AlphaZero gegen MCTS: Eine Studie zur KI-basierten Spielstrategie in Gomoku

Bachelorarbeit

Yamac Eren Ay
456272

7. Mai 2024

Betreuer: Prof. Dr. Benjamin Blankertz
Dr.- Ing. Stefan Fricke

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Neurotechnologie

Kurzfassung

Gomoku, ein kombinatorisches Brettspiel mit unkomplizierten Spielregeln, bietet vielfältige Lernherausforderungen im Bereich Künstlicher Intelligenz. Im Fokus steht der AlphaZero-Algorithmus, der den Monte-Carlo-Baumsuche (MCTS) mit Reinforcement-Learning kombiniert, um die aufwendigen MCTS-Schritte – insbesondere Simulation und Expansion – sowohl in Bezug auf Zeit als auch Leistung zu optimieren. Durch die Nutzung konvolutioneller neuronaler Netzwerke (CNNs) für die automatische Merkmalsextraktion, gelingt es AlphaZero, ohne vorgegebenes Expertenwissen bemerkenswerte Leistungen zu vollbringen, die traditionelle MCTS-Ansätze mit einer wesentlich höheren Anzahl an Iterationen übertreffen. Die Auswertung von Trainingsmetriken und Wettbewerbsergebnissen in drei unterschiedlichen Spielkonfigurationen zeigt, dass AlphaZero mit fortschreitendem Training immer bessere Resultate erzielt und sogar effektive Gewinnstrategien aufdecken kann. Die Erkenntnisse deuten außerdem darauf hin, dass es Potenzial für weitere Verbesserungen in Trainings- sowie Evaluierungsmethoden gibt, was das transformative Potenzial von AlphaZero in der Welt der Spielalgorithmen unterstreicht.

Danksagung

Mein tiefer Dank gilt den Personen und Ressourcen von der Forschungsgruppe NEURO TU Berlin, die einen unverzichtbaren Beitrag zu dieser Arbeit geleistet haben, vor allem meinem Erstbetreuer Prof. Dr. Benjamin Blankertz für seine stetige Unterstützung, transparente Kommunikation, und konstruktive, schnelle Rückmeldungen. Außerdem bedanke ich mich herzlich bei ihm und Dominik Kühne für ihre hilfreiche Unterstützung bei Fragen rund um den Zugriff auf Universitätsrechner und die Gewährleistung dessen Funktionsfähigkeit. Dr.-Ing. Stefan Fricke bin ich im Besonderen für die Betreuung meiner Bachelorarbeit und die Einblicke in die spieltheoretische Perspektive zu großem Dank verpflichtet. Ich verdanke Junxiao Song für sein inspirierendes KI-Projekt [1], Katherina Babenkova für die Weitergabe ihrer Bachelorarbeit [11], und Philipp Reinke für wissenschaftliche Diskussionen, die unter anderem zur Erwähnung des MuZero-Algorithmus führten. Schließlich möchte ich meinem Freundeskreis und meiner Mutter meinen tiefsten Dank aussprechen für ihre bedingungslose mentale Unterstützung während der Anfertigung dieser Bachelorarbeit. Jeder Einzelne von Ihnen hat auf einzigartige Weise zum Gelingen dieser Arbeit beigetragen, vielen Dank!

Im Laufe dieser Arbeit habe ich die folgenden Programme verwendet, die einen wesentlichen Bestandteil meiner Arbeit ausmachten: ChatGPT [3] für die Umformulierung von einigen Sätzen in Kapitel 1 und 4, Draw.io [4] für die Erstellung von allen erforderlichen Abbildungen, und PyTorch [8] für die Bereitstellung vieler nützlichen Klassen und Methoden zur Erstellung und Trainierung von KI-Modellen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1. Einleitung

1.1. Gomoku als Strategiespiel

Gomoku, also bekannt als "Fünf in der Reihe", ist ein kombinatorisches 2-Spieler-Spiel. Bekannt durch seine gewissen Ähnlichkeiten zu Go und Gobang, wird es heutzutage nicht nur von Spielexperten untersucht, sondern auch häufig im Bereich Künstliche Intelligenz geforscht. Dabei spielt es eine wichtige Rolle, dass Gomoku ein leicht implementierbares Spiel mit einfachen klaren Spielregeln ist.

1.1.1. Spielregeln und -eigenschaften von Gomoku

Auf traditionelle Weise wird das Spiel mit (schwarzen und weißen) Go-Steinen auf einem Go-Brett der Größe 19×19 gespielt, aber auch auf Bretter mit 15×15 bzw. 17×17 Feldern. Ursprünglich heißt das Spiel auf Japanisch als "Gomoku narabe" (übersetzt als "Fünf Steine aneinanderreihen"). Zudem existieren aber auch unterschiedlichen Varianten mit verschiedenen (M, N, K)-Konfigurationen, wobei $M \times N$ die Brettgröße und K die Mindestlänge einer Gewinnreihe ist.

Der Spielverlauf wird folgendermaßen beschrieben: In der Regel fängt Schwarz das Spiel an und beide Spieler setzen abwechselnd einen Stein ihrer Farbe auf ein freies Feld, bis eine Gewinnreihe gebildet ist, oder alle Felder besetzt sind. Eine Gewinnreihe ist eine ununterbrochene Reihe, bestehend aus 5 Steinen gleicher Farbe, und sie kann horizontal, vertikal oder diagonal verlaufen. Wenn am Ende des Spiels keine Gewinnposition vorliegt, endet das Spiel unentschieden.

Interessanterweise dürfen - im Gegensatz zu Go - die besetzten Felder auf keine Weise neu freigesetzt werden, also bleiben solche Felder weiterhin besetzt und wächst die Anzahl der besetzten Felder streng monoton bis zum Ende des Spiels. Aus diesem Grund könnte dieses Spiel alternativerweise auf einem Blatt gespielt werden, wobei der erste bzw. zweite Spieler üblicherweise ein Feld mit einem Kreuz X bzw. Kreis O markieren würde.

1.1.2. Komplexitätsanalyse und Modellierung von Gomoku

Gomoku besitzt ein "deterministisches sequentielles endliches 2-Spieler-MDP" (siehe Anhang A.1), deshalb wird es stark vermutet, dass es ein PSPACE-vollständiges Spiel ist [33]. In anderen Worten ist ein verallgemeinertes Gomoku-Spiel mit einem Brett der Größe $M \times N$ und einer Mindestlänge K ein schwierigstes Problem in PSPACE, und somit können alle QSAT-Probleme durch ein Gomoku-Spiel repräsentiert werden.

Diese Eigenschaft zur Kenntnis nehmend kann theoretisch jede solche Frage in endlicher Zeit beantwortet werden, ob eine Gewinnstrategie für einen Spieler in einem beliebigen Spielzustand vorliegt. Dabei handelt es sich um das folgende Minimax-Problem:

- Ein Minimax-Suchbaum wird aufgestellt, wobei die Blätter mit einer vorliegenden Gewinnposition von dem ersten bzw. zweiten Spieler eine Bewertung von $+1$ bzw. -1 annehmen, und sonst alle anderen Knoten 0.
- Der Suchbaum ist in abwechselnden Max- bzw. Min-Ebenen unterteilt, wobei das Ziel für die jeweilige Max- und Min-Ebene ist, die Kindknoten so auszuwählen, dass die Bewertung des aktuellen Knotens maximiert bzw. minimiert wird.
- Dann sollte die rekursiv definierte Minimax-Formel $\max\{\min\{\max \ldots \{r_i, \ldots\}\}\}$ immer die korrekte Antwort in endlicher Zeit liefern, falls es eine gibt. Hat der erste Spieler bspw. eine Gewinnstrategie, liefert diese Formel einen Wert von $+1$ heraus.

