das Regelwerk wird direkt erlernt ([32], S. 790). Gegenüber wertbasierten Methoden hat das den Vorteil, dass dadurch auch Regelwerke modelliert 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 kontinuierliche Aktionsräume abbilden können, während wertbasierte Methoden nicht dazu in der Lage sind ([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 dasselbe Lernverfahren einsetzen. Im Training lernen die Agenten, gegenseitig ihre Schwächen auszunutzen und diese Schwächen zu beheben. Es ist jedoch auch möglich, Mixed-Play anzuwenden, was bedeutet, 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 nicht-stationären 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 eignen 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. Agenten mit On-Policy-Algorithmen hingegen lernen nur anhand des letzten Lernvorgangs und damit der aktuellsten Strategie der anderen Agenten. Das kann zu stabilerem Lernverhalten führen ([1], S. 224 f.).

4.3.2. 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 4.3.1 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 so lange, bis die Ausgabeschicht erreicht wurde. Sie enthält so viele Neuronen, wie Ausgabewerte berechnet werden sollen ([1], S. 165 f.; [32], 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 ([32], 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; [36], 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 Schichten anstatt mehr Neuronen pro Schicht besitzen ([32], S. 769).

Als Aktivierungsfunktion sind derzeit Rectified Linear Unit (ReLU) und Variationen davon verbreitet. ReLU gibt für Eingabewerte < 0 den Wert 0 und ansonsten den Eingabewert zurück ([1], S. 167 f.; [32], 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 ([32], 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 ([36], 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 $\nabla_\theta L(\theta)$ 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. Im Zuge des Gradientenverfahrens werden die Parameter θ wiederholt nach Formel (2) angepasst, bis ein Minimum erreicht wurde.

$$\theta \leftarrow \theta - \alpha \cdot \nabla_\theta L(\theta) \tag{2}$$

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] [34].

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 ([32], 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 wesentlich niedrigeren Wert zu finden, sehr klein ist ([36], S. 200; [14], Kapitel 5.4.4; [10]).

4.3.3. Advantage Actor-Critic

Advantage 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 parametrisierte 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 kombiniert ([1], S. 202 ff.).

Strategiebasierte 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 ([36], S. 324; [1], S. 195)

Das Policy-Gradient-Verfahren, das die Grundlage des Actors in A2C bildet, ist das REINFORCE-Verfahren ([1], S. 203; [36], S. 326 f.). 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 wird der Verlust L in Abhängigkeit der Parameter ϕ des Regelwerks π als

$$L(\phi) = \gamma^t \cdot G_t \cdot log(\pi(A_t|S_t, \phi)) \tag{3}$$

betrachtet. Die Parameter ϕ werden im Rahmen des Gradientenverfahrens nach Formel (4) angepasst, wobei gilt $\nabla log(x) = \frac{\nabla x}{x}$.

$$\phi \leftarrow \phi + \alpha\gamma^t \cdot G_t \cdot \frac{\nabla\pi(A_t|S_t, \phi)}{\pi(A_t|S_t, \phi)} \tag{4}$$

Somit wird die Wahrscheinlichkeit erhöht, bei Zustand S_t die Aktion A_t auszuführen. Das geschieht proportional zur diskontierte Belohnung G_t, sodass die Wahrscheinlichkeit für Züge mit größerer Belohnung stärker erhöht wird. Der Gradient $\nabla\pi(A_t|S_t, \phi_t)$ 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 diskontierte Belohnung G_t errechnet sich dabei nach Formel (5) aus einer gewichteten Summe der in der Episode erhaltenen Belohnungen.

$$G_t \leftarrow \sum_{k=t+1}^{T} \gamma^{k-t-1} \cdot R_k \tag{5}$$

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 ([36], S. 55, 328).

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 ([36], 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 ([36], 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 ([1], 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 ([1], S. 202 ff.)

Bei A2C wird während des Trainings nach jeder durchgeführten Aktion ein Advantage-Wert anhand von Formel (6) 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.

$$Adv(s_t, a_t) = r_t + \gamma V(s_{t+1}, \theta) - V(s_t, \theta) \tag{6}$$

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 gemäß Formel (2) aktualisiert. Der Verlust L wird hierbei mit

$$L(\theta) = (r_t + \gamma V(s_{t+1}, \theta) - V(s_t, \theta))^2 \tag{7}$$

als Quadrat des Advantage-Wertes definiert ([1], S. 205).

Es ist anzumerken, dass hierbei Endzustände einer gesonderten Betrachtung bedürfen (vgl. [1], 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 nach Formel (8) die Parameter bei jedem Schritt im Training proportional zum über die Critic-Komponente berechneten Advantage-Wert aktualisiert ([1], S. 205; [36], S. 331).

$$\theta \leftarrow \theta + \alpha Adv(s_t, a_t) \frac{\nabla \pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)} \tag{8}$$

Das bedeutet, der zu minimierende Verlust L wird durch

$$L(\theta) = -Adv(s_t, a_t) log(\pi(A_t|S_t, \theta_t)) \tag{9}$$

berechnet. Somit besteht das Ziel der Actor-Komponente darin, den Advantage-Wert zu maximieren. Dadurch wird die Wahrscheinlichkeit für Züge, die besser sind als erwartet erhöht, und für solche die schlechter sind als erwartet, reduziert.

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 ([1], S. 202).

Im Rahmen der Realisierung kommt eine optimierte Variante von A2C zum Einsatz, die sich Proximal Policy Optimization (PPO) nennt. Sie enthält unter anderem Mechanismen, die große Sprünge in den Veränderungen des Regelwerks verhindert, was zu einem effizienterem Trainingsverhalten und einer geringeren Sensibilität gegenüber Hyperparametervoreinstellungen führt ([1], S. 206 ff.).

4.4. 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“ ([22], S. 64).

In der Studie von Moos et al. 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 zuverlässig 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 [27] [28].

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 nachdem in welcher Hinsicht dadurch erhöhte Robustheit erzielt werden soll. Die Kategorien betreffen jeweils unterschiedliche Aspekte 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 ([27], S. 301 ff.). 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 ([27], S. 304 ff.). Auch das lässt sich in Vier Gewinnt abbilden, indem die Agenten nicht den tatsächlichen Zustand, sondern eine manipulierte Version als Entscheidungsgrundlage erhalten.

Es ist anzumerken, dass hierbei aus dem MDP ein Partially Observable MDP (POMDP) wird. Ein POMDP ist ein MDP, dessen Umgebung nur teilweise oder fehlerhaft beobachtbar ist. Der betroffene Agent kann nicht mit Sicherheit sagen, 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 erhalten. 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 ([32], 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.

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 Modellierungsfehlern im Training erhöht werden ([27], S. 291 ff.). 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 ([27], S. 301).

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 ein Kontrahent trainiert wird, der durch bestimmte Störfaktoren, z.B. die Änderung von physikalischen Parametern der Umgebung, die Leistungsfähigkeit des ersten Agenten senken soll ([27], S. 290). 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 solche, die auf Robustheit optimiert sind ([27], 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.

5. Konzept

In diesem Kapitel wird erläutert, wie die Fragestellung dieser 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, und im Gegensatz zum Minimax-Algorithmus und Alpha-Beta-Pruning auch bei komplexeren Problemen mit einer akzeptablen Laufzeit angewandt 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 4.3 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 4.4 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 oder sein 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.

Die Reduktion 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.

Daraus ergibt sich Formel (10) zur Bewertung des Gewinnratenverlustes als Maß für die Robustheit eines Agenten:

$$l = \frac{r_n - r_s}{r_n - 0.5} \tag{10}$$

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 im betrachteten Szenario dieselbe Gewinnrate, gilt $l = 0$. Erzielt ein untersuchter Agent im betrachteten Szenario eine Gewinnrate von 50 %, so gilt $l = 1$. 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.

Bei den durchgeführten Messungen handelt es sich um Stichproben mit stochastisch bedingt variierenden Ergebnissen. Um dennoch eine Interpretierbarkeit der Messergebnisse zu gewährleisten, werden im Rahmen dieser Arbeit Konfidenzintervalle mit einem Konfidenzniveau von 95 % eingesetzt. Sie enthalten mit einer Wahrscheinlichkeit von 95 % den wahren Wert. Zur Ermittlung der Konfidenzintervalle für die Mittelwerte der Spieldauer wird das verbreitete Verfahren von Neyman angewandt ([16], S. 11 ff.; [23], S. 66 ff.). Die Konfidenzintervalle für die Gewinnraten werden mit Hilfe des Wilson-Verfahrens berechnet, das für die Berechnung von Anteilen in binomial verteilten Daten empfohlen wird [42] [6].

Um Robustheit gegenüber Unsicherheit bezüglich Aktionen zu untersuchen, wird die Unsicherheit bezüglich Aktionen so abgebildet, dass eine durch einen Agenten gewählte Aktion mit einer variierenden Wahrscheinlichkeit durch eine zufällige Aktion ersetzt wird. Beträgt diese Wahrscheinlichkeit 0 %, verhält sich der Agent ungestört. Bei 100 % spielt der Agent wie ein zufällig spielender Agent. Interessant zu beobachten ist es daher, wie sich der Verlauf des Gewinnratenverlustes zwischen 0 % und 100 % verhält. Der MCTS-Agent wird insofern eingeschränkt, dass er sich in seiner langfristigen Planung nicht darauf verlassen kann, dass er die Aktionen wählt, die er für am besten hält. Der RL-Agent hingegen in der Form, als dass er nicht immer die Aktionen durchführt, die sich im Training bewährt haben. Es gibt keine Hinweise darauf, dass ein Verfahren in diesem Szenario besser abschneidet als das andere. Durch die experimentelle Bewertung soll untersucht werden, ob dennoch Unterschiede auftreten.

Unsicherheit bezüglich Beobachtungen wird im Rahmen der Arbeit so auf Vier Gewinnt übertragen, dass die Agenten nicht den tatsächlichen Zustand des Spielfeldes beobachten können, sondern in den Beobachtungen im Vergleich zum tatsächlichen Zustand stets ein Spielstein oder mehrere Spielsteine hinzugefügt oder entfernt werden, sodass dadurch illegale Zustände entstehen, die im normalen Spiel nicht vorkommen können. Dabei ist zu beachten, dass sich bei bestimmten Modifikationen des Spielfeldzustands auch ändert, welche Aktionen möglich sind und welche nicht. Da es viele Probleme gibt, bei denen unabhängig vom Zustand immer alle Aktionen des Aktionsraums möglich sind, sollen die Messungen unabhängig von dieser Eigenschaft von Vier Gewinnt sein. Das bedeutet, dass bei den stattfindenden Modifikationen des Zustands nie der letzte Stein einer Spalte platziert, und auch kein Stein aus einer vollen Spalte entfernt werden darf. Die Anzahl der hinzuzufügenden oder zu entfernenden Steine wird bei den Messungen variiert.

Es sind dabei vor allem die Beobachtungen interessant, die keine legalen Zustände des Spiels darstellen. Denn bei gleich gut funktionierenden Agenten besteht die Erwartung, dass sie bei der Beobachtung eines falschen legalen Zustands, gleichwertig ungünstige Entscheidungen treffen. Werden fälschlicherweise illegale Zustände beobachtet, lässt sich die Hypothese bilden, dass dies vor allem für den RL-Agenten problematisch sein könnte. Dessen neuronales Netzwerk wurde lediglich auf Grundlage von legalen Zuständen trainiert, und kann nicht erfolgreich auf illegale Zustände generalisieren, da bei der Beobachtung von illegalen Zuständen Kombinationen von Neuronen aktiviert werden, dessen Gewichte im Training nicht entsprechend optimiert wurden. Das führt dazu, dass die Ausgabe des neuronalen Netzwerks stark verfälscht wird. Der MCTS-Agent hingegen wird bei illegalen Zuständen weiterhin versuchen, den bestmöglichen Zug zu ermitteln. Seine Leistungsfähigkeit wird durch die fehlerhaften Eingabewerte gestört, seine grundsätzliche Funktionsweise ist jedoch weiterhin gegeben.

Da bei Vier Gewinnt, wie in Kapitel 4.1 erwähnt, der Spieler, der den ersten Stein platziert, einen Vorteil hat, wechselt bei den Messungen, sofern nicht anders gekennzeichnet, zu Beginn jedes Spiels das Anzugsrecht.

Es ist zu beachten, dass die Agenten auf einem ähnlichen und gewissermaßen starken Level spielen. Denn durch ohnehin schwächer spielende Agenten werden in den verschiedenen Szenarien zur Untersuchung der Robustheit aussagekräftige Messungen von Änderungen in den Gewinnraten erschwert. Außerdem ist anzunehmen, dass stärkere Agenten Entscheidungen treffen, die längerfristig zum Erfolg führen. Dadurch kann den Beeinträchtigungen in den verschiedenen Szenarien besser standgehalten werden. 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.

6. Realisierung

In diesem Kapitel wird erläutert, wie die Ideen aus Kapitel 5 in die Praxis umgesetzt wurden und welche Abstriche dabei erforderlich waren. Dazu wird zunächst beschrieben, wie die Messumgebung, der MCTS-Agent und der PPO-Agent implementiert wurden. Anschließend wird dargestellt, wie die Szenarien zur Untersuchung der Robustheit der implementierten Agenten abgebildet werden. Der gesamte im Rahmen dieser Arbeit entstandene Quellcode ist auf GitHub unter `https://github.com/LeoHerrmann/connect_four_robustness/releases/tag/v1.0.0` einsehbar.

6.1. Messumgebung

Als Grundlage für die Messumgebung dient die Open-Source Python-Bibliothek PettingZoo der Farama Foundation, 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 Schnittstelle ähnelt dabei der Schnittstelle des Frameworks Gymnasium desselben Herausgebers, das Single-Agent-Umgebungen bereitstellt.