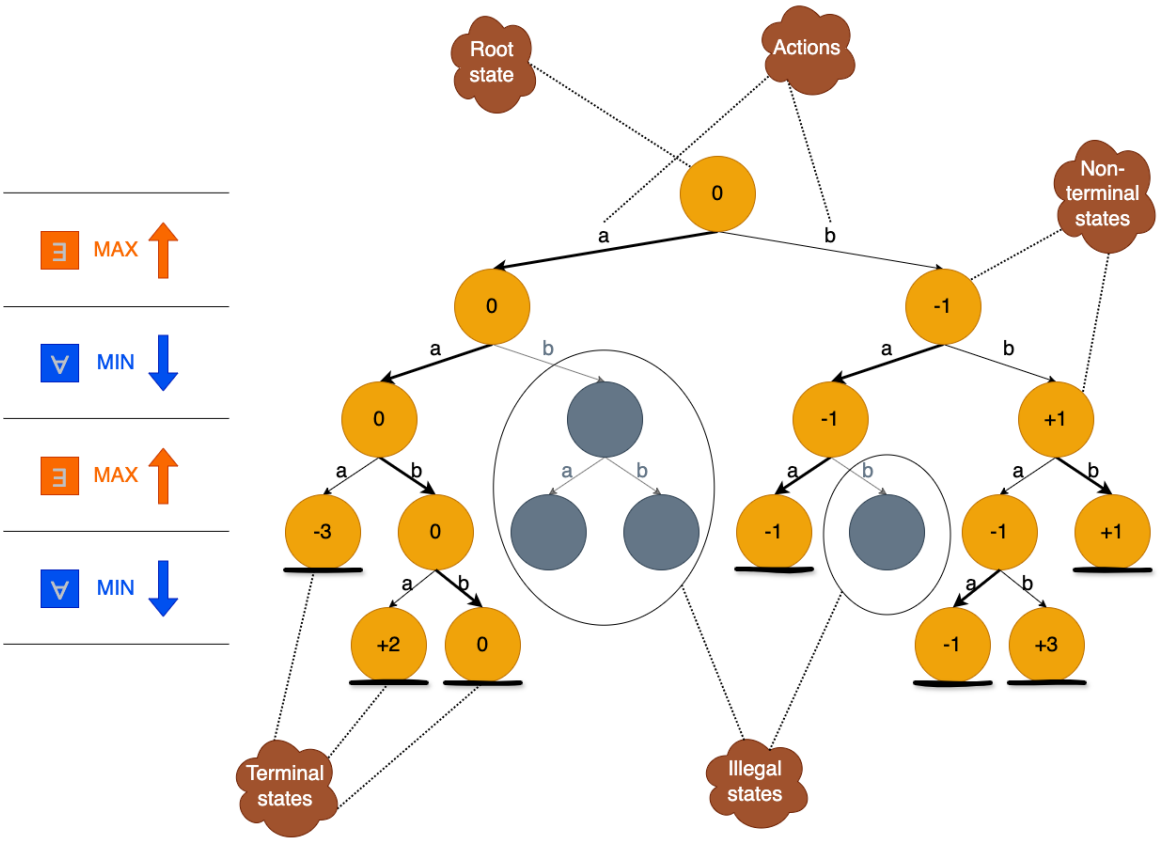


Abbildung 1.1.: Ein Beispiel von Minimax-Problem

Jedoch ergibt sich daraus im schlimmsten Fall ein hoher Branchfaktor von n und eine hohe Suchtiefe von n für eine Brettgröße $n := M \times N$ (siehe Anhang A.2). Sogar in einem Spiel der Größe 5×5 gibt es theoretisch insgesamt $25! \approx 1,5511 \cdot 10^{25}$ verschiedene Endzustände zu untersuchen, somit ist ein solcher Ansatz in der Praxis kaum realisierbar. Daher werden effizientere und leistungsfähigere Minimax-Algorithmen benötigt.

1.2. Relevante Ansätze für Gomoku-Spielstrategien

Im Folgenden werden verschiedene Ansätze für die Strategiefindung in Gomoku vorgestellt, die die Anzahl der untersuchten Knoten im Suchbaum erheblich reduzieren.

1.2.1. Alpha-Beta-Varianten

Alpha-Beta-Suche ist eine allzu bekannte Suchmethode, die ohne Informationsverlust das Minimax-Suchverfahren zeitlich verbessert ([23], *0. Introduction*). Statt eine vollständige Baumsuche - wie in Minimax-Algorithmus vorgestellt wird - werden die irrationalen Zugentscheidungen nicht weiter betrachtet, die die Spielsituation eines Spielers unbedingt verschlechtern.

Das Alpha-Beta-Intervall dient dazu, solche irrationalen Entscheidungen auszusortieren. Wenn ein Spieler weiß, sein Gegner würde einen Zug auf keinen Fall durchführen, dann braucht er nicht mehr diesen Zug zu betrachten.

In der Praxis sorgt Alpha-Beta-Suche für deutlich kürzere Laufzeiten, jedoch genießt im schlimmsten Fall dieselbe Worst-Case-Komplexität wie Minimax. Um den Algorithmus effizienter zu gestalten, wurden verschiedene Varianten ([35], *S.3*) vorgeschlagen, wie z. B. :

- Iterative-Deepening, wie im Namen darauf hingewiesen wurde, durchsucht den Baum schrittweise nach Suchtiefe und ist bekannt dafür, zu früheren und strikteren Cutoffs im Laufe der Iterationen zu führen.
- Transpositionstabelle, also eine Lösungsansatz aus Dynamischer Programmierung, verhindert duplikate Zuguntersuchungen in Überschneidungsfällen mittels Zwischenspeicherung von bereits bearbeiteten Positionen in einer großen Hashtabelle.
- Aspirationssuche, also die Durchführung von Alpha-Beta Suche mit einem kleineren Alpha-Beta-Intervall, sorgt für frühere und striktere Cutoffs, kann aber die optimalen Lösungen übersehen, wenn das Intervall zu klein gewählt wird.

Alle bisherigen Varianten werden im Laufe dieser Arbeit als Alpha-Beta-Varianten zusammengefasst, denn sie alle haben die gemeinsame Eigenschaft, schlimmstenfalls den ganzen Suchbaum durchsuchen zu müssen.

1.2.2. MCTS und UCT

Monte-Carlo Tree Search (auch MCTS) ist eine iterative asymmetrische Methode zur Baumdurchsuche, wobei in jeder Iteration jeweils die Knoten beliebiger Tiefe mit hohen Bewertungen vorrangig behandelt werden (vgl. [12], *1.1. Overview*). Der MCTS-Algorithmus besteht aus 4 Schritten, die solange wiederholt werden, bis die maximale Anzahl der Iterationen oder eine ähnliche Terminierungsbedingung erreicht ist:

1. *Selection*: Die bestbewerteten Knoten werden gemäß einer gut geeigneten Auswahlstrategie rekursiv ausgewählt, bis ein Blattknoten erreicht wird.

2. *Expansion*: Ein neuer Knoten wird generiert, der noch nicht im Suchbaum untersucht worden ist.

3. *Simulation*: Der neu expandierte Spielzustand wird anhand eines Zufallsexperiments ausgewertet. In einfachsten MCTS-Ansätzen wird eine zufällige Zugfolge bis zu einem Blattknoten durchgeführt und dann die Bewertung eingeholt.

4. *Backpropagation*: Die aus der Simulation erzielten Ergebnisse werden nach oben hin propagiert und jeweils die Anzahl der besuchten Kindknoten und die erwartete Bewertung aller Vorgängerknoten aktualisiert. Je nach der Anwendung können die Bewertungen nach jeder Update mit einem Discountfaktor multipliziert oder auch negiert werden.

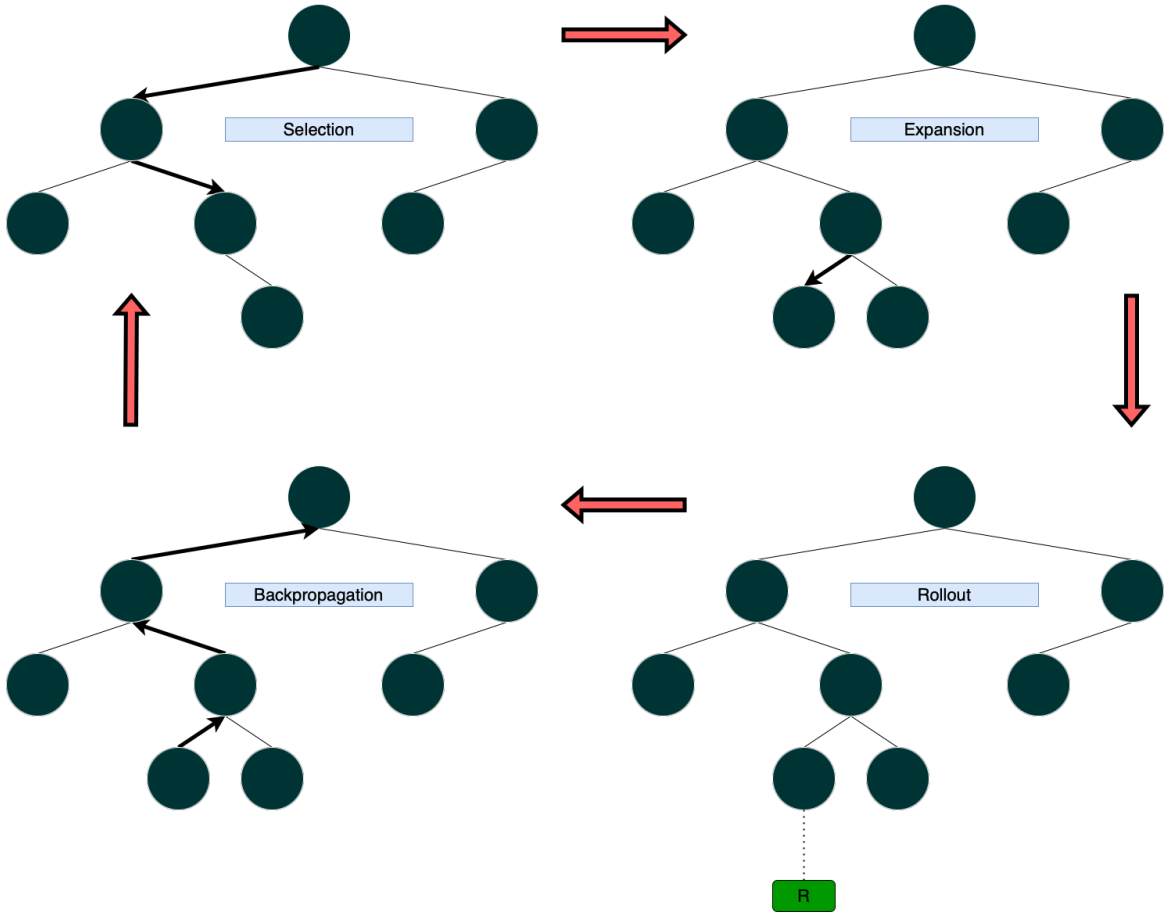


Abbildung 1.2.: Eine Iteration von Monte-Carlo-Tree-Search

Eine Policy mit reinem Fokus auf Exploitation würde danach streben, die Knoten zu bevorzugen, die die höchsten Bewertungen anbieten, und dabei den Erwartungswert der Bewertungen zu maximieren. Dieses Verhalten würde aber dazu führen, dass andere noch nicht entdeckte Knoten mit hohen Bewertungen nur selten entdeckt werden könnten. Um dieses Problem zu beheben, wird zusätzlich ein Explorationsterm in eine Policy eingefügt, die die Entropie (Uninformiertheit, Unsicherheit) (vgl. [34], *3. Characterization*) minimiert und die Knoten begünstigt, die noch nicht genug untersucht

worden sind. Upper Confidential Bound (UCB) ist eine sehr häufig benutzte Policy, die den Konflikt zwischen Exploration und Exploitation in MCTS-Anwendungen auflöst und somit das Multi-Armed-Bandit-Problem ([27], *S. 123*) löst. Konventionell werden die MCTS-Algorithmen mit UCB als Policy speziell nach UCT (Upper Confidential Bound Applied to Trees) benannt ([24], *2. The UCT Algorithm*). Die UCB-Formel mit einer festgelegten Explorationsrate c wird wie folgt definiert:

$$UCT_i = \frac{Q_i}{n_i} + c \cdot \sqrt{\frac{2 \ln N}{n_i}} \tag{1.1}$$

wobei Q_i bzw. n_i die jeweilige Summe der Bewertungen bzw. Anzahl der Besuche des Knotens i und N die Anzahl der Besuche des Elternknotens ist.

MCTS ist - im Gegensatz zu Alpha-Beta-Varianten - ein Anytime-Algorithmus und kann jederzeit eine Lösung liefern, die je nach der Anzahl der Iterationen eine derartig hohe Qualität aufweist. Die Vermutung liegt nahe, dass ein MCTS-Suchbaum eventuell nach genügend vielen Iterationen zum wahren Minimax-Suchbaum konvergiert (vgl. [25], *1. Introduction*). Es spricht für MCTS, dass der Kompromiss zwischen Laufzeit und Lösungsqualität empirisch je nach dem Fall optimiert werden kann.

1.2.3. Verbesserung von MCTS

Theoretisch haben pure MCTS-Varianten wie z. B. UCT ein zeitliches Vorteil gegenüber Alpha-Beta-Varianten, wenn z. B. unter Zeitdruck gespielt wird oder der Rechner über begrenzte Ressourcen verfügt. In der Praxis sind jedoch pure MCTS-Varianten durch eine hohe Zeitkomplexität charakterisiert, ähnlich wie Alpha-Beta-Varianten, insbesondere wenn die Suchtiefe und der Branchfaktor groß ist. In solchen Fällen kann eine zeitliche Verbesserung erreicht werden, indem Expansions- bzw. Simulationsschritte neu angepasst werden.

Es besteht möglicherweise ein Bedarf an Verbesserung des Simulationsschritts, wenn die Ergebnisse der Simulationen nur selten der Wahrheit entsprechen (wegen hohen Branchfaktors) oder die durchgeführten Simulationen zeitlich problematisch sein können (wegen hoher Suchtiefe).

In einigen Anwendungen könnte dieser Schritt durch eine Evaluierungsfunktion ersetzt werden, die in polynomieller Zeit lösbar ist. Die Implementierung einer solchen Funktion ist je nach der Anwendung unterschiedlich. Beispielsweise kann an dieser Stelle eine a-priori Expertenheuristik benutzt werden, die die Anzahl und die Dringlichkeit der Threats (Drohungen) betrachtet.

Alternativ können verschiedene Machine-Learning-Modelle dazu trainiert werden, die wahre Bewertungsfunktion nachzubilden. Am häufigsten werden flache Neuronale Netze bevorzugt, teils weil sie eine hohe Informationskapazität aufweisen und kaum Fachwissen brauchen.

Im Falle großer Suchräume kann auf ähnliche Weise eine Policy-Funktion erlernt und in der Expansion-Phase benutzt werden, um die vielversprechenden Knoten vorab zu generieren, insbesondere wenn die Simulationsergebnisse der Knoten nur wenig zielführend sind. Hierfür ist AlphaZero ([43], *15.4. AlphaZero*) ein ausgezeichnetes Beispiel.

1.2.4. ADP

Die Evaluierungsfunktionen können in MCTS-Simulationen als Ersatz für zufällige Stichproben angewendet werden, um überall zeitliche und leistungsmäßige Verbesserungen zu erzielen. Die Bildung (oder das Training) einer solchen Funktion setzt jedoch einige kritische Designentscheidungen voraus.

Beispielsweise würde es vollkommen ausreichen, möglicherweise einen a-priori-polynomiellen Algorithmus einzubauen, der (nahezu) perfekt informierte Bewertungen für beliebige Zustände liefern könnte. Dieser Algorithmus wäre dazu fähig, anhand ”essenziellen” Mustern unbekannte Muster sowie die Kombinationen von bekannten Mustern wahrheitsgemäß zu bewerten. Jedoch ist es in Gomoku unwahrscheinlich, dass ein solcher Algorithmus entdeckt wird (vgl. [39], *6. Conclusion*). Stattdessen können verschiedene Reinforcement-Learning-Ansätze zum Einsatz kommen, um die Werte der Zustände ausschließlich durch Selbstspieldaten zu lernen und dabei den Bedarf an Expertenwissen zu minimieren.