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 solche, die kooperative Probleme zum Benchmarking von MARL-Systemen oder auch rundenbasierte Spiele wie Vier Gewinnt abbilden.

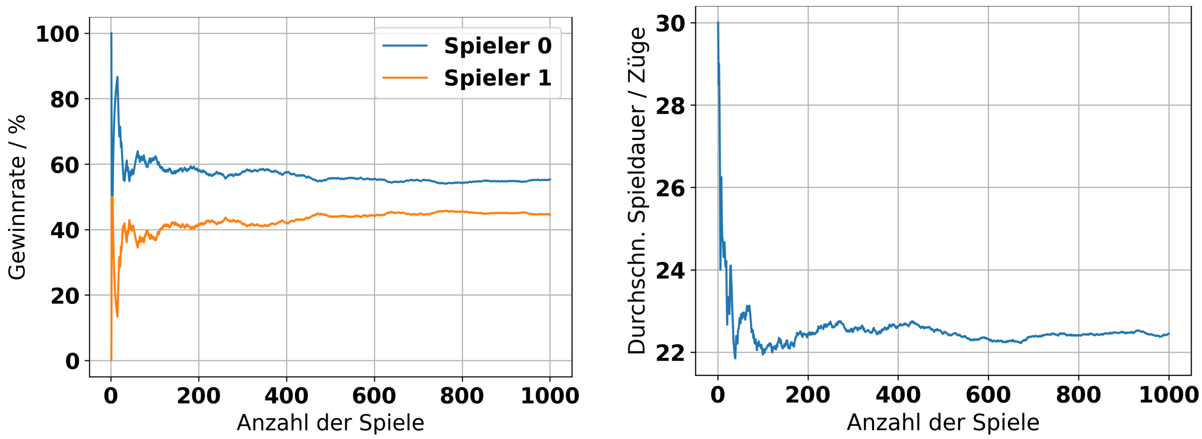
Die durch PettingZoo bereitgestellte Schnittstelle ermöglicht es, aus Sicht eines Agenten den aktuellen Zustand der Umgebung zu beobachten, die im aktuellen Zustand möglichen Aktionen und erhaltenen Belohnungen zu ermitteln und eine Aktion auszuwählen, die in der Umgebung durchgeführt werden soll.

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 solche, die ihre Entscheidungen rein zufällig oder unter Einbezug von menschlichen Eingaben treffen [39].

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 aufzuzeichnen. Die Open-Source-Eigenschaft von PettingZoo ermöglicht es, den Quellcode der Umgebung 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 durch Spieler 0 mit 55,3 % eine wesentlich höhere Gewinnrate erzielt wird als durch Spieler 1 mit 44,5 %. Der in Abbildung 3a gezeigte im Wesentlichen gegenläufige Verlauf der Gewinnrate über die Anzahl der durchgeführten Spiele zeigt, dass das Spielergebnis selten unentschieden ist. Eine signifikante Anzahl von nicht-entschiedenen Spielen würde sich in einem teils parallelen Verlauf der Gewinnrate äußern.

Um sicherzustellen, dass es sich bei der Differenz in der Gewinnrate nicht um einen stochastisch bedingten Messfehler handelt, wurde ein Binomialtest durchgeführt. Bei gleichen Chancen für beide Spieler wird eine Gewinnrate von jeweils 50 % erwartet. Der Test mit $n = 998$ entschiedenen Spielen, wovon $k = 553$ durch den ersten Spieler gewonnen wurden, liefert unter der Hypothese, dass die beiden Spieler gleiche Chancen haben, p-Werte von 0,0007, was unter dem Signifikanzniveau von $\alpha = 0,05$ liegt. Dadurch wird die Hypothese verworfen, was bedeutet, dass die Differenz der Gewinnraten trotz stochastischer Schwankungen signifikant ist.



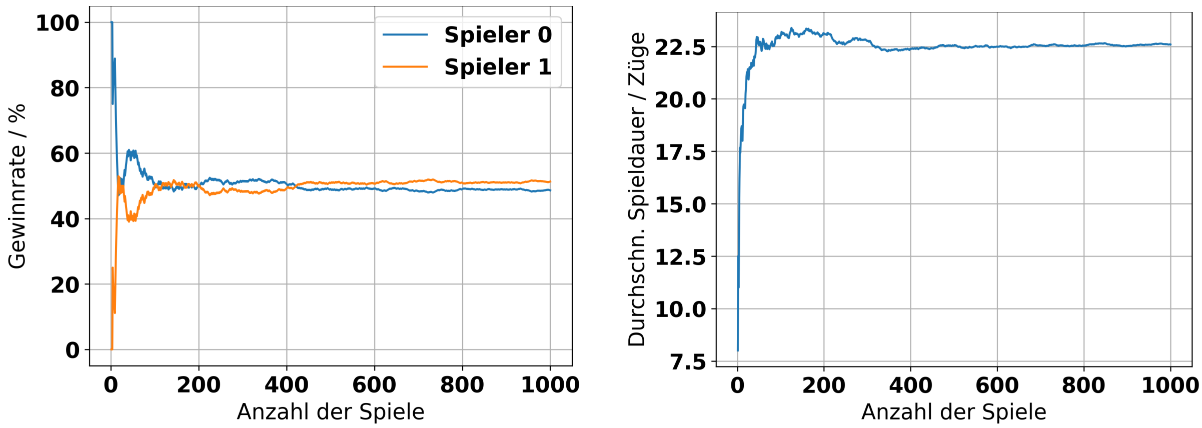
(a) Gewinnrate.

(b) Durchschnittliche Spieldauer.

Abb. 3: Gewinnrate und durchschnittliche Spieldauer bei konstanter Spielerreihenfolge.

Das lässt sich dadurch erklären, dass die Vier-Gewinnt-Umgebung so implementiert ist, dass Spieler 0 stets derjenige 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 erwähnt, dass bei Vier Gewinnt Spieler mit Anzugsrecht 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 Anzugsrecht abwechseln.

Wird das Anzugsrecht nach jedem Spiel gewechselt, weisen die Spiele wesentlich ausgeglichenere Ergebnisse auf. 48,6 % der Spiele werden durch Spieler 0 gewonnen und 51,3 % durch Spieler 1. Der zuvor bereits durchgeführte Binomialtest liefert mit den Werten $n = 999$, $k = 486$ einen p-Wert von 0,41, was deutlich über dem Signifikanzniveau von $\alpha = 0,05$ liegt, wodurch die Hypothese behalten wird. Die durchschnittliche Spieldauer bleibt dabei mit 22,45 Zügen (95 %-CI: 22,00 - 22,90) vor der Änderung gegenüber 22,60 Zügen (95 %-CI: 22,14 - 23,05) nach der Änderung abgesehen von den stochastisch bedingten Schwankungen nahezu unverändert.



(a) Gewinnrate.

(b) Durchschnittliche Spieldauer.

Abb. 4: Gewinnrate und durchschnittliche Spieldauer bei abwechselnder Spielerreihenfolge.

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

- `visitation_count: int` , `player_0_wins: int`, und `player_1_wins: 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) -> str | None`: 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 zurückgegeben wird der Gewinner, bzw. None, wenn das Spiel nicht entschieden werden konnte.

- `backpropagate(selected_node: MctsNode, winner: str | None) -> None`: Von ausgewählten Knoten wird das Attribut `visitation_count` erhöht und ggf.

wird `player_0_wins` oder `player_1_wins` hochgezählt. 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, wodurch Aufwand in der Implementierung gespart wird. 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(..., 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 [39]. 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 Methode der Umgebung `reset(..., options: dict)` 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.

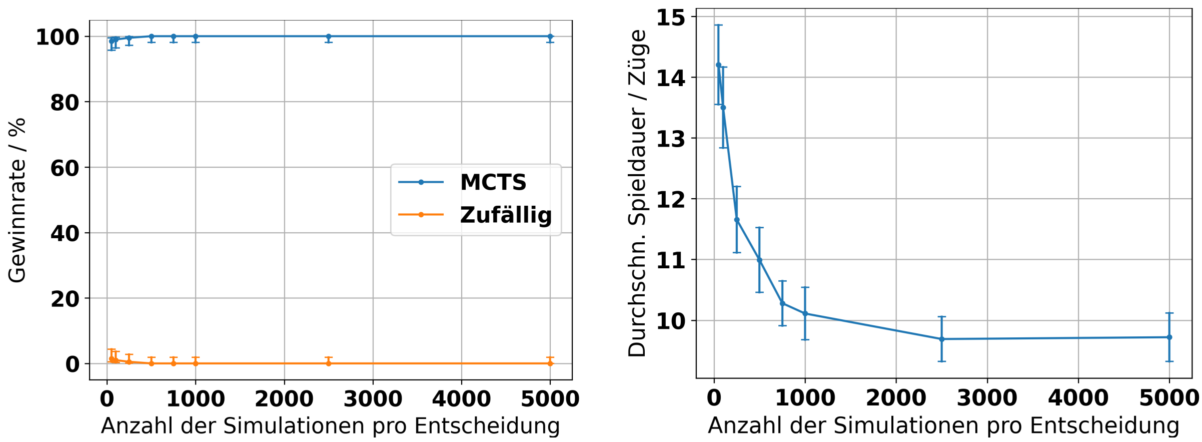
6.2.1. Zeitlicher Aufwand von Entscheidungen des MCTS-Agenten

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 damit 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. Wenn keine andere Aussage darüber getroffen wird, ist der MCTS-Agent im Rahmen dieser Arbeit so konfiguriert, dass 5000 Simulationen pro Entscheidungen durchgeführt werden.

6.2.2. Quantitative Untersuchung

Um einen Eindruck von der Strategie 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.

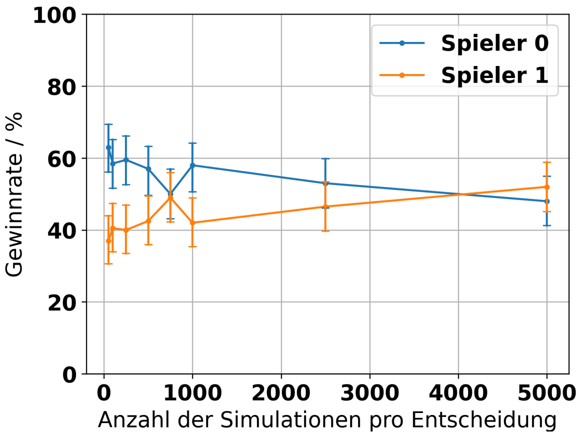


(a) Gewinnrate.

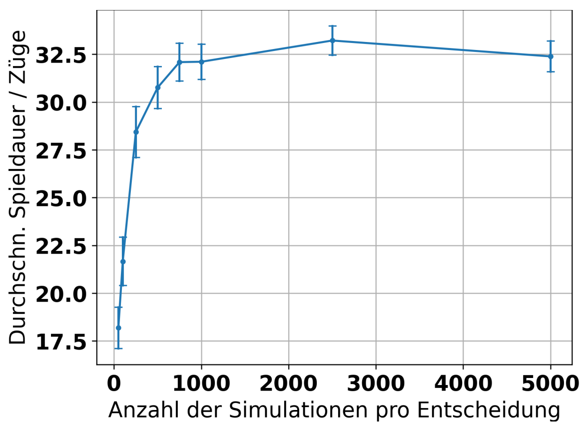
(b) Durchschnittliche Spieldauer.

Abb. 5: 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 % (95 %-CI: 95,7 - 99,5) und steigt bis 1000 Simulationen auf 100 % (95 %-CI: 98,1 - 100,0), welche der Agent auch bei 2500 und 5000 Simulationen hält. Die mittlere Spieldauer startet bei 50 Simulationen mit 14.21 Zügen (95 %-CI: 13,55 - 14,86) und flacht bei 2500 Simulationen mit 9,69 Zügen (95 %-CI: 9,32 - 10,06) ab. Bei dem leichten Anstieg der Spieldauer bei 5000 Simulationen auf 9,72 Züge (95 %-CI: 9,32 - 10,12) steigt, liegt durch die starke Überschneidung der Konfidenzintervalle eine stochastisch bedingte Messungenauigkeit nahe. Die kürzeste mögliche durchschnittliche Spieldauer beträgt 7.5 Züge, 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. 6: Gewinnrate und durchschnittliche Spieldauer in Abhängigkeit von der Anzahl der Simulationen pro Entscheidung beim Spiel von zwei MCTS-Agenten gegeneinander.

Spielen zwei MCTS-Agenten mit konstantem Anzugsrecht gegeneinander, hat von 50 bis 250 durchgeführten Simulationen pro Entscheidung Spieler 0 mit Gewinnraten zwischen 63,0 % (95 %-CI: 56,1 - 69,4) und 58,5 % (95 %-CI: 51,6 - 65,1) einen signifikanten Vorteil gegenüber seines Gegenspielers, was bedeutet, dass der Vorteil, das Anzugsrecht zu besitzen, ausgenutzt werden kann. Ab 500 Simulationen liegen die Messwerte nahe bei einander und die Konfidenzintervalle sind so groß, dass aus den Messungen nicht hervorgeht, ob einer der beiden Spieler signifikant einen signifikanten Vorteil hat. Bei 750 Simulationen wurden mit 50,0 % (95 %-CI: 43,1 - 56,9) für Spieler 0 und 49,0 % (95 %-CI: 42,2 - 55,9) für Spieler 1 nahezu identische Werte gemessen. Bei 5000 Simulationen wurde sogar eine mit 48,0 % (95 %-CI: 41,2 - 54,9) niedrigere Gewinnrate für Spieler 0 als Spieler 1 mit 52,0 % (95 %-CI: 45,1 - 55,8) aufgezeichnet. Aus den Konfidenzintervallen geht jedoch nicht hervor, ob sich die tatsächlichen Werte auch so verhalten.