Modellbasierte RL-Ansätze lernen ein explizites Umgebungsmodell, das die Endzustandsbewertungen und die Überführungswahrscheinlichkeiten einschätzt, und schätzen somit die Bewertung bzw. optimale Policy eines Zustands anhand Dynamischer Programmierung bzw. Planung (vgl. [30], *1. Introduction*). Die Bellman-Gleichung gibt die Optimalitätsbedingung vor, die besagt, dass eine optimale Strategie immer optimale Aktionen vorschlägt, unabhängig von dem Spielzustand, in dem das Spiel gespielt wird [2], und ruft dabei die Notwendigkeit von Dynamischer Programmierung hervor (vgl. [41], *1. Introduction*).

Aufgrund der ”Curse of Dimensionality” (oder hohen Anzahl von Dimensionen) ist es jedoch in der Praxis nicht sinnvoll, Dynamische Programmierung einzusetzen, da die Anzahl der Rückwärtsberechnungen direkt mit der hohen Anzahl der möglichen Züge skaliert (vgl. [32], *2. ADP Modeling*). Mit anderen Worten eignet sich ein solcher Ansatz im Falle von Gomoku nicht, da die Anzahl und Komplexität der Zustände zu hoch ist.

Statt einer expliziten Modellbildung versuchen modellfreie RL-Ansätze, die optimalen Werte der Zustände direkt aus den Selbstspiel-Erfahrungen zu erlernen (vgl. [16], *I. Introduction*). Dies ermöglicht eine anpassungsfähige Bewertungsfunktion, ohne auf ein vorab festgelegtes Umgebungsmodell angewiesen zu sein.

In jeder Iteration wird der Bellman-Error (definiert als der Abstand zwischen optimalen und aktuellen Werten) verringert (aber nicht erhöht), somit können die optimalen Werte beliebig annähert werden. Beispielsweise ist ein Q-Learning-Modell (bzw. TD-Learning-Modell) mit einer geeigneten Lernrate dazu fähig, nach ausreichender Zahl an Iterationen optimale Ergebnisse zu liefern wie Q-Iteration (bzw. Value-Iteration) (vgl. [16], *4. Discussions and Conclusions*).

In Gomoku ist eine diskrete Zustand-Bewertung-Zuordnung nicht erwünscht, weil aus den erlernten Ergebnissen keine allgemeine Rückschlüsse gezogen werden können. Damit die gut bewerteten Zustände nicht aussortiert werden, die noch nicht abgedeckt sind, müssten alle (redundanten) Zustände abgedeckt werden. Alternativerweise können direkt die interessanten lokalen Merkmale aus Zuständen gelernt werden, die einen Großteil der interessanten Informationen bezüglich Spielzustän-

den repräsentieren.

Adaptive (oder Approximative) Dynamische Programmierung (kurz ADP) umfasst eine Reihe von sowohl modellbasierten als auch modellfreien RL-Ansätzen, die - bewusst die Relaxierung der Optimalität in Kauf nehmend - effizient Approximationsfunktionen mit polynomieller Laufzeit erlernen (vgl. [32], *2. ADP Modeling*). Typischerweise werden an dieser Stelle Neuronale Netze als universelle Funktionsapproximatoren verwendet, die üblicherweise eine hohe Informationskapazität und Leistung in ML-Aufgaben aufweisen.

Ein nicht-gieriger ADP-Spieler wählt immer den Spielzug, der die prognostizierten Bewertungen eines Spielzustandes maximiert (bzw. minimiert) und somit repräsentiert eine Bestensuche, was aber dazu führen könnte, dass die tatsächlich besten Züge übersehen werden. Um die Explorationsrate abzustimmen, kann ein gieriger Ansatz eingesetzt werden, der mit einer Wahrscheinlichkeit $\varepsilon \in]0, 1[$ eine zufällige Zugentscheidung trifft.

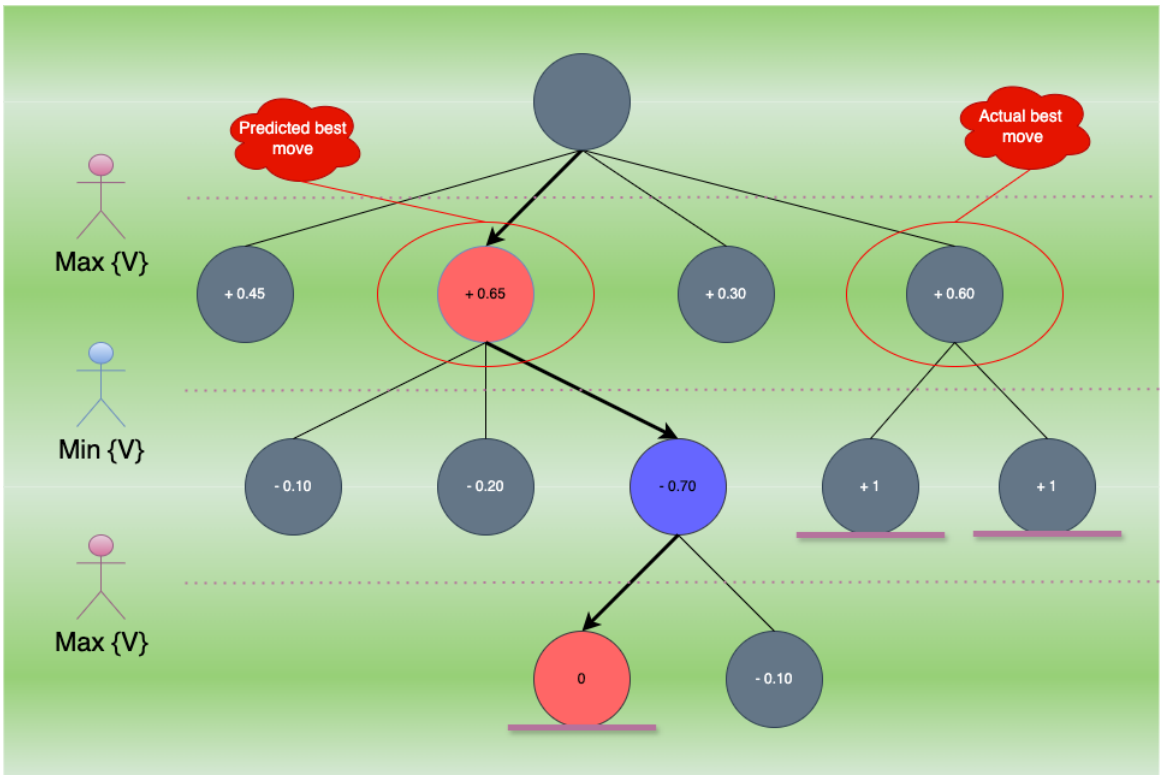


Abbildung 1.3.: Die Limitation eines nicht-gierigen ADP-Ansatzes

ADP, Zhao et al.

In der vorhandenen Literatur ist eine ADP-Anwendung in Gomoku [44] zu finden, die grundlegende Designentscheidungen für spätere Hybridansätze festlegt, die im Abschnitt 1.2.5 vorgestellt werden. Das wichtigste Bekenntnis dieses Projekts ist, dass ein wettbewerbsfähiges ST-Gomoku-Modell für ein Spiel der Größe 15×15 mit ausschließlich 60000 Trainingsepochen durch Selbstspiele entwickelt werden kann (vgl. [44], *4.3. Discussion*). Außerdem würden selbstlernende Modelle, die an-

hand Selbstspieldaten trainiert werden, eine höhere Leistung als beobachtende Modelle erzielen, die auf Basis von Expertenpartien trainiert werden (vgl. [44], *4.2. Comparison*). Die Ergebnisse aus dem Projekt heben zudem die Notwendigkeit der (manuellen) Merkmalsextraktion hervor (siehe Anhang A.3), da die Nutzung von Rohdaten zur Enkodierung der Spielzustände deutlich schlechte Leistungen nach sich zieht (vgl. [44], *4.1. Comparison*).

1.2.5. Gemischte Ansätze

Alpha-Beta-Ansätze liefern zwar immer die bestmöglichen Strategien, jedoch mit der höchsten Zeitkomplexität unter allen, da es eine vollständige Baumsuche erfordert.

MCTS-Ansätze sind dazu fähig, mit deutlich geringerem Zeitaufwand eine hohe Lösungsqualität wie Alpha-Beta-Varianten zu erzielen, wenn genügend viele Iterationen durchgeführt werden. Trotzdem kann ein naiver Simulationsschritt eventuell zu Verschlechterungen führen (sowohl zeitlich als auch leistungsmäßig).

ADP-Ansätze sind durchaus effizienter als andere und besitzen sogar eine polynomielle Worst-Case-Laufzeit. Wenn mit genug Spielen trainiert, können sie in der Praxis die bestbewerteten Zuständen erkennen, weil Gesamtwissen aus früheren Spieldaten als die gelernten Modellparameter enkodiert worden sind. Reine ADP-Ansätze sind aber - wegen ihrer probabilistischen Natur - selber nicht dafür geeignet, endgültige Aussagen über die Werte der Spielzustände zu liefern.

Erwünscht ist eine geeignete Mischung von den erwähnten Ansätzen, um eine hohe Leistung und eine niedrige Zeitkomplexität unter einen Hut zu bringen. Im Laufe der Entwicklungen im Bereich Reinforcement-Learning wurden auch verschiedene Lösungsansätze ([40], [13]) vorgeschlagen, die MCTS-Baumsuchen mit ADP-Vorhersagen kombinieren und dabei zu sogenannten Leistungs- und Laufzeitverbesserungen führen.

UCT + ADP, Tang et al.

MCTS erfordert eine exponentielle Worst-Case-Laufzeit, ADP kann hingegen die Optimalität nicht garantieren. Deshalb wird eine Mischung aus MCTS und ADP [40] vorgeschlagen.

Am Anfang benennt die ADP-Funktion $k = 5$ bestbewertete Lösungskandidaten mit jeweiligen Gewinnwahrscheinlichkeiten W_{ADP}. Danach werden diese Knoten jeweils in eine MCTS-Instanz als Wurzelknoten eingegeben, die am Ende die Gewinnwahrscheinlichkeit des jeweiligen Knotens W_{MCTS} zurückgibt. Danach wird eine gewichtete Summe beider Gewinnwahrscheinlichkeiten W_{MCTS}+ $C \cdot W_{ADP}$ mit $C(= 1)$ berechnet. Schließlich wird der Knoten mit der höchsten Bewertung ausgewählt.

Dieser Ansatz kann zwar hohe Leistungen in spezifischen Anwendungsfällen wie Gomoku erzielen, aber er kann vermutlich nicht auf allen Fällen angewendet werden, ohne einige manuellen Anpassungen im Design vorzunehmen. Dieser Ansatz beruht bei der Merkmalsextraktion auf traditionelle Mustererkennung, weshalb es unvermeidbar ist, dass die Sichtweise des Algorithmus durch Expertenwissen begrenzt ist.

Zudem geht die flache Analyse eines Zustands implizit davon aus, dass die optimale Handlung stets unter den k am höchsten bewerteten Aktionen des ADP-Modells zu finden sein sollte. Daher tritt die Problematik der Bestensuche mit abnehmender k immer häufiger.

UCT + ADP + PB, Cao et al.

Es wird behauptet, der Ansatz von Tang et al. [40] erziele zwar eine Leistungsverbesserung im Vergleich zu ADP oder UCT, jedoch habe sich die Laufzeit des Algorithmus wegen MCTS-Anteil nicht verbessert (vgl. [13], *IV. UCT-ADP-PB Algorithm*). Deshalb wird ein ähnliches Konzept wie AlphaGo vorgeschlagen, so dass die MCTS-Simulationen durch ADP ersetzt werden.

In diesem Ansatz wird die Auswahlsstrategie von MCTS auf vielfältige Weise verbessert. Das Erkenntnis aus statischen Threat-Space-Analysen [14], dass die Züge, die mehr als zwei Gitter entfernt von einer Threat sind, aussortiert werden können (vgl. [13], *IV. MCTS with ADP*), führt zur Idee der 2-Adjazenzgitter. Zusätzlich wird die Auswahlstrategie neu angepasst, so dass eine Progressive-Bias in die UCB-Formel hinzugenommen wird. Progressive Bias ist nämlich eine heuristische Funktion, die angibt, wie "interessant" der Spielzustand für den aktuellen Spieler ist. Da die heuristische Funktion ([21], *Formula (3)*) bei der Erkennung einiger Grenzfälle scheitert, wurde als Gegenvorschlag eine neue exponentielle heuristische Funktion vorgeschlagen ([13], *Formula (3)*). Dank der verbesserten Heuristik ist das neue Modell besonders in Situationen mit begrenzter Anzahl von MCTS-Simulationen in der Lage, besser bewertete Aktionen zu identifizieren und diesen Vorrang einzuräumen.

Ähnlicherweise werden die Nachteile von UCT + ADP von Tang et al. weiter an diesem Ansatz vererbt, wie z. B. manuelle Merkmalsextraktion. Ferner werden einige Designentscheidungen getroffen, die die spezifischen Spielregeln von Gomoku möglicherweise ausnutzt, um die höhere Leistung eines feinabgestimmten Algorithmus zu rechtfertigen, wie z. B. 2-Adjazenzgitter oder Progressive Bias.

Schließlich kommen beide Arten von Algorithmen vermutlich nur in einer relativ kleinen Untermenge von kombinatorischen Spielen wie z. B. Go oder Checkers zunutze, besonders wenn die Spielregeln gewisse Ähnlichkeiten besitzen. In ganz anderen Anwendungsfällen wie z. B. Hex (wobei das Spielbrett ein Parallelogramm ist), sollten beispielsweise die Kernkonzepte wie Adjazenz erneut ausgelegt und die Programme neu implementiert werden. Daher sind neue lernfähigere Ansätze wie z. B. AlphaZero wünschenswert, die auf möglichst wenig Annahmen beruhen, wie im Folgenden vorgestellt wird.

1.2.6. AlphaZero

AlphaGo ist ein Algorithmus, der dadurch bekannt ist, im Jahre 2016 den weltbesten Spieler, Lee Sedol, besiegt zu haben und somit zum ersten Mal in der Geschichte die Überlegenheit von KI in Go bewiesen zu haben (vgl. [10]). Die hohe Leistung von AlphaGo, die über dem menschlichen Niveau liegt, resultiert daraus, dass das Modell mit Millionen von Spielpartien trainiert wurde.

Analog zu einem Erkenntnis aus dem Unterabschnitt 1.2.4, dass die selbstlernenden ADP-Modelle höhere Leistungen als die beobachtenden erbracht haben, wurde im Jahr 2017 eine selbstlernende Variante von AlphaGo vorgeschlagen: AlphaGo Zero. Wie es aus dem Namen ”Zero” selbstverständlich ist, ist dieser Algorithmus ohne menschliche Interaktion fähig dazu, ausschließlich durch Selbstspiele die Bewertung und die optimale Policy der Spielzustände in Go zu lernen. Nach fünf Millionen Spielpartien gegen sich selbst ist es dem AlphaGo Zero gelungen, bisherige AlphaGo-Modelle zu besiegen (vgl. [17], *S. 83*).

AlphaZero ist ein verallgemeinerter, spielunabhängiger Algorithmus von AlphaGo Zero, der in verschiedenen kombinatorischen Spielen wie Shogu, Schach (oder in diesem Fall Gomoku) angewendet werden kann (vgl. [43], *15.4. AlphaZero*). Der Algorithmus verdankt sich seine hohe Leistung einer verbesserten Version von MCTS mit NN als Ersatz für zufallsbedingte Expansions- sowie Simulationsschritte (wie im Teil 1.2.3. beschrieben).

In AlphaZero werden - im Gegensatz zu domänenspezifischen Algorithmen mit rechenintensiver Merkmalsextraktion - möglichst einfache und redundante Eingabeformate bevorzugt (wie z. B. das gesamte Spielbrett). Angenommen, dass das menschliche Expertenwissen nicht ausreicht, ein Spielzustand vollständig zu beschreiben, werden die wichtigsten Merkmale beispielsweise durch Konvolutionelle Schichten neu gelernt. Allerdings müssen meist ein hoher Rechenaufwand, längere Trainingszeiten und größere Datenmengen in Kauf genommen werden.