Dieser Verlauf in der Hinsicht überraschend, als dass der MCTS-Algorithmus bei Vier Gewinnt mit steigender Simulationsanzahl zu perfektem Spielverhalten konvergieren sollte, wodurch der anziehende Spieler mit zunehmender Anzahl von durchgeführten Simulationen mehr Spiele für sich entscheiden sollte. Eine Erklärung dafür, dass bei den durchgeführten Messungen ein solches Verhalten nicht beobachtet wird, ist, dass dafür der MCTS-Agent auch mit 5000 Simulationen pro Entscheidung vom perfekten Spielverhalten noch zu weit entfernt ist. Bis 5000 Simulationen erhöht sich die Fähigkeit des MCTS-Agenten zu verteidigen schneller als seine Fähigkeit anzugreifen, wodurch beide Spieler jeweils etwa die Hälfte der Spiele gewinnen.

Die mittlere Spieldauer steigt von 50 Simulationen bis 750 Simulationen stetig von 18,19 (95 %-CI: 17,11 - 19,27) auf 32,11 % (95 %-CI: 31,10 - 33,03). Der Verlauf flacht dabei ab, sodass für die darauf folgenden Werte keine Aussagen darüber getroffen werden können, ob sie weiterhin steigen oder nicht. Die zunächst steigende Spieldauer deutet darauf hin, dass die MCTS-Agenten längerfristige Strategien entwickeln, sodass komplexere Spielsituationen zustande kommen.

6.2.3. 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 250 Simulationen pro Entscheidung durchzuführen, lässt sich beobachten, dass bei drei bereits in einer Kette platzierten Spielsteinen, stets auch der vierten Spielstein platziert wird, um zu gewinnen, sofern dies möglich ist. Wenn hingegen durch seinen Gegenspieler drei Steine in einer Kette platziert wurden, wird durch den MCTS-Agenten häufig versäumt, 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 1.000 Simulationen pro Entscheidung wird durch den MCTS-Agenten sein erster Stein meistens im untersten Feld der mittleren Spalte platziert, sofern es frei ist, was laut Allis auch der stärkste Anfangszug ist [3]. 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. Werden durch den Gegenspieler beispielsweise seine Steine zu Beginn des Spiels so platziert,

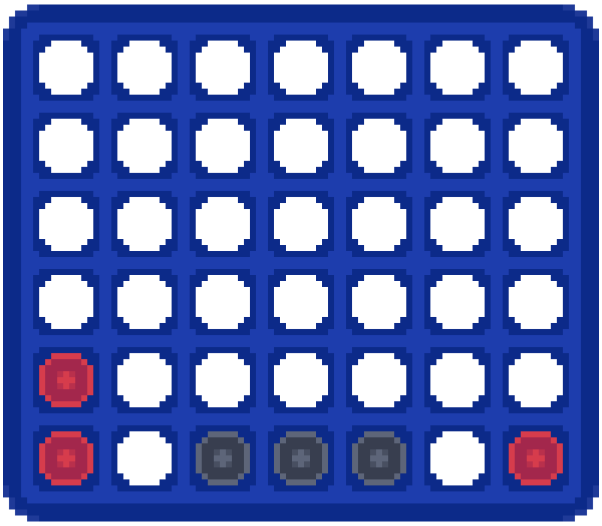


Abb. 7: Zwickmühle, die durch den MCTS-Agenten (blau) hervorgerufen wurde. Grafikdesign aus [39].

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 wie in Abbildung 7 dargestellte Zwickmühle, indem 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 250 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 wird der erste Spielstein des MCTS-Agenten, wenn möglich, stets in das unterste Feld der mittlere Spalte platziert. 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.

6.3. PPO-Agent

Die Implementierung des PPO-Agenten erfolgte im Rahmen dieser Arbeit mit Hilfe des Frameworks Stable Baselines3. Es wurde zunächst versucht, mit der Trainingsstrategie von Zhong et al. einen PPO-Agenten zu entwickeln, der mit dem MCTS-Agenten mithalten kann. Dies ist jedoch mangels zeitlicher Ressourcen nicht gelungen. Aus diesem Grund wurde auf eine einfachere Trainingsstrategie umgestellt, die darin besteht, das PPO-Modell gegen einen zufällig spielenden Agenten zu trainieren.

6.3.1. Einführung in Stable Baselines3

Das PPO-Verfahren wird in dieser Arbeit nicht von Grund auf implementiert. Stattdessen wird auf eine Implementierung des Verfahrens aus dem Framework Stable Baselines3 zurückgegriffen. Dieses Framework enthält Implementierungen von verschiedenen RL-Verfahren, die auf den Referenzimplementierungen von OpenAI aufbauen. Das Framework bietet eine Schnittstelle, die es ermöglicht, die implementierten RL-Verfahren auf Gymnasium-Umgebungen anzuwenden. Aufgrund der Ähnlichkeit der Schnittstellen von Gymnasium und PettingZoo lässt sich Stable Baselines3 auch auf PettingZoo-Umgebungen anwenden [31] [39].

Die grundlegende Vorgehensweise bei der Verwendung des Frameworks besteht darin, aus einer Klasse, die ein ausgewähltes RL-Verfahren implementiert (z.B. PPO), ein Objekt zu erzeugen, das ein zu trainierendes RL-Modell repräsentiert. Diesem Modell wird dabei eine Gymnasium- oder PettingZoo-Umgebung zugeordnet, in der das Modell trainiert werden soll. Anschließend kann das Training über die Methode des Modells `learn(total_timesteps: int)` starten. Dabei beschreibt `total_timesteps` die Anzahl von Schritten, die in der Trainingsumgebung durchgeführt werden sollen [31].

Bei jedem Schritt wird die Step-Methode der Umgebung aufgerufen, die die Aktion eines Agenten entgegennimmt und den Zustand der Umgebung entsprechend verändert. Wurde ein Endzustand erreicht, wird die Reset-Methode der Umgebung aufgerufen, die die Umgebung wieder in den Ursprungszustand versetzt. Über optionale Funktionsparameter ist es möglich, Hyperparameter des Trainings anzupassen. Dazu gehören beispielsweise die Lernrate, die Architektur der neuronalen Netzwerke des RL-Modells, oder die Anzahl von durchzuführenden Trainingsschritten bis die Parameter der Netzwerke auf Grundlage der zuletzt gesammelten Erfahrungen aktualisiert werden sollen [31].

Wrapper Wird einem Modell eine PettingZoo-Umgebung direkt zugeordnet und anschließend ein Training gestartet, so wird das Modell die Züge für alle Agenten der

Umgebung durchführen und im Laufe des Trainings seine Parameter auf Grundlage der Belohnungen jedes Agenten anpassen [39]. Dadurch wird das in MARL allgegenwärtige Moving-Target-Problem verstärkt. Wenn alle Agenten der Umgebung ihre Parameter gleichzeitig aktualisieren, versucht jeder Agent, sich an das Verhalten der anderen sich ändernden Agenten anzupassen. Das kann zyklisches und instabiles Lernverhalten hervorrufen ([1], S. 12). Aus diesem Grund ist es gängige Praxis bei MARL, nur einen Agenten auf einmal zu trainieren und die anderen Agenten stationär zu halten ([1], S. 300; [44]).

Um dieses Verhalten abbilden zu können, wird für die Implementierung der Trainingsstrategien die PettingZoo-Umgebung von Vier Gewinnt von einem Wrapper umschlossen, der nach außen dieselbe Schnittstelle bereitstellt, wie die Umgebung selbst. Die Vier-Gewinnt-Umgebung wird nicht direkt dem Modell zugeordnet, sondern der Wrapper, der diese Umgebung enthält. Die Step-Methode wurde über den Wrapper erweitert, nicht nur den Zug auszuführen, den das trainierende Modell gewählt hat, sondern auch den Zug, der durch ein stationäres Regelwerk gewählt wird. Damit das Modell trotzdem lernen kann, nicht nur als erster sondern auch als zweiter Spieler zu spielen, kann in der Reset-Methode Logik eingebunden werden, um einen Zug für einen stationären Agenten durchzuführen, und damit das Anzugsrecht zu wechseln.

Action Masking Neben den bereits in Kapitel 4.3.3 genannten Vorteilen besitzt das Verfahren PPO auch den Vorteil, dass Stable Baselines3 die Implementierung `MaskablePPO` enthält, die Action Masking unterstützt. Für A2C existiert derzeit keine solche Implementierung [31]. Action Masking sorgt dafür, dass das Modell im Training und in der Anwendung daran gehindert wird, Entscheidungen zu treffen, die die Umgebung nicht erlaubt [20]. Im Vier Gewinnt bedeutet das beispielsweise, einen Spielstein in eine Spalte zu platzieren, in der bereits alle Felder belegt sind. Die PettingZoo-Umgebung ist zwar darauf ausgelegt, bei illegalen Aktionen das Spiel zu beenden und den verantwortlichen Agenten mit einer entsprechenden negativen Belohnung zu bestrafen [39]. Es wurde jedoch gezeigt, dass der Einsatz von Action Masking effektiver ist als sich auf die Bestrafung über die Umgebung zu verlassen [20].

Damit Action Masking in der Implementierung der Trainingsumgebung funktioniert, muss die dem Modell zugeordnete Umgebung eine Methode `action_mask()` implementieren, die Auskunft darüber gibt, welche Aktionen möglich und welche nicht möglich sind. Das ist in den PettingZoo-Umgebungen standardmäßig nicht der Fall. Diese Funktionalität wurde daher dem oben genannten Wrapper hinzugefügt.

6.3.2. Implementierte Trainingsverfahren

Im Rahmen der Arbeit wurden zwei verschiedene Trainingsverfahren implementiert. Ersteres konnte aufgrund mangelnder Zeit nicht erfolgreich umgesetzt werden. Aus diesem Grund wurde auf eine einfachere Trainingsstrategie umgestellt.

Trainingsverfahren nach Zhong et al. Zunächst wurde versucht, die in [44] von Zhong et al. herausgearbeitete Trainingsstrategie für Zwei-Spieler-Nullsummenspiele zu implementieren. Im Zuge dieser Strategie werden mehrere Modelle auf einmal trainiert. Die Modelle sind dabei in n 2-Tupel organisiert. Die jeweils ersten Modelle der Tupel spielen stets als erster Spieler und die jeweils zweiten Modelle der Tupel spielen stets als zweiter Spieler. Folgende zwei Schritte werden für N Iterationen wiederholt:

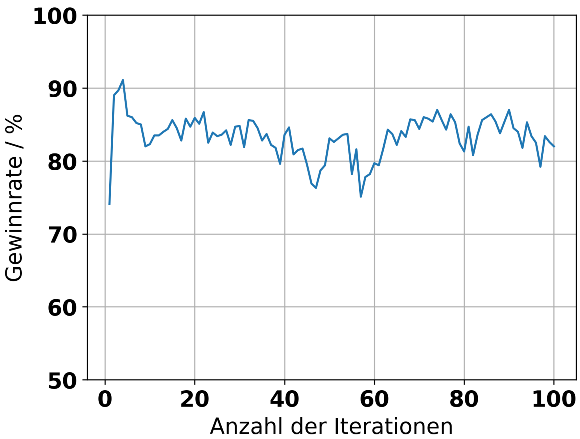
- Jedes erste Modell aller Tupel wird gegen jedes zweite Modell aller Tupel durch wiederholte Spiele evaluiert. Dabei werden die Gewinnraten aufgezeichnet.
- Anschließend werden alle Modelle für l Trainingsschritte jeweils gegen das Modell trainiert, gegen das die niedrigste Gewinnrate erzielt wurde.

In der genannten Arbeit wurde für verschiedene exemplarische Anwendungsfälle gezeigt, dass diese Strategie zu leistungsfähigeren Modellen führt als andere weit verbreitete Self-Play-Strategien. Diese anderen Strategien bestehen häufig darin, ein Modell wiederholt für eine bestimmte Anzahl von Trainingsschritten gegen eine ältere und stationäre Version von sich selbst zu trainieren und zwischenzuspeichern. Bei jeder Wiederholung wird der Gegenspieler durch das neuste, beste oder ein zufälliges zwischengespeichertes Modell ersetzt.

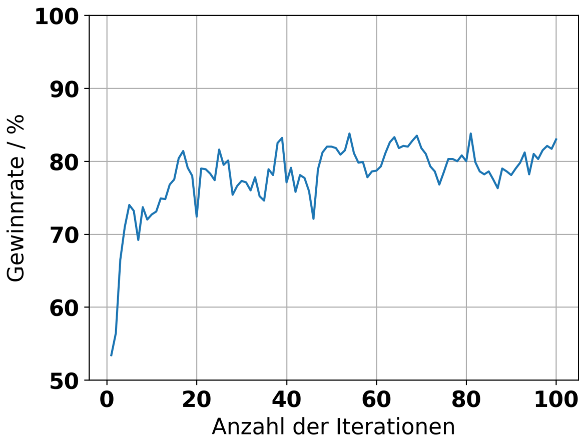
Ein Anwendungsfall, an dem in [44] die überlegene Leistungsfähigkeit des Verfahrens gezeigte wurde, ist die 9 x 9 Variante des Spiel Gomoku. Das Konzept des Spiels ist dabei ähnlich wie das von Vier Gewinnt. Das Spielfeld ist jedoch nicht 7 x 6, sondern 9 x 9 Felder groß, zum Gewinnen müssen fünf und nicht vier Spielsteine in einer Kette platziert werden, und die Spielsteine fallen nicht in das unterste Feld einer Spalte, sondern können auf beiden Achsen beliebig positioniert werden. Da Vier Gewinnt damit einen kleineren Zustands- und Aktionsraum besitzt, liegt nahe, dass die in [44] vorgestellte Trainingsstrategie auch für Vier Gewinnt funktioniert.