Im einem Artikel [26] werden die mathematischen Grundlagen von AlphaZero erklärt sowie, wie ein AlphaZero-Modell trainiert werden kann und welche Ergebnisse zu erwarten sind. Für weitere Informationen zu Trainingsdetails und -ergebnissen kann eine entsprechende Implementierung von AlphaZero [1] als Grundlage verwendet werden.

1.3. AlphaZero gegen MCTS in Gomoku

Diese Arbeit kreist sich hauptsächlich um die Forschungsfrage, inwiefern AlphaZero-Ansätzen bei der Auswertung und Strategiefindung in Gomoku-Spielen besser als MCTS-Ansätzen funktionieren.

Im Kapitel 2 werden die wichtigsten Methoden und deren Implementierungen vorgestellt sowie einige Designentscheidungen, die beim Erstellen eines AlphaZero-Modells getroffen werden. Zusätzlich werden die wichtigsten Evaluierungsmetriken erklärt.

Im Kapitel 3 werden einige Benchmarks bezüglich Laufzeit, Ressourcenaufwand sowie Leistung veröffentlicht, die anhand Log-Dateien und Zweikämpfe von AlphaZero- und MCTS-Varianten erfasst werden.

Im Kapitel 4 werden die Ergebnisse von Benchmarks in einem breiten Spektrum diskutiert, und dafür angewendet, um die Frage zu beantworten, welche Ansätze in welchen Umständen eine bessere Leistung aufweisen, und wie der Ansatz noch verbessert werden könnte.

2. Methoden

In diesem Kapitel werden die Struktur des Projekts sowie die wichtigsten Methoden und Klassen präsentiert. Als Nächstes werden die Trainingsdetails, die Hyperparameter und die Evaluationsmetriken mit ausführlichen Beschreibungen aufgelistet, die für den Kapitel 3 relevant sind.

2.1. Projektstruktur

Das Programm [6] weist im Groben eine folgende Ordnerstruktur auf:

```
root/
├── src/
├── 6_6_4/
│   ├── train.log
│   ├── models/
│   │   ├── best.pkl
│   │   ├── curr.pkl
│   │   └── <weitere Modelle ...>
│   ├── competition/
│   ├── timeseries/
│   └── <Bilder ...>
├── 8_8_5/
└── 10_10_5/
```

Alle Python-Skripten unter dem Verzeichnis *src/*, das die Gesamtfunktionalität des Projekts beinhaltet, werden wie folgt ausführlich erklärt:

- *src/calc.py*: In diesem Skript werden eine Liste von nützlichen mathematischen Funktionen definiert, die im Rest des Programms häufig benutzt werden, insbesondere in Neuronalen Netzen und AlphaZero-Training.
- *src/gomoku.py*: Hier wird eine Klasse namens *Gomoku* implementiert, die eine *State*-Interface nach Gomoku-Regeln implementiert und somit die spielspezifischen Details weg abstrahiert.
- *src/player.py*: Hier wird eine abstrakte Klasse für Gomoku-Spieler deklariert, die über einige Methoden zur Vereinfachung der Zugdurchführung wie z. B. *next_move* verfügt.
- *src/data.py*: Hier werden einige Hilfsfunktionen implementiert, die von der Durchführung von Spielen bis hin zur Erfassung von Spielerdaten reichen.
- *src/net.py*: Die Klasse *Zero_Net*, die auf einem zugrundeliegenden CNN basiert, umfasst Methoden zum Lernen der Zustandswerte sowie der optimalen Policy.

- *src/mcts.py*: Eine allgemeine MCTS-Klasse namens *Deep_Player* wird implementiert, die später entweder als UCT oder als AlphaZero konfiguriert werden kann.
- *src/train.py*: Dieses Skript enthält die Trainingspipeline für das Trainieren von AlphaZero gegen MCTS.
- *src/comp.py*: In diesem Skript befinden sich Methoden zur Evaluierung von verschiedenen Spielern anhand paarweisen Vergleichen.
- *src/stats.py*: Diese Datei enthält Pipelines zum Testen von Evaluierungshypothesen sowie eine Liste von Funktionen zur Visualisierung der Ergebnisse.

2.2. Maschinelles Lernen

Maschinelles Lernen (abgekürzt als ML) ist ein Feld von Künstlicher Intelligenz, inspiriert von der menschlichen Biologie, das grundsätzlich die Lernfähigkeit von Computer-Programmen forscht. Im Gegensatz zu traditionellen Algorithmen mit vordefinierten Regeln passen sich ML-Modelle dynamisch an neue Daten an, welches eine breitere Anwendung ermöglicht wie z. B. Vorhersageanalysen, Bilderkennung (Computer Vision) und Sprachverarbeitung (NLP).

Ein ML-Modell sei im Allgemeinen in der Lage, anhand Leistungsmetriken und Erfahrungen eine bestimmte Aufgabe ausschließlich durch Minimierung der Fehler zu lernen, ohne dafür explizit programmiert zu werden (vgl. [29], *1. Defining Questions*). Im Falle von komplexen Datensätzen, die über das menschliche Verständnis hinausgehen, ist ein solcher Ansatz besonders praktikabel.

2.2.1. Neuronale Netzwerke (NNs)

Traditionelle ML-Modelle, wie z. B. Lineare Regression [19] und Support-Vector-Machines (SVM) [28], beruhen oft auf einer Vielzahl von Annahmen über Datenverteilungen oder -strukturen. Im Gegensatz dazu bieten Neuronale Netzwerke (NNs) einen flexibleren Ansatz und sind in der Lage, komplexe Muster in den Daten ohne die strengen Voraussetzungen traditioneller Modelle zu erkennen.

Inspiriert von der Struktur des menschlichen Gehirns, beruhen NNs grundsätzlich auf künstliche Neuronen, die jeweils einen Eingabevektor einnehmen, eine gewichtete Summe davon berechnen, eine Verzerrung dazu addieren, und zum Schluss das Ergebnis anhand einer nichtlinearen Aktivierungsfunktion transformieren. Die im Laufe dieser Arbeit verwendeten Aktivierungsfunktionen sind:

- Rectified Linear Unit: $\mathrm{ReLU}(x) = \max(0, x) \geq 0$
- Sigmoid $\sigma(x) = \frac{1}{1+e^{-x}} \in [0, 1]$
- Tanh: $\tanh(x) = \frac{e^{x}-e^{-x}}{e^{x}+e^{-x}} \in [-1, 1]$
- Softmax: $s(x_i) = \frac{e^{x_i}}{\sum_{j=1}^{n} e^{x_j}} \in [0, 1]$ (mit $\sum_{i=1}^{n} s(x_i) = 1$)

Die NN-Modelle bestehen typischerweise aus hintereinander ausgeführten Schichten, die jeweils über eine bestimmte Anzahl von Neuronen verfügen, die denselben Eingabevektor verarbeiten und jeweils ein verschiedenes Merkmal entdecken.

2.2.2. Konvolutionelle Neuronale Netzwerke (CNNs)

Die ML-Modelle sind grundsätzlich mathematische Funktionen, deshalb müssen die komplexen Datenstrukturen wie bspw. ein Spielzustand in Merkmalsvektoren umgewandelt werden, bevor die von ML-Modellen zur Mustererkennung oder klassifizierung genutzt werden können. Manuelle Merkmalsextraktion, wie z. B. musterbasierte Enkodierung von einem Spielbrett, basiert auf dem begrenzten und subjektiven menschlichen Wissen und ist somit fehleranfällig.

Inspiriert von der Art und Weise, wie die Menschen Bilder verarbeiten und interpretieren, sind die Konvolutionellen Neuronalen Netzwerke (CNNs) [31] hingegen eine besondere Art von Neuronalen Netzen, die die Merkmalsextraktion automatisiert. Diese Netze nutzen spezielle Schichten namens "Konvolutionelle Schichten", um lokale Muster wie z. B. Kanten, Texturen und Formen zu erkennen. Durch die schrittweise Abstraktion solcher Merkmale können CNNs komplexe visuelle Konzepte in tieferen Schichten entschlüsseln und genauere Vorhersagen treffen.