Im Rahmen dieser Arbeit wurde das Verfahren mit den Lernraten (siehe Formel 2) $\alpha_0 = 0,001$, $\alpha_1 = 0,003$ (Standardwert aus Stable Baselines3), $\alpha_2 = 0,0001$, $\alpha_3 = 0,00003$ und $\alpha_4 = 0,00001$ durchgeführt. In [44] werden nach der Evaluation jeweils so viele Trainingsschritte durchgeführt, bis $l = 10$ Parameteraktualisierungen der RL-Modelle hervorgerufen wurden. Nach dem voreingestellten Hyperparametern der Implementierung von PPO in Stable Baselines3 werden die Parameter der neuronalen Netzwerke nach 2048 Trainingsschritten aktualisiert. Bei der Anwendung des Verfahrens in dieser Arbeit werden daher 20480 Schritte durchgeführt, um die gleiche Anzahl von $l = 10$ Parameteraktualisierungen hervorzurufen wie in [44]. Die Anzahl der Iterationen N beträgt in [44] je nach Anwendungsfall 40 bzw. 50. Um die Wahrscheinlichkeit zu reduzieren, dass das Training aufgrund von zu kurzer Dauer fehlschlägt, wurde in dieser Arbeit $N = 100$ gewählt. Die Architektur der neuronalen Netzwerke und alle anderen Hyperparameter werden bei den Standardwerten von Stable Baselines3 belassen.

Während der Trainings mit den verschiedenen Lernraten wurde nach allen N Iterationen jeweils der durchschnittliche Policy-Gradient-Verlust (siehe Formel 9) und Werteverlust (siehe Formel 7) der Iteration, und die über 500 Spiele gemittelte Gewinnrate und Spieldauer von zufälligen Modelltupeln gegen einen zufällig spielenden Agenten aufgezeichnet. Der Policy-Gradient- und Werteverlust werden jeweils als Maß dafür verwendet, wie stark die Gewichte des Actor- bzw. Ciritc-Netzwerks angepasst werden. Konvergieren die Verluste gegen Ende des Trainings in der Nähe von 0, deutet das darauf hin, dass das Modell sein Optimierungspotenzial ausgeschöpft hat und kein neues Verhalten mehr lernt. Ergänzend zu den hier präsentierten Trainingsmetriken mit ausgewählten Gewinnraten finden sich im Anhang A.1 zusätzliche Diagramme zu den Trainings mit allen weiteren Gewinnraten.



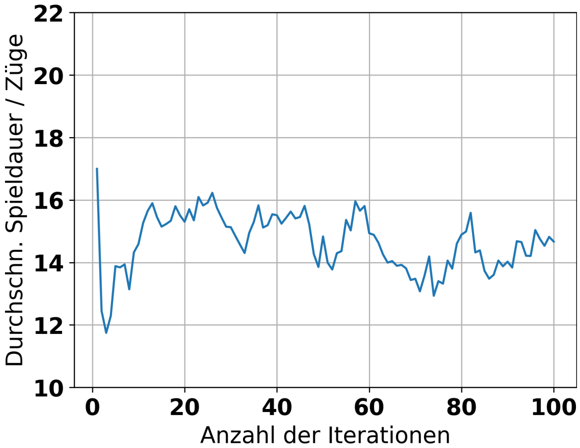
(a) Verlauf der Gewinnrate mit Lernrate α_0.



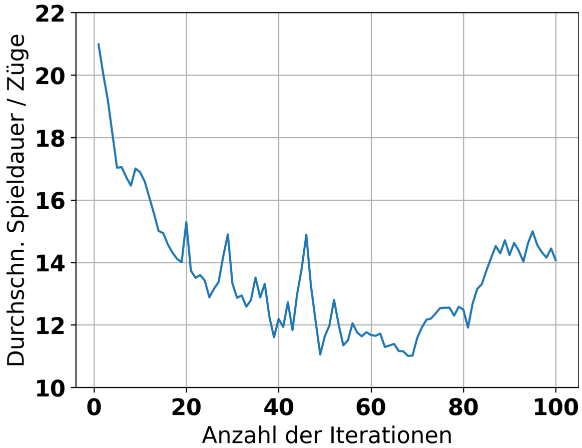
(b) Verlauf der Gewinnrate mit Lernrate α_4.

Abb. 8: Verlauf der Gewinnrate während des Trainings nach Zhong et al. mit den Lernraten α_0 und α_4

Die Gewinnrate steigt bei den Trainings mit den kleineren Lernraten α_2, α_3 und α_4,



(a) Verlauf der Spieldauer mit Lernrate α_0.



(b) Verlauf der Spieldauer mit Lernrate α_4.

Abb. 9: Verlauf der Spieldauer während des Trainings nach Zhong et al. mit den Lernraten α_0 und α_4.

innerhalb der ersten 20 Iterationen und fluktuiert von dort bis zum Ende des jeweiligen Trainings auf einem konstanten Niveau. Dieses Niveau beträgt bei der Lernrate α_2 etwa 85 % und bei α_3 und α_4 etwa 80 %. Mit den Lernraten α_0 und α_1 steigt die Gewinnrate nach etwa 5 Iterationen auf knapp über 90 % und fällt danach auf etwa 80 %.

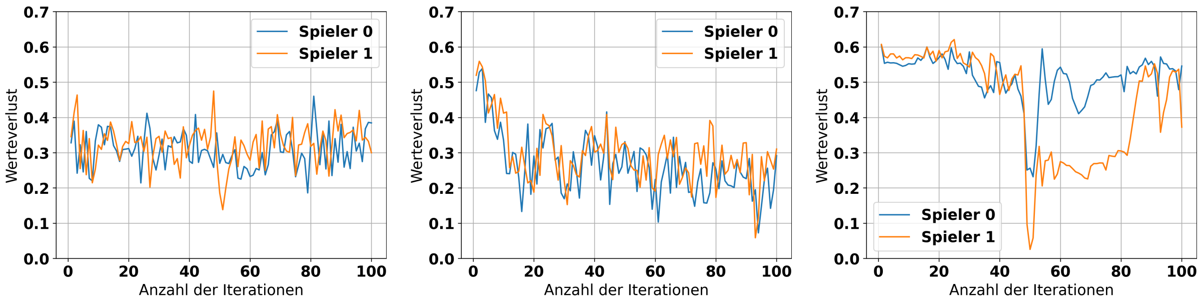
Die durchschnittliche Spieldauer verläuft bei allen Trainings unabhängig von der Lernrate ähnlich. Die Spieldauer sinkt im Verlauf des Trainings auf knapp unter 12 Züge und steigt wieder auf zwischen 14 und 18 Züge. Der Punkt mit der niedrigsten Spieldauer tritt dabei mit kleineren Lernraten tendenziell erst nach mehr Iterationen auf.

Eine sinkende Gewinnrate bzw. eine konstante Gewinnrate mit steigender Spieldauer deuten darauf hin, dass Verhalten, das stark gegen einen zufällig spielenden Agenten ist, im Laufe des Trainings erlernt und anschließend wieder verlernt wird. Die Spielstärke des MCTS-Agenten, der gegen zufällig spielende Gegenspieler eine Gewinnrate von 100 % und eine Spieldauer von 9 Zügen erzielt, wird im Laufe des Trainings an keiner Stelle erreicht.

Um die Ursache für das stagnierende bzw. rückläufige Lernverhalten zu ergründen, wurden der Policy-Gradient-Verlust und der Werteverlust näher betrachtet.

Bei den großen Lernraten α_0 und α_1, fluktuiert der Werteverlust weitgehend auf einem konstanten Niveau zwischen 0,2 und 0,4. Dieses Verhalten kann auf eine zu große Lernrate zurückzuführen sein, die dazu führt, dass die Parameter um ein Verlustminimum herum oszillieren. Bei der mittleren Lernrate α_2 lässt sich beobachten, dass der Werteverlust vor einer konstanten Fluktuation in den ersten 20 Iterationen von 0,5 auf etwa 0,25 sinkt. Das kann ein Hinweis dafür sein, dass sich die Parameter während der ersten

Iterationen auf ein lokales Minimum zu bewegen. Bei den kleineren Lernraten α_3 und α_4 werden kleinere Fluktuationen, dafür ein unregelmäßiges Verhalten mit Einbrüchen der Werteverluste über mehrere aufeinanderfolgende Iterationen beobachtet. Die Einbrüche erfolgen dabei teilweise für beide Spieler unterschiedlich. Plötzlich einbrechender und wieder steigender Verlust können darauf hinweisen, dass die trainierenden Modelle neues Verhalten erlernen und auf die Verhaltensänderungen voneinander reagieren.



(a) Verlauf des Werteverlustes mit Lernrate α_0.

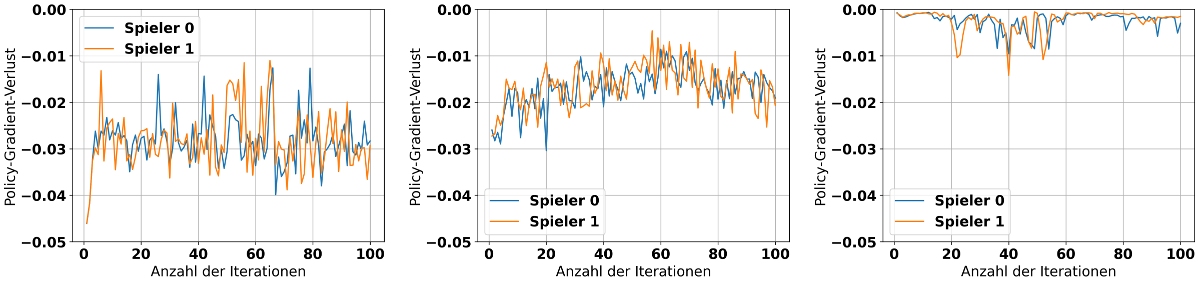
(b) Verlauf des Werteverlustes mit Lernrate α_2.

(c) Verlauf des Werteverlustes mit Lernrate α_4.

Abb. 10: Verlauf des Werteverlustes während des Trainings nach Zhong et al. mit den Lernraten α_0 und α_4.

Der Policy-Gradient-Verlust fluktuiert bei den Trainings mit den Lernraten α_0 und α_2 über den gesamten Verlauf um einen konstanten Wert von -0,025 bzw. -0,015. Die Fluktuation um einen konstanten Wert deutet dabei auf eine zu große Lernrate hin. Bei der zweitgrößten Lernrate α_1 wird eine Fluktuation konstanten Ausmaßes beobachtet, wobei der Wert um den fluktuiert wird, in den ersten 60 Iterationen von etwa -0,025 auf -0,010 steigt und danach auf -0,020 sinkt. Ähnlich wie beim Werteverlust bei der mittleren Lernrate α_2 kann der sich zunächst an Null annähernde Verlust als eine positive Entwicklung interpretiert werden. Je kleiner die Lernrate ist, desto näher bewegen sich die Werte für den Policy-Gradient-Verlust bei Null und desto kleiner werden über den größten Teil des Verlaufs die Fluktuationen. Dafür treten im Verhältnis zur sinkenden Fluktuation häufiger einzelne größere Ausreißer auf. Dies kann wie der Werteverlust mit plötzlichen, länger andauernden Einbrüchen auf neu erlerntes Verhalten hindeuten.

Leider wurden keine Korrelationen zwischen dem Verhalten des Policy-Gradient- oder Werteverlustes und der Leistung der trainierten Modell hinsichtlich Gewinnrate oder Spieldauer gefunden. Es lassen sich daher auch keine Aussagen darüber treffen, ob fluktuierende Verlust-Verläufe, die sich über die Iterationen der 0 annähern oder weniger fluktuierende Verläufe mit plötzlichen Einbrüchen oder Ausreißern erstrebenswerter sind. Um die Ursachen für das stagnierende Training weiter zu ergründen, sind die Analyse



(a) Verlauf des Policy-Gradient-Verlustes mit Lernrate α_0.

(b) Verlauf des Policy-Gradient-Verlustes mit Lernrate α_1.

(c) Verlauf des Policy-Gradient-Verlustes mit Lernrate α_4.

Abb. 11: Verlauf des Policy-Gradient-Verlustes während des Trainings nach Zhong et al. mit den Lernraten α_0, α_1 und α_4.

von weiteren Trainingsmetriken und Experimente mit weiteren Hyperparametern erforderlich.

Anstatt nur den Verlauf der mittleren Verluste der einzelnen Iterationen, und die Leistung gegen den zufälligen Agenten am Ende der Iterationen zu beobachten, könnte es helfen, mehr über den Verlauf der Verluste und Belohnung innerhalb der Iterationen zu erfahren. Dies wäre möglich, indem die Verluste und die mittlere in den Episoden erhaltene Belohnung über die einzelnen Trainingsschritte aufgezeichnet werden. Eine weitere hilfreiche Trainingsmetrik könnte die Entropie der Wahrscheinlichkeitsverteilungen über die Aktionen sein, die durch das Regelwerk generiert werden. Eine hohe Entropie bedeutet, dass alle Aktionen mit ähnlich großer Wahrscheinlichkeit gewählt werden, während eine niedrige Entropie bedeutet, dass wenige verschiedene Aktionen mit hoher Wahrscheinlichkeit gewählt werden. Sie ist ein Maß dafür, wie sicher sich das Regelwerk in seinen Entscheidungen ist und sollte über den Verlauf des Trainings sinken [25].

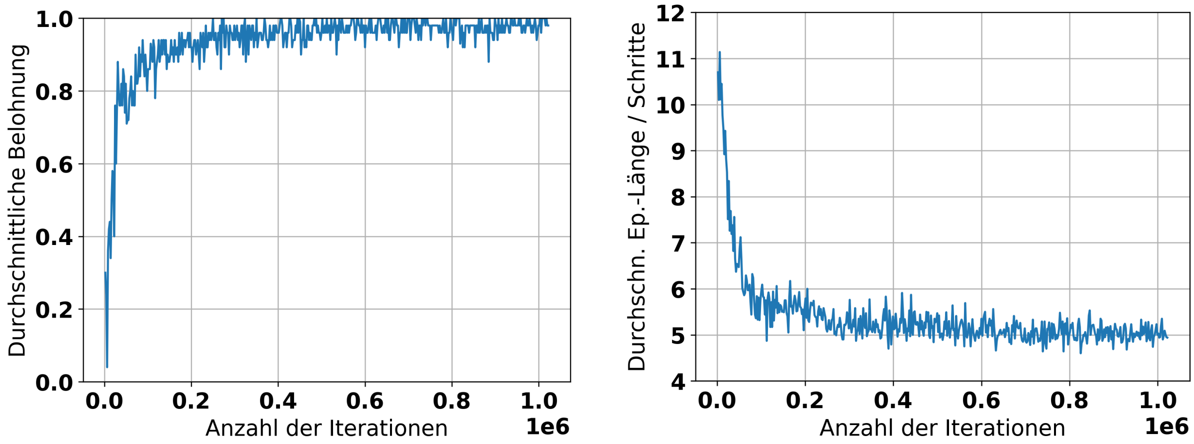
Um das RL-Modell für Gomoku zu trainieren, wurden in [44] neuronale Netzwerke mit Faltungsmatritzen in den ersten Netzwerkschichten eingesetzt, die bei der Erkennung von Mustern in Eingabedaten unterstützen. Trotz der niedrigeren Komplexität ist dies bei Vier Gewinnt möglicherweise ebenfalls notwendig, um starke Agenten zu trainieren. In [43] wurde mit einem mit Faltungsmatritzen ausgestattetes PPO-Modell erfolgreich trainiert, sodass es gegen einen MCTS-Agenten standhalten konnte, der 2000 Simulationen pro Zug durchführt. In [40] wurden N-Tupel-Netzwerke eingesetzt, welche ebenfalls Mechanismen zur Mustererkennung enthalten. Dabei konnte ein RL-Modell trainiert werden, das als erster Spieler mit einer Wahrscheinlichkeit von 90 % gegen einen perfekt spielenden Minimax-Agenten gewinnt.

In [38] wurden keine Mechanismen zur Mustererkennung eingesetzt und der darin trainierte Q-Learning-Agent erzielt in Vier Gewinnt eine Gewinnrate von lediglich 80 %. Das muss jedoch nicht an der fehlenden Mustererkennung liegen. Wie der Autor selbst erwähnt, liegt das vermutlich an der fehlenden Fähigkeit des Q-Learning-Verfahrens zu verallgemeinern. Aus der Literatur geht nicht hervor, dass Faltungsmatritzen erforderlich sind, um ein starkes PPO-Modell für Vier Gewinnt zu trainieren, aber es liegt nahe, dass deren Einsatz gewinnbringend sein könnte.

Training gegen einen zufällig spielenden Agenten Aufgrund begrenzter zeitlicher Ressourcen war es nicht möglich, weitere Trainingsmetriken zu analysieren oder zusätzliche Experimente mit anderen Hyperparametern wie Netzwerkarchitekturen mit Faltungsmatritzen durchzuführen. Daher wurde die Entscheidung getroffen, auf eine einfachere Trainingsstrategie umzustellen. Sie besteht darin, ein RL-Modell gegen einen zufällig spielenden Agenten zu trainieren. Auch hier wird als RL-Verfahren PPO eingesetzt. Es wurden verschiedene Trainings mit den Lernraten $\alpha_0 = 0.003$, $\alpha_1 = 0.001$, $\alpha_2 = 0.0005$ und $\alpha_3 = 0.0001$ durchgeführt. Jedes Training war 1.024.000 Trainingsschritte lang. Alle anderen Parameter, wie die Architektur der Netzwerke oder die Anzahl der Schritte, nach denen die Gewichte des Modells angepasst werden, wurden bei den durch Stable Baselines3 voreingestellten Parameter belassen.

Beobachtet wurden dabei die durchschnittlich in den vergangenen 100 Episoden erhaltene Belohnung und die durchschnittliche Länge der Episoden. Eine Episode bezeichnet in diesem Fall ein Spiel, das im Training vom Anfang bis zum Ende durchgespielt wurde. Das trainierende Modell erhält von der PettingZoo-Umgebung eine Belohnung von 1 für Züge, die zum Gewinn geführt haben, -1 für Züge, die zum Verlust geführt haben und 0 für alle anderen Züge [39]. Die Episodenlänge bezeichnet die Anzahl der im Training durchgeführten Schritte, bis eine Episode endet. Wie in 6.3.1 erwähnt, besteht ein Schritt aus einer Aktion des trainierenden Modells und einer Aktion seines Gegenspielers, dem zufällig spielendem Agenten. Da die Spieler im Training nach jedem Spiel das Anzugsrecht wechseln, beträgt hier das Minimum 3,5, da mindestens 7 Steine platziert sein müssen, damit ein Spieler seine Spielteine in einer Viererkette platziert haben kann und das Spiel endet. Das Maximum ist die Hälfte Anzahl der Spielfelder, also 21. Kürzere Episoden deuten darauf hin, dass der trainierende Agent im Training stärker ist.

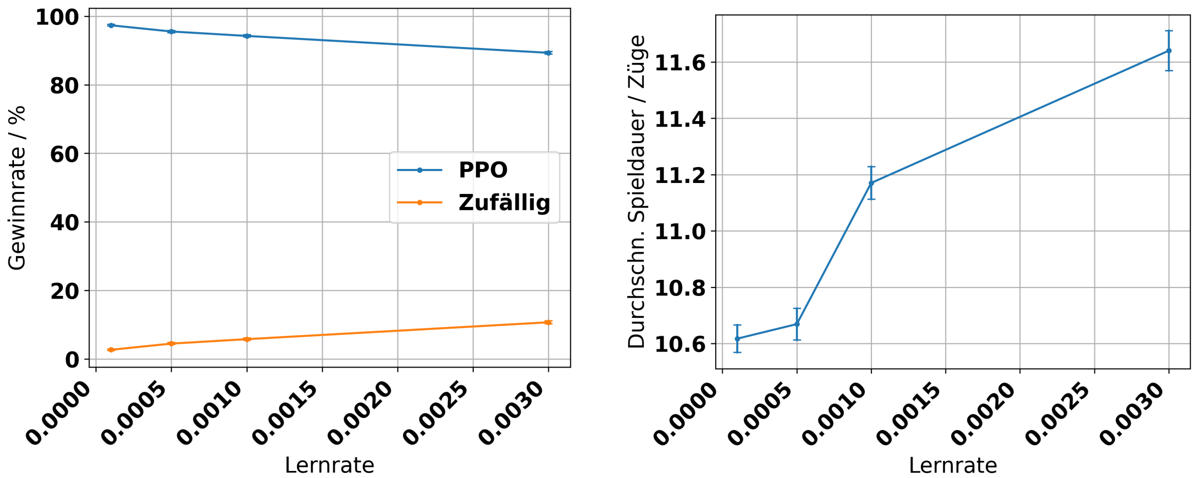
Unabhängig von der Lernrate ist bei den betrachteten Trainingsmetriken nach wenigen hunderttausend Trainingsschritten ein erstrebenswertes Konvergenzverhalten zu beobachten. Die durchschnittliche in den Episoden erhaltene Belohnung konvergiert zu 1, was bedeutet, dass das Modell im Training die meisten Spiele gewinnt. Die durch-



(a) Durchschnittliche Belohnung. (b) Durchschnittliche Episodenlänge.

Abb. 12: Verlauf der durchschnittlichen Belohnung und Episodenlänge während des Trainings gegen einen zufällig spielenden Agenten mit Lernrate α_3.

schnittliche Episodenlänge konvergiert bei etwa 4,5 bis 5, was 9 bis 10 Zügen entspricht. Ergänzend zu Abbildungen 12a und 12b befinden sich Diagramme zu den Trainings mit den Lernraten α_0 bis α_2 im Anhang A.2.



(a) Gewinnrate. (b) Durchschnittliche Spieldauer.

Abb. 13: Gewinnrate und durchschnittliche Spieldauer in Abhängigkeit von der Lernrate beim Training eines PPO-Modells gegen einen zufällig spielenden Agenten.

Um die Leistung der trainierten Modelle gegen den zufällig spielenden Agenten zu evaluieren, wurden mit jedem Modell 20.000 Spiele durchgeführt und die Gewinnrate und Spieldauer wurde aufgezeichnet. Mit steigender Lernrate sinkt die durch das trainierte Modell erzielte Gewinnrate stetig von 97,3 % (95 %-CI: 97,1 - 97,6) auf 89,3 % (95

%-CI: 88,9 - 89,7) und die durchschnittliche Spieldauer steigt von 10,62 Zügen (95 %-CI: 10,57 - 10,67) auf 11,64 Züge (95 %-CI: 11,57 - 11,71). Durch das Training mit kleineren Lernraten werden somit bessere Ergebnisse erzielt, wobei die Verbesserungen mit kleiner werdender Lernrate immer geringfügiger werden. Das Training mit weiteren kleineren Lernraten bleibt daher im Rahmen dieser Arbeit erspart. Für die weiteren Untersuchungen kommt das Modell zum Einsatz, das aus dem Training mit der Lernrate α_3 entstanden ist.

Wie beim MCTS-Agenten wurde eine einfache qualitative Analyse im Spiel gegen einen Menschen durchgeführt, um ein Bild von der Strategie des trainierten PPO-Agenten zu bekommen. Daraus geht hervor, dass der PPO-Agent stets als erstes das unterste Feld in der mittleren Spalte besetzt, wenn er dazu die Möglichkeit hat, was wie in Kapitel 6.2.3 bereits erwähnt, der stärkste Anfangszug ist. Der PPO-Agent tendiert dazu, seine Spielsteine vertikal zu stapeln, in manchen Fällen baut er auch horizontale Ketten. Wenn er drei Spielsteine in einer Kette platziert hat, setzt er meistens auch den vierten Stein zum Gewinnen, wenn dies möglich ist. Er blockiert hingegen äußerst selten seinen Gegenspieler, wenn dieser droht, in seinem nächsten Zug das Spiel für sich zu entscheiden. So ist es als Mensch sehr einfach möglich, gegen den RL-Agenten zu gewinnen, beispielsweise indem man selbst seine Steine in einer Spalte stapelt und von dieser Strategie nur abweicht, um den PPO-Agenten daran zu hindern, den vierten Stein in eine seiner Ketten zu platzieren.

Die in der quantitativen Analyse ermittelte Gewinnrate und Spieldauer des PPO-Agenten liegt mit 97,3 % (95 %-CI: 97,1 - 97,6) bzw. 10,62 Zügen (95 %-CI: 10,57 - 10,67) nahe bei den durch den MCTS-Agenten erzielten Werten von 100 % (95 %-CI: 98,1 - 100,0) bzw. 9,72 Zügen (95 %-CI: 9,32 - 10,12). Es scheint, als wären die Agenten ähnlich stark. Aus der qualitativen Analyse geht jedoch hervor, dass der MCTS-Agent eine langfristig ausgerichtete Strategie verfolgt, die auch gegen zielgerichtete Gegenspieler standhalten kann, während der PPO-Agent auf kurzfristige Gewinne abzielt, wodurch er mit einfachen Mitteln besiegt werden kann. Die kurzfristig ausgerichtete Strategie des PPO-Agenten entsteht dadurch, dass sie ausreicht, um im Training gegen den zufällig spielenden Agenten in den meisten Fällen zu gewinnen, sodass kein Lernanreiz für Strategien besteht, die auch gegen intelligentere Gegenspieler funktionieren.

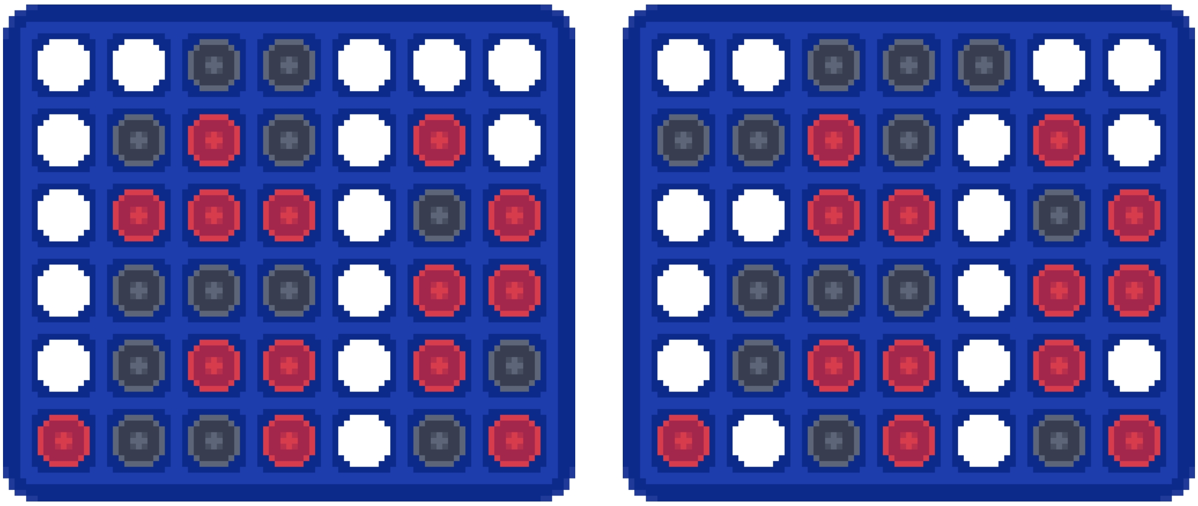
Die im Konzept geforderte Bedingung, dass die zu untersuchenden Agenten gleich stark sein müssen, um aussagekräftige Aussagen über die Robustheit der Verfahren zu äußern, konnte somit weiterhin nicht erfüllt werden. Es ist davon auszugehen, dass der implementierte RL-Agent alleine aufgrund seiner unterlegenen Spielstrategie auch in den Untersuchungen zur Robustheit schlechter abschneiden wird. Die Untersuchungen bezüglich Robustheit werden dennoch weiter fortgesetzt, denn noch ist nicht ausgeschlossen, dass der implementierte PPO-Agent entgegen der Erwartungen bessere Ergebnisse erzielt als der MCTS-Agent.