2.2.3. Eine CNN-Architektur von AlphaZero

In der aktuellen Implementierung werden die folgenden Informationen eines Spielzustandes anfangs jeweils in einer Matrix der Größe $M \times N$ binär enkodiert: (1) Die bisherigen Spielzüge von dem ersten Spieler und (2) die von dem zweiten Spieler, (3) die Position des letzten Zuges sowie (4) der aktuelle Spieler. Hierbei ist es zu beachten, dass die redundanten Filter (3) bzw. (4) nur zwecks Ausführlichkeit (und nicht zur Notwendigkeit) verwendet werden: In der Theorie könnte die Information (4) aus der Anzahl der Züge gewonnen werden und sogar die Information (3) verworfen werden, was jedoch möglicherweise den Orientierungsverlust der Spieler zur Folge hätte.

Im Anschluss an die manuelle Merkmalsextraktion wird der Eingangstensor der Größe $4 \times M \times N$ hintereinander durch drei konvolutionelle Schichten geführt. Diese Schichten erhöhen die Anzahl der Filter jeweils um das Doppelte oder Vierfache, um so eine stufenweisen Abstraktion der Merkmalsextraktion zu ermöglichen. Anschließend wird das erzeugte Zwischenergebnis auf zwei separate Teilmodelle aufgeteilt. Jedes dieser Modelle leitet das Ergebnis zunächst durch eine konvolutionelle Schicht mit weniger Filteranzahl, formt es dann in einen Vektor um, gibt diesen an ein (dichtes) neuronales Netzwerk weiter und wendet schließlich eine spezifische Aktivierungsfunktion zur Transformation des Endergebnisses an. Das erste Teilmodell verwendet die logarithmische Softmax-Funktion als finale Aktivierung und sagt die Log-Wahrscheinlichkeiten für die Spielzüge vorher, repräsentiert durch einen Vektor der Größe $M \times N$. Das zweite Teilmodell nutzt die Tanh-Funktion als Endaktivierung, um den Wert eines Spielzustands zu schätzen. Standardmäßig wird als Aktivierungsfunktionen in den vorherigen Schichten ausschließlich ReLU verwendet.

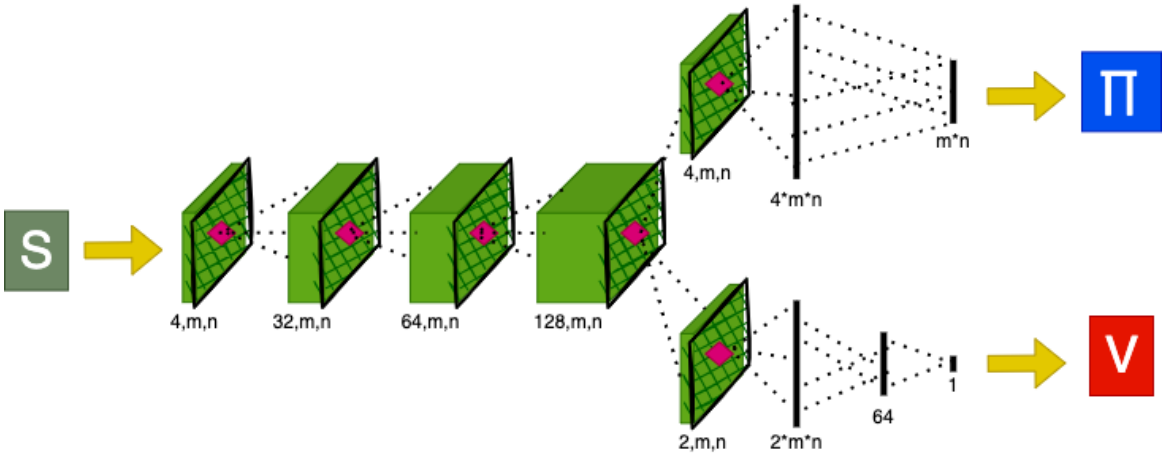


Abbildung 2.1.: Gesamt-Architektur von einem CNN-Modell variabler Größe

2.3. Zero und UCT

In diesem Teil werden die wichtigsten Methoden in verschiedenen Sinnabschnitten vorgestellt.

Sei *State* eine Datenstruktur, die über die folgenden Schnittstellen verfügt:

- *play*: Eine gültige Aktion wird ausgeführt und folglich wird die Auswertung des neuen Zustands zurückgegeben.
- *fin*: Es wird überprüft, ob der gegebene Zustand ein Endzustand ist.
- *actions*: Im Falle, dass der Zustand nicht ein Endzustand ist, werden alle gültigen Aktionen zurückgegeben.

2.3.1. MCTS-Iteration

Im folgenden Codeabschnitt wird eine möglichst allgemeine MCTS-Iteration vorgestellt, die beliebig oft wiederholt werden kann:

Algorithm 1 MCTS.iterate(*state:* State)

```
    Require: not state.fin()
node: Node ← root
reward: int ← 0                                  ▷ Beginne mit Nullauswertung
while not node.leaf() do
    (action, node) ← select(node)
    reward ← state.play(action)                  ▷ Zustand wird direkt aktualisiert
end while
player: int ← state.player
if not state.fin() then
    reward ← simulate(state)
end if
reward ← reward · player                         ▷ Ist es der Zug vom zweiten Spieler, dann negiere
backpropagate(node, −reward)                     ▷ Rekursiver Aufruf mit negativem Vorzeichen
```

Zusammenfassung: Beginnend vom Wurzelknoten, wird immer dem Knoten mit der höchsten UCB-Bewertung navigiert, bis ein Blattknoten erreicht wird. Wenn ein Endzustand erreicht ist, ist die Bewertung bereits als *reward* zwischengespeichert. Sonst wird ein Simulationsschritt *simulate* ausgeführt, der die Bewertung des Zustands sowie die Expandierungswahrscheinlichkeiten zurückgibt, und ein Nachfolgerknoten wird generiert. Schließlich wird die Bewertung vom Blattknoten nach oben hin propagiert.

2.3.2. UCT-Simulation

Als Erstes wird der Evaluierungsschritt in UCT wie Folgendes implementiert:

Algorithm 2 UCT.simulate(*state:* State, *node:* Node)

```
    Require: not state.fin()
    Ensure state.fin(), uniform probs
actions: [Act] ← state.actions()
probs: [float] ← {1/size(actions), ..., 1/size(actions)}
node ← expand(node, (probs, actions))            ▷ Expandiere maximal uninformativ
reward: int ← 0
while not state.fin() do
    action ~ random(state.actions())             ▷ Wähle einen zufalligen unbesetzten Zug
    reward ← state.play(action)
end while
return reward
```

Zusammenfassung: Die Expandierungswahrscheinlichkeiten von allen möglichen Aktionen werden uniform gesetzt, so dass alle Nachfolgerknoten mit derselben Wahrscheinlichkeit generiert werden können. Dem Knoten wird der neu expandierte Kindknoten zugewiesen. Die aktuelle Partie wird mit

zufälligen Zügen bis zum Ende gespielt und die Bewertung herausgeholt.

Monte-Carlo-Algorithmen beruhen auf der Annahme, dass die Sample nach einer ausreichenden Zahl an Iterationen die zugrundeliegende Population nahezu beliebig genau approximieren kann. Daher sollten die Extremwerte der Rollout-Ergebnisse nach mehreren Iterationen weniger Einfluss haben.

2.3.3. AlphaZero-Simulation

Als Zweites wird die Implementierung vom Evaluierungsschritt in AlphaZero dargestellt:

Algorithm 3 Zero.simulate(*state:* State, *node:* Node)

```
    Require: not state.fin()
  actions: [Act] ← state.actions()
  features: [Vector<float>] ← state.encode()        ▷ Spielzüge, aktueller Spieler, letzter Zug
  (log_probs, reward) ← net.predict(features)       ▷ forward-Methode vom CNN-Modell
  probs: [float] ← exp(log_probs)
  node ← expand(node, (probs, actions))
  return reward
```

Zusammenfassung: Wichtige Informationen von dem vorliegenden Zustand werden als eine Liste von Vektoren enkodiert und in das CNN-Modell eingegeben, um wiederum die Log-Wahrscheinlichkeiten sowie die erwartete Bewertung des Zustandes herauszubekommen. Die Expand-Wahrscheinlichkeiten der Knoten, die sich aus den Exponenten von Log-Wahrscheinlichkeiten zur natürlichen Basis ergeben, werden zum Expandierungsschritt verwendet.