6.4. Szenarien zur Untersuchung von Robustheit

Um die im Konzept genannten Szenarien zur Untersuchung der Robustheit der Agenten 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 in einer Beobachtung veränderten Spielsteinen zu konfigurieren.

Die Klasse 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. Somit führt diese Methode Unsicherheiten bezüglich Aktionen ein.

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, deren unteren Nachbarn frei sind, und in die zweite Liste werden die Felder hinzugefügt, deren obere Nachbarn besetzt sind. Wie in Kapitel 5 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. Diese Methode ruft somit Unsicherheiten bezüglich Beobachtungen hervor. In Abbildung 14 wird beispielhaft dargestellt, wie durch diese Methode Spielfeldkonfigurationen verändert werden.



(a) Ohne veränderte Spielsteine. (b) Mit 5 veränderten Spielsteinen.

Abb. 14: Eine Spielfeldkonfiguration mit und ohne veränderten Spielsteinen im Vergleich. Grafikdesign aus [39].

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. Durch die in dieser Implementierung eingesetzte PettingZoo-Umgebung, wird der Fall, dass ein Stein in eine Spalte platziert wird, in der sich ein Stein über einem freien Feld befindet, so gehandhabt, 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 bespielbar 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.

7. Ergebnisdiskussion

Um die Robustheit der im Rahmen dieser Arbeit implementierten MCTS- und PPO-Agenten miteinander zu vergleichen, wurden sie in den in Kapitel 6.4 beschriebenen Szenarien im Spiel gegen einen zufällig spielenden Agenten evaluiert. Die Agenten spielen dabei mit abwechselndem Anzugsrecht gegen einen zufällig spielenden Agenten. Gemessen wird die Gewinnrate der zu untersuchenden Agenten in Abhängigkeit des Ausmaßes der in den Szenarien eingeführten Unsicherheit.

Für jeden untersuchten Unsicherheitsgrad wurden mit dem PPO-Agenten 2000 Spiele durchgeführt. Bei den Untersuchungen mit dem MCTS-Agenten wurden auf Grund des erhöhten Rechenbedarfs und der begrenzten Zeit nur 200 Spiele durchgeführt. Wie in Kapitel 5 beschrieben, wird der Gewinnratenverlust als relatives Maß für die Robustheit der Verfahren verwendet. Ein höherer Gewinnratenverlust deutet dabei auf ein weniger robustes Verfahren hin. Die Diagramme zu den tatsächlich gemessenen Gewinnraten im Spiel gegen den zufällig spielenden Agenten befinden sich im Anhang unter A.3.

7.1. Unsicherheit bezüglich Aktionen

Beim Vergleich der Verfahren hinsichtlich der Robustheit gegenüber Unsicherheiten bezüglich Aktionen wurde die Wahrscheinlichkeit, anstelle des durch den Agenten gewählten Zuges einen zufälligen Zug durchzuführen, in 10-Prozentpunktschritten von 0 % bis 100 % variiert.

Beträgt diese Wahrscheinlichkeit 0 %, befindet sich der Gewinnratenverlust bei beiden Agenten ebenfalls bei 0 %, da es sich bei den dabei erzielten Gewinnraten um den Ausgangswert für die Berechnung des Gewinnratenverlustes handelt.

Ab einer Wahrscheinlichkeit für zufällige Aktionen von 20 % ist für den MCTS-Agenten ein signifikant niedrigerer Verlust von 1,0 % (95 %-CI: 0,2 - 5,6) gegenüber von 11,3 % (95 %-CI: 9,0 - 14,0) bei PPO zu verzeichnen. Einen solche signifikante Differenz hält sich bis zu einer Wahrscheinlichkeit von 80 % für zufällige Aktionen, wo ein Gewinnratenverlust von 54 % (95 %-CI: 42,6 - 67,0) beim MCTS-Agenten und 78,7 % (95 %-CI: 74,2 - 83,3) beim PPO-Agenten ermittelt wurde. Damit ist der MCTS-Agent in diesem Bereich signifikant robuster als der PPO-Agent.

Bei 90 % liegen die Messpunkte von MCTS mit 66,0 % (95 %-CI: 53,8 - 79,6) gegenüber 85,4 % (95 %-CI: 80,8 - 90,1) zwar weiterhin höher, jedoch liegen die Konfidenzintervalle der Messpunkte so nah bei einander, dass aus den Werten allein nicht mit ausreichender Gewissheit Schlüsse über die tatsächlichen Werte gezogen werden können.

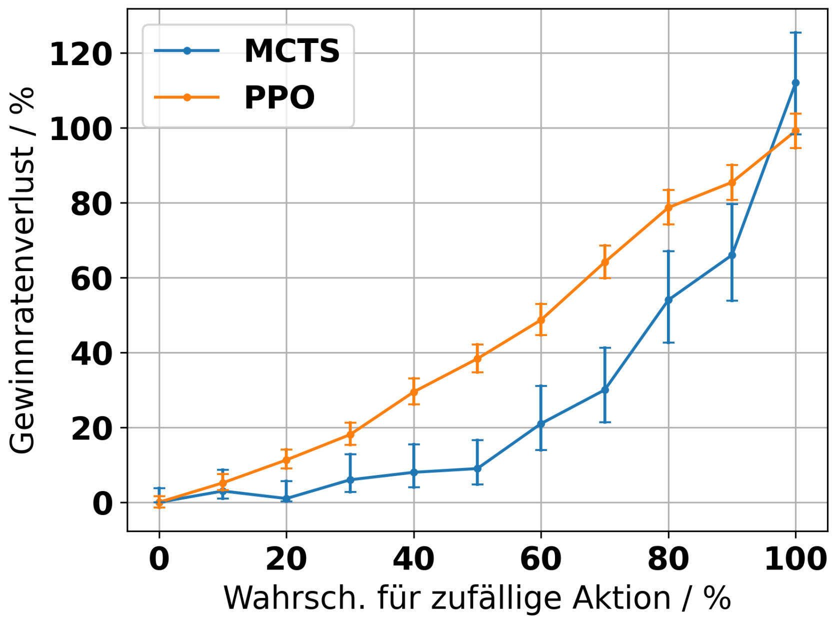


Abb. 15: Gewinnratenverlust in Abhängigkeit von der Wahrscheinlichkeit, eine zufällige Aktion durchzuführen.

Bei einer Wahrscheinlichkeit für zufällige Aktionen von 100 % sollten beide Agenten nach genügend durchgeführten Spielen gegen den zufällig spielenden Agenten eine Gewinnrate von 50 % und damit einen Gewinnratenverlust von 100 % erzielen. Die Messpunkte liegen für MCTS mit 112,0 % (95 %-CI: 98,2 - 125,4) leicht darüber bzw. für PPO mit 99,2 % (95 %-CI: 94,5 - 108,3) leicht darunter. Da ein Gewinnratenverlust von 100 % innerhalb der Konfidenzintervalle liegt, ist es naheliegend, dass die gemessenen Abweichungen stochastisch bedingt sind.

Aus den Messungen geht für eine Wahrscheinlichkeit für zufällige Aktionen von 20 % bis 80 % der MCTS-Agent als der robustere Agent hervor. Für darunter bzw. darüber liegende Wahrscheinlichkeiten kann aufgrund von Messungenauigkeiten kein signifikanter Unterschied gemessen werden. Der Verlauf des Gewinnratenverlustes liegt jedoch nahe, dass MCTS auch dort weiterhin einen niedrigen tatsächlichen Gewinnratenverlust aufweist.

Es wurde nicht erwartet, im Szenario mit Unsicherheiten bezüglich Aktionen eine signifikante Differenz im Gewinnratenverlust zwischen den beiden Verfahren zu sehen, die durch die Verfahren selbst bedingt ist. Es liegt nahe, dass die Diskrepanz dadurch hervorgerufen wird, dass der PPO-Agent unzureichend trainiert wurde, wodurch er eine kurzfristigere Strategie besitzt, die anfälliger für Störungen ist.

7.2. Unsicherheit bezüglich Beobachtungen

Beim Vergleich der Verfahren hinsichtlich der Robustheit gegenüber Unsicherheiten bezüglich Beobachtungen wurde die Anzahl von veränderten Spielsteinen, in 2er-Schritten von 0 bis 20 variiert. Aufgrund der größer werdenden Konfidenzintervalle wurden ab 10 veränderten Spielsteinen nicht mit 200 sondern 500 Spiele zwischen dem MCTS-Agenten und dem zufällig spielenden Agenten ausgetragen.

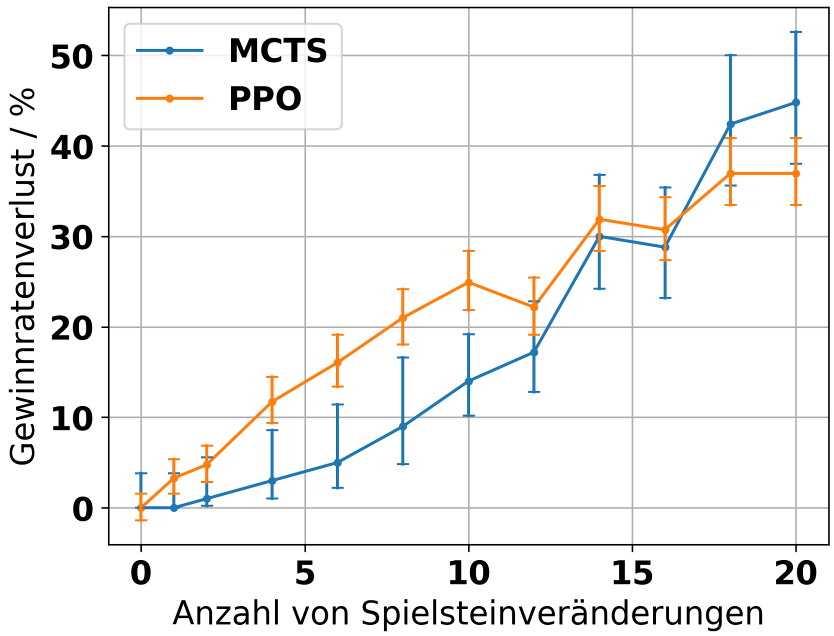


Abb. 16: Gewinnratenverlust in Abhängigkeit von der Anzahl der veränderten Spielsteine.

Ähnlich wie im vorherigen Abschnitt 7.1 startet der Gewinnratenverlust beider Agenten bei 0 veränderten Spielsteinen mit 0 %.

Ab 4 veränderten Spielsteinen lässt sich beim MCTS-Agenten mit 3,0 % (95 %-CI: 1,0 - 8,6) ein signifikant niedrigerer Gewinnratenverlust gegenüber des PPO-Agenten mit 11,7 % (95 %-CI: 9,4 - 14,5) verzeichnen. Diese Diskrepanz hält sich bis zu einer Anzahl von 10 veränderten Spielsteinen, wobei der MCTS-Agent einen Gewinnratenverlust von 14,0 % (95 %-CI: 10,2 - 19,2) und der PPO-Agent einen Verlust von 24,9 % (95 %-CI: 21,9 - 28,4) erzielt. Das zeigt, dass sich der MCTS-Agent in diesem Bereich signifikant robuster verhält als der PPO-Agent.

Ab 12 veränderten Spielsteinen sind die Differenzen zwischen den Messpunkten zu gering und die Konfidenzintervalle zu groß, dass nicht mit ausreichender Sicherheit bestimmt werden kann, bei welchem Agenten die tatsächlichen Werte höher oder niedriger liegen.

Bei 18 und 20 veränderten Spielsteinen liegt der gemessene Verlust von MCTS mit 42,4 % (95 %-CI: 35,6 - 50,0) und 44,8 % (95 %-CI: 38,0 - 52,6) sogar über dem von PPO mit 37,0 % (95 %-CI: 33,5 - 40,9) in beiden Fällen. Aufgrund der großen Konfidenzintervalle kann jedoch nicht gesagt werden, ob sich die tatsächlichen Werte genauso verhalten. Um Aussagen darüber treffen zu können, sind Messungen mit mehr Wiederholungen nötig.

Bei 4 bis 10 veränderten Spielsteinen geht der MCTS-Agent als der robustere Agent hervor. Aufgrund des Verlaufs des Gewinnratenverlustes in Abhängigkeit der Anzahl der veränderten Spielsteinen liegt nahe, dass dies auch für eine Anzahl von unter 4 gilt. Liegt die Anzahl der veränderten Spielsteinen über 10, können aufgrund der zu großen Messungenauigkeiten keine signifikanten Unterschiede festgestellt werden. Die beobachtete höhere Robustheit des MCTS-Agenten könnte durch die Anfälligkeit von neuronalen Netzwerken gegenüber unerwarteten Beobachtungen verursacht worden sein. Ebenso wie in Abschnitt 7.1 könnte die Ursache für dieses Verhalten jedoch auch in der kurzfristigen Strategie des in dieser Arbeit unzureichend trainierten PPO-Modells liegen.

7.3. Verallgemeinerungen

Die Messergebnisse zeigen, dass der im Rahmen dieser Arbeit implementierte MCTS-Agent robuster gegenüber Unsicherheiten bezüglich Aktionen ist als der implementierte PPO-Agent. Dies gilt auch für die Robustheit gegenüber Unsicherheiten bezüglich Beobachtungen, zumindest bis zu dem Grad, ab dem die Messungenauigkeiten den Vergleich des Gewinnratenverlustes zwischen den beiden Agenten nicht mehr zulassen.

In Kapitel 6.3.2 wurde gezeigt, dass der implementierte PPO-Agent schlechter spielt als der MCTS-Agent, da das Training gegen einen zufällig spielenden Agenten nicht genügt, um eine langfristig ausgerichtetes Regelwerk zu erlernen. Es liegt nahe, dass die kurzfristig ausgerichtete Strategie des PPO-Agenten auch anfälliger für Störungen ist. Damit kann nicht bestimmt werden, ob die Ursache für die gemessene niedrigere Robustheit in der bereits in der neutralen Umgebung schlechteren Leistungsfähigkeit oder tatsächlich an der niedrigeren Robustheit des PPO-Verfahrens liegt. Daher gelten die Aussagen nur für die konkreten Implementierungen der Verfahren in dieser Arbeit und können nicht auf alle Implementierungen der Verfahren verallgemeinert werden. Damit sind auch keine allgemeinen Aussagen über die Robustheit von symbolischen Algorithmen und RL-Verfahren im Vergleich möglich.

Um Aussagen über das Verhalten der Agenten auf die Verfahren MCTS und PPO bzw. symbolische Algorithmen und RL-Verfahren verallgemeinern zu können, muss sichergestellt sein, dass die Agenten unter neutralen Bedingungen möglichst eine gleichwertige Leistung erzielen. Im Rahmen dieser Arbeit konnten durch quantitative und qualitative Analysen der Agenten gezeigt werden, dass dies in diesem Fall nicht zutrifft. Es ist kritisch zu hinterfragen, inwiefern diese Methodiken ausreichen würden, um zu zeigen, dass die Agenten eine gleichwertige Leistung erzielen. Denn im Spiel gegen einen zufällig spielenden Agenten spiegeln sich große Leistungsunterschiede nur in sehr geringfügigen Differenzen wieder. Und in der qualitativen Analyse können subjektive Verzerrungen der Wahrnehmung das Ergebnis beeinflussen. Besser geeignet sind beispielsweise das Spiel gegen einen perfekt spielenden Agenten wie in [40] oder gegen unterschiedliche nicht-perfekt spielende Agenten in einer Turnier-Umgebung, so wie es in [44] umgesetzt wurde.

Es liegt nahe, dass durch die Untersuchung von zwei Verfahren an einem Anwendungsfall nicht ausreicht, um verallgemeinernde Aussagen über den Vergleich der Robustheit zwischen symbolischen Algorithmen und RL-Verfahren treffen zu können, sondern dazu verschiedene Anwendungsfälle und verschiedene Lösungsverfahren untersucht werden müssen.

8. Zusammenfassung und Ausblick

Aus den im Rahmen dieser Arbeit durchgeführten Experimenten geht hervor, dass der in dieser Arbeit implementierte MCTS-Agent robuster hinsichtlich Unsicherheiten gegenüber Aktionen ist als der in dieser Arbeit implementierte PPO-Agent. Hinsichtlich Robustheit bezüglich Unsicherheiten gegenüber Beobachtungen konnte dies aufgrund von stochastisch bedingten Messungenauigkeiten nur bis zu einem gewissen Grad an Unsicherheit ebenfalls festgestellt werden. Aufgrund begrenzter Ressourcen war es nicht möglich, einen PPO-Agenten zu trainieren, der in der neutralen Umgebung ohne Unsicherheiten mit dem MCTS-Agenten mithalten kann. Es ist davon auszugehen, dass die kurzfristigere Strategie des PPO-Agenten anfälliger für die in den Szenarien zur Untersuchung von Robustheit eingeführten Unsicherheiten ist. Die Beobachtung, dass der MCTS-Agent robuster als der PPO-Agent ist, kann damit nicht auf die Verfahren MCTS und PPO und schon gar nicht auf symbolische Algorithmen und Reinforcement Learning verallgemeinert werden. Damit wurde nur ein Teilbereich der zu untersuchenden Frage beantwortet, inwiefern symbolische Algorithmen oder RL-Verfahren robuster sind.

Mit der in dieser Arbeit angewandten Methodik war es jedoch möglich, die Robustheit von zwei konkreten Implementierungen zu quantifizieren. Es ist naheliegend, dass auch weitere Teilbereiche der zu untersuchenden Frage beantwortet werden können, indem die Methodik auf weitere MDPs und entsprechenden Lösungsverfahren übertragen wird. Diese erfordert lediglich, dass das zu untersuchende MDP gezielt auf Aspekte von Robustheit modifiziert, die zu untersuchenden Lösungen auf das modifizierte MDP angewandt und der Verlust ihrer Leistungsfähigkeit untersucht wird.

Um die Frage zu beantworten, wie sich der Vergleich der Robustheit von MCTS und PPO bei Vier Gewinnt allgemein verhält, ist es notwendig, zwei Agenten zu untersuchen, die unter neutralen Bedingungen möglichst gleich stark sind. Für das Training eines stärkeren PPO-Agenten scheint die Trainingsstrategie von Zhong et al. in Kombination mit mit Faltungsmatritzen ausgestatteten neuronalen Netzwerken einen vielversprechenden Ansatz darzustellen. Um Aussagen über die Robustheit von MCTS und PPO über verschiedene Anwendungsfälle verallgemeinern zu können, sind Untersuchungen mit anderen Anwendungsfällen notwendig, die beispielsweise verschiedene Zustands- und Aktionsräume aufweisen oder die Möglichkeit bieten, physikalische Parameter der Problemumgebung zu variieren. Aussagen über den Vergleich von Robustheit zwischen symbolische Algorithmen und RL-Verfahren allgemein erfordern Untersuchungen mit ausreichend verschiedenen Verfahren und Anwendungsszenarien. Hierbei stellt sich jedoch die Frage, inwiefern dies aufgrund der Vielzahl von zu lösenden Problemen und entsprechenden Lösungsverfahren möglich oder überhaupt erstrebenswert ist.

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

[3] 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.

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

[5] 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.

[6] Lawrence D. Brown, T. Tony Cai, Anirban DasGupta. „Interval Estimation for a Binomial Proportion". In: *Statistical Science* 16.2 (2001), S. 101 –133. DOI: 10.1214/ss/1009213286. URL: https://doi.org/10.1214/ss/1009213286.

[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] Irasianty Frost. *Konfidenzintervalle und Standardfehler-Balken: Das Konzept verstehen und Ergebnisse angemessen interpretieren*. Jan. 2023. ISBN: 978-3-662-67097-2. DOI: `10.1007/978-3-662-67098-9`.

[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] Shengyi Huang, Santiago Ontañón. „A Closer Look at Invalid Action Masking in Policy Gradient Algorithms“. In: *The International FLAIRS Conference Proceedings* 35 (Mai 2022). ISSN: 2334-0762. DOI: `10.32473/flairs.v35i.130584`. URL: `http://dx.doi.org/10.32473/flairs.v35i.130584`.

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

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

[23] Markus Janczyk, Roland Pfister. *Inferenzstatistik verstehen: Von A wie Signifikanztest bis Z wie Konfidenzintervall*. Jan. 2020. ISBN: 978-3-662-59908-2. DOI: `10.1007/978-3-662-59909-9`.

[24] 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.

[25] M. Lapan. *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing, 2020. ISBN: 9781838820046. URL: `https://books.google.de/books?id=O0vODwAAQBAJ`.

[26] 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 Bein 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`.

[27] 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`.

[28] 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`.

[29] 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`.

[30] 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.

[31] Antonin Raffin u. a. „Stable-Baselines3: Reliable Reinforcement Learning Implementations“. In: *Journal of Machine Learning Research* 22.268 (2021). `https://stable-baselines.readthedocs.io/`, S. 1–8. URL: `http://jmlr.org/papers/v22/20-1364.html` (besucht am 26. 01. 2025).

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

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

[34] 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.

[35] 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.

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

[37] 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.

[38] 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.

[39] J Terry u. a. „Pettingzoo: Gym for multi-agent reinforcement learning“. In: *Advances in Neural Information Processing Systems* 34 (2021). https://pettingzoo.farama.org/, S. 15032–15043. (Besucht am 26. 01. 2025).

[40] 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.

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

[42] Sean Wallis. „Binomial Confidence Intervals and Contingency Tests: Mathematical Fundamentals and the Evaluation of Alternative Methods“. In: *Journal of Quantitative Linguistics* 20.3 (2013), S. 178–208. DOI: 10.1080/09296174.2013.799918. eprint: https://doi.org/10.1080/09296174.2013.799918. URL: https://doi.org/10.1080/09296174.2013.799918.

[43] 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.

[44] Yuanyi Zhong, Yuan Zhou, Jian Peng. *Efficient Competitive Self-Play Policy Optimization*. 2020. arXiv: 2009.06086 [cs.LG]. URL: https://arxiv.org/abs/2009.06086.

A. Anhang

A.1. Trainingsmetriken aus dem PPO-Training nach Zhong et al.

A.1.1. Gewinnrate

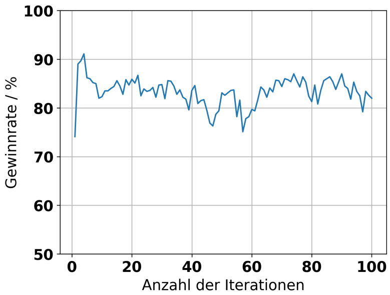


Abb. 17: Verlauf der Gewinnrate mit Lernrate α_0

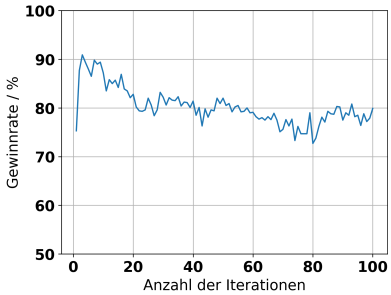


Abb. 18: Verlauf der Gewinnrate mit Lernrate α_1

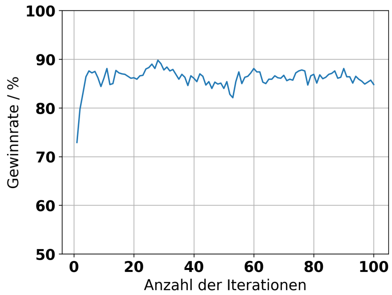


Abb. 19: Verlauf der Gewinnrate mit Lernrate α_2

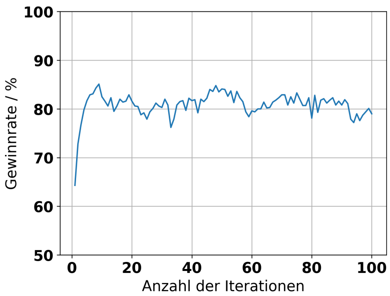


Abb. 20: Verlauf der Gewinnrate mit Lernrate α_3

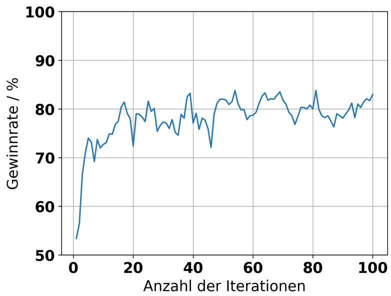


Abb. 21: Verlauf der Gewinnrate mit Lernrate α_4

A.1.2. Spieldauer

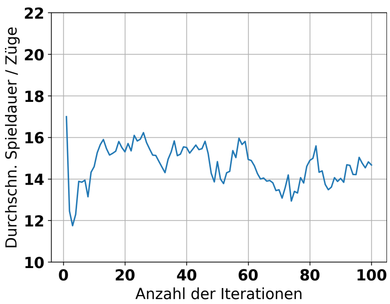


Abb. 22: Verlauf der Spieldauer mit Lernrate α_0

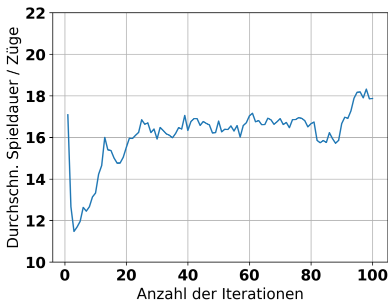


Abb. 23: Verlauf der Spieldauer mit Lernrate α_1

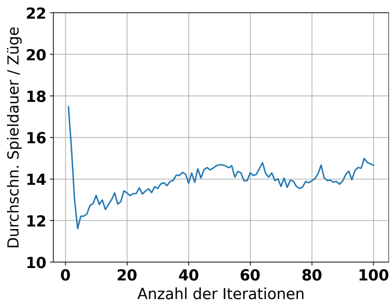


Abb. 24: Verlauf der Spieldauer mit Lernrate α_2

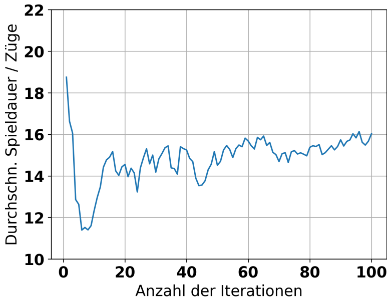


Abb. 25: Verlauf der Spieldauer mit Lernrate α_3

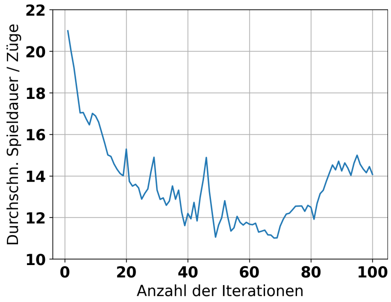


Abb. 26: Verlauf der Spieldauer mit Lernrate α_4

A.1.3. Werteverlust

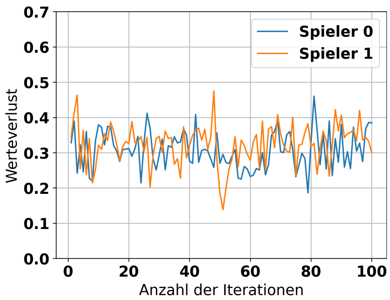


Abb. 27: Verlauf des Werteverlustes mit Lernrate α_0

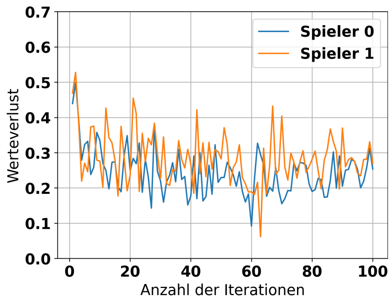


Abb. 28: Verlauf des Werteverlustes mit Lernrate α_1

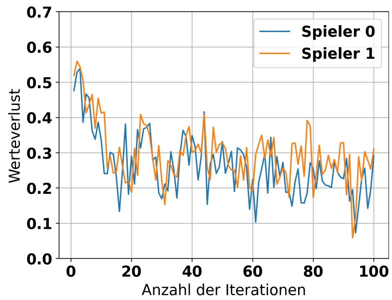


Abb. 29: Verlauf des Werteverlustes mit Lernrate α_2

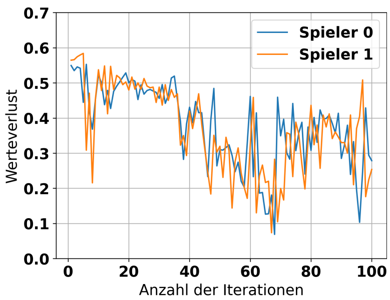


Abb. 30: Verlauf des Werteverlustes mit Lernrate α_3

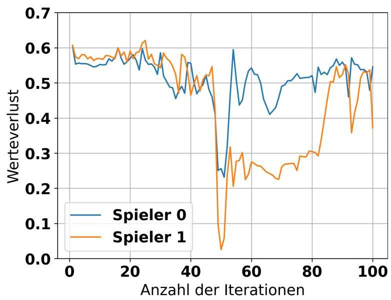


Abb. 31: Verlauf des Werteverlustes mit Lernrate α_4

A.1.4. Policy-Gradient-Verlust

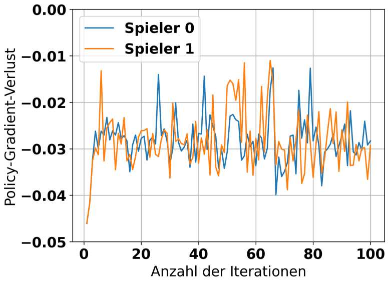


Abb. 32: Verlauf des Policy-Gradient-Verlustes mit Lernrate α_0

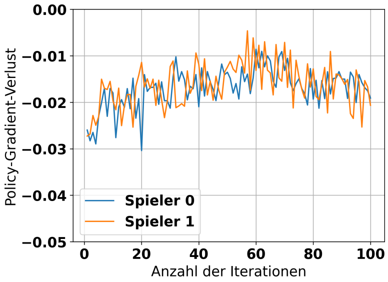


Abb. 33: Verlauf des Policy-Gradient-Verlustes mit Lernrate α_1

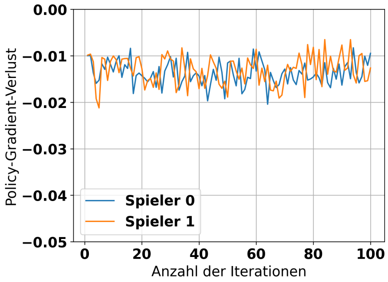


Abb. 34: Verlauf des Policy-Gradient-Verlustes mit Lernrate α_2

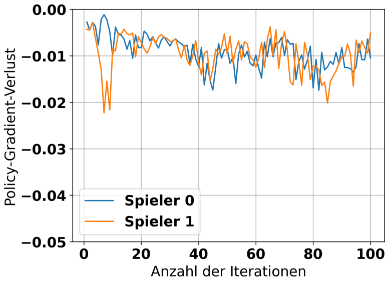


Abb. 35: Verlauf des Policy-Gradient-Verlustes mit Lernrate α_3

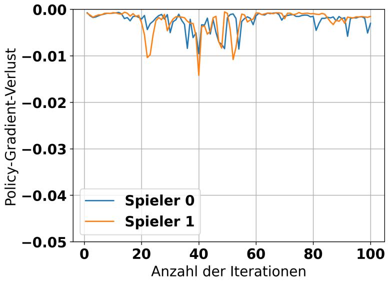


Abb. 36: Verlauf des Policy-Gradient-Verlustes mit Lernrate α_4

A.2. Trainingsmetriken aus dem PPO-Training gegen einen zufällig spielenden Agenten

A.2.1. Durchschnittliche Belohnung

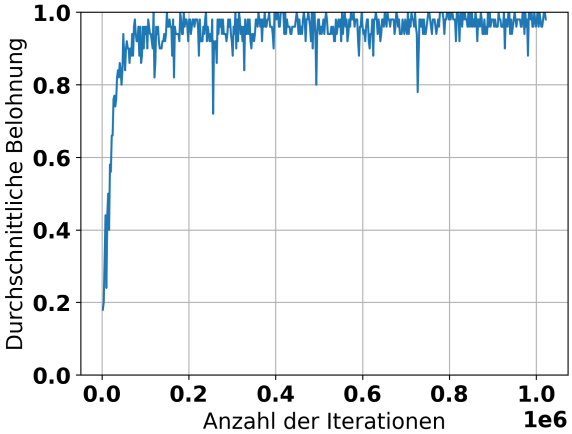


Abb. 37: Verlauf der durchschnittlichen Belohnung mit Lernrate α_0

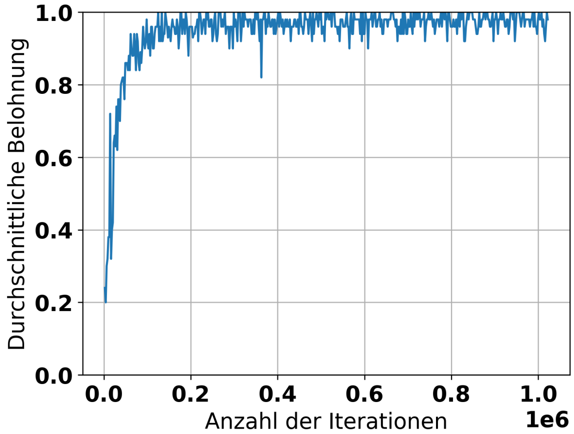


Abb. 38: Verlauf der durchschnittlichen Belohnung mit Lernrate α_1

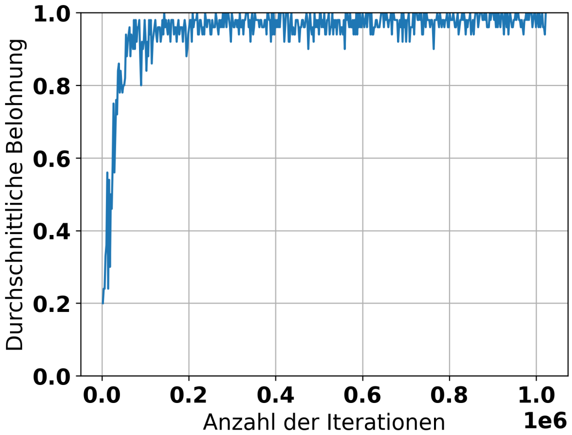


Abb. 39: Verlauf der durchschnittlichen Belohnung mit Lernrate α_2

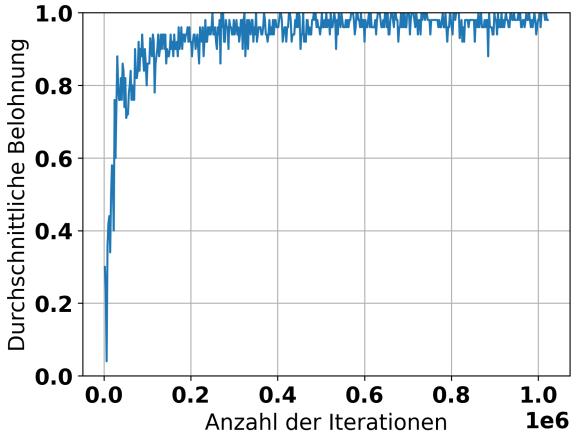


Abb. 40: Verlauf der durchschnittlichen Belohnung mit Lernrate α_3

A.2.2. Durchschnittliche Episodenlänge

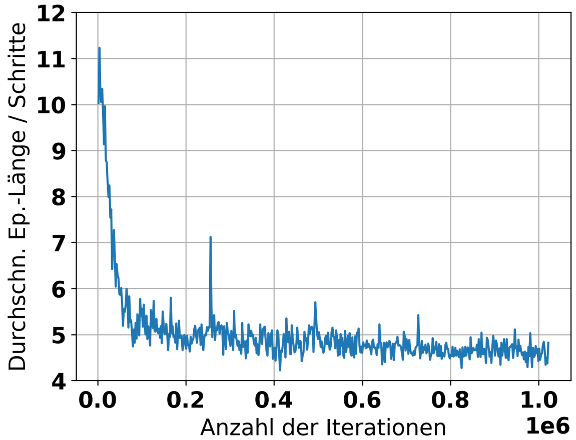


Abb. 41: Verlauf der durchschnittlichen Episodenlänge mit Lernrate α_0

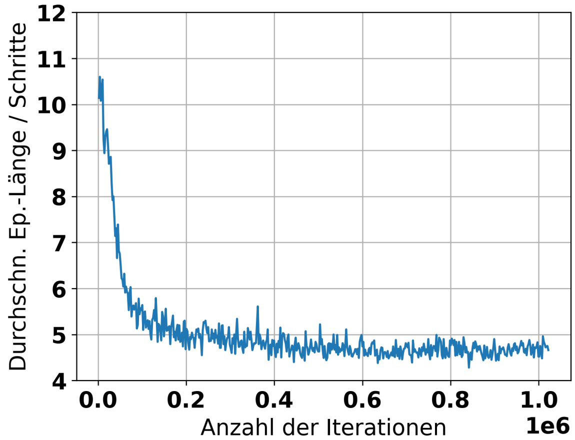


Abb. 42: Verlauf der durchschnittlichen Episodenlänge mit Lernrate α_1

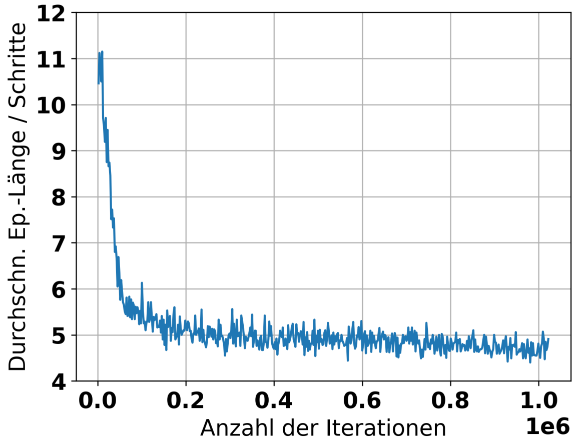


Abb. 43: Verlauf der durchschnittlichen Episodenlänge mit Lernrate α_2

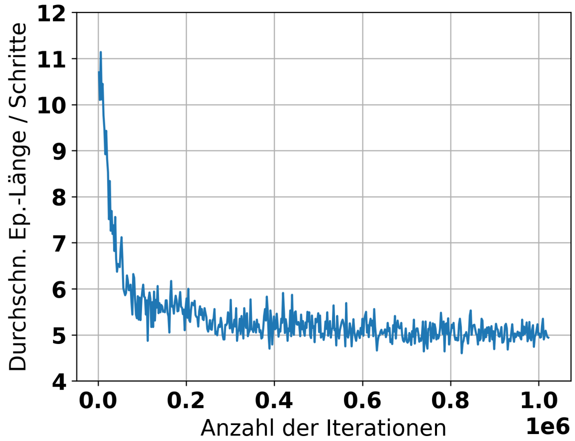


Abb. 44: Verlauf der durchschnittlichen Episodenlänge mit Lernrate α_3

A.3. Ursprüngliche Gewinnraten zur Berechnung des Gewinnratenverlustes

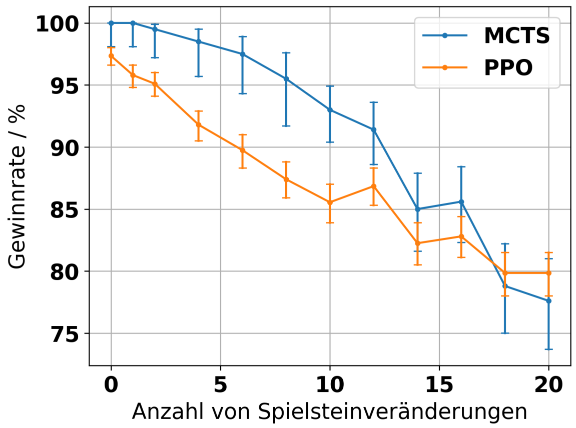


Abb. 45: Gewinnrate in Abhängigkeit von der Anzahl der veränderten Spielsteine

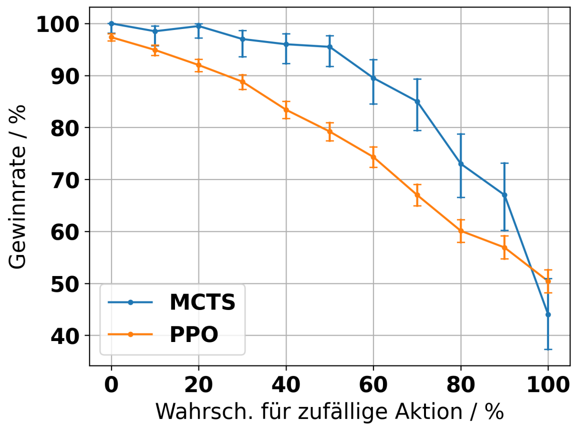


Abb. 46: Gewinnrate in Abhängigkeit von der Wahrscheinlichkeit, eine zufällig Aktion durchzuführen