Im Gegensatz zum Algorithmus 2 ist in dieser Implementierung keine **while**- oder **for**-Schleife zu sehen. Daher bleibt die Laufzeit dieses Evaluierungsschritts konstant unabhängig von dem Spielzustand.

2.3.4. Wiederverwendung von MCTS-Bäumen

Die Vermutung liegt nahe, dass ein MCTS-Algorithmus umso klügere Entscheidungen trifft, je höher die Anzahl der MCTS-Iterationen ist. Daher ist es wünschenswert, die bereits generierten MCTS-Bäume wiederzuverwenden.

tree_memory: Falls im Laufe einer Spielpartie eine bestimmte Spielposition bereits untersucht wurde, wird zu dem Knoten navigiert und von da an MCTS fortgeführt, statt frisch neuzustarten. Dank der folgenden Anpassung ist ein MCTS-Algorithmus in der Lage, mehr Iterationen pro Zug pro Zeiteinheit zu erzielen und eventuell die Simulationsergebnisse zu stabilisieren.

Wenn *tree_memory* = *True* gilt, wird die folgende Methode aufgerufen:

Algorithm 4 MCTS.reuse_tree(*state:* State, *tree:* Tree, *cached_history:* [Act])

Require: *tree_memory = True, not state.fin(), single-thread MCTS*

root ← *tree.root*
reused ← *False*
if *state.history* ⊃ *cached_history* **then** ▷ *cached_history* stimmt mit *state* überein
  rest_history ← *state.history* \ *cached_history*
  if *found_path(root, rest_history)* **then** ▷ *state* wurde bereits untersucht
    for *action* ∈ *rest_history* **do** ▷ Setze Knoten von *state* als neuen Wurzel
      root ← *root.select(action)*
      root.parent ← ⊥
    end for
    tree.root ← *root*
    reused ← *True*
  end if
end if
if *not reused* **then**
  tree.reset()
end if

Zusammenfassung: Erstens wird es überprüft, ob die zwischengespeicherte Spielhistorie mit der aktuellen Historie übereinstimmt. Wenn der aktuelle Spielzustand bereits in MCTS-Baum untersucht worden ist, muss es genau einen Knoten geben, der den aktuellen Spielzustand vertritt. Dieser Knoten wird zum neuen Wurzel erklärt, und sein Elternknoten werden aus dem Baum entfernt. Wenn einer der Bedingungen nicht erfüllt ist, wird der Baum neu initialisiert.

2.3.5. Flat: Flacher Spieler

Sei ein zugrundeliegendes CNN-Modell eines Zero-Spielers ausreichend trainiert, dann wird es angenommen, dass das Modell die optimale Policies von den (nahezu) allen Spielzuständen approximativ gelernt hat. Die Vermutung liegt nahe, dass eine Erhöhung der MCTS-Iterationen eines ausreichend trainierten Zero-Spielers die Gesamtleistung kaum verbessert. Um die Laufzeit möglichst gering und unabhängig von der Suchraumgröße zu halten und trotzdem eine ausreichend gute Leistung zu erzielen, kann im Gegenteil die Anzahl der MCTS-Iterationen reduziert werden. Im Extremfall $n = 1$ ergibt sich ein flacher Spieler mit konstanter Zeitkomplexität, der im Laufe dieser Arbeit als **Flat** (= Flach aus dem Englischen) benannt wird.

Verglichen mit ADP-Ansätzen bietet **Flat** einige wichtige Vorteile, insbesondere im zeitlichen Sinne. Während ein ADP-Ansatz alle Nachfolgerzustände im Einzelnen bewertet und anschließend sortiert, liefert **Flat** direkt eine entsprechende Aktion im eingegebenen Zustand anhand der ermittelten Policy. Außerdem wird im Falle von ADP davon ausgegangen, dass ein Spielzustand in unterschiedlichen Spielszenarien mehrfach auftreten kann, weshalb es erwünscht wird, dass alle Zustände in derselben Spielrunde t bezüglich der Bewertungen untereinander vergleichbar sind. Im Gegensatz dazu tritt diese Problematik bei **Flat** nicht auf, da die erlernten Werte dementsprechend im lokalen Umfang

der Zustand-Aktion-Paaren angepasst werden. Im übertragenen Sinne weist der Kontrast zwischen ADP-Ansatz und **Flat** gewisse Ähnlichkeiten mit dem Kontrast zwischen Value-Iteration und Policy-Iteration.

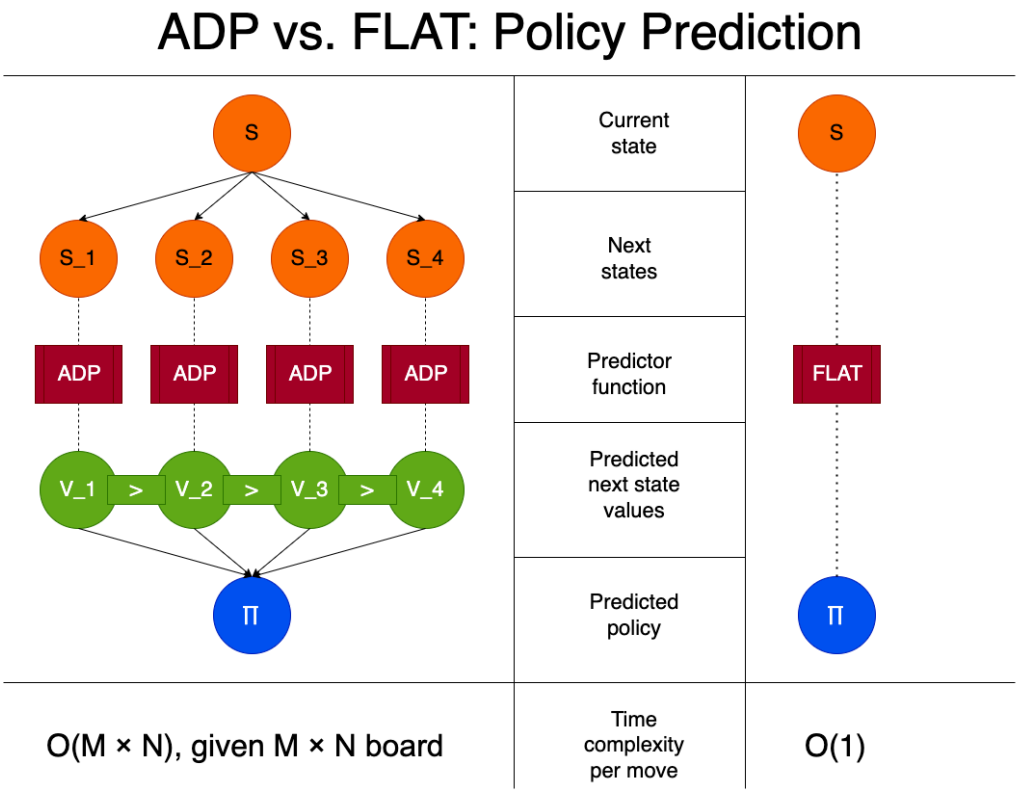


Abbildung 2.2.: Laufzeitvergleich von ADP- und Flat-Ansatz bei der Zugentscheidung

2.4. Training

Im folgenden Codeabschnitt wird die Trainingspipeline (befindlich in *src/train.py*) dargestellt. Mit der Absicht, den Fokus nur auf die allgemeine Struktur des Trainings zu legen, werden hierbei die Hyperparameter ausgeschlossen.

Algorithm 5 train(. . .)

Require: game parameters are valid, model and buffer paths are set correctly
Input: *n_batches, batch_size, n_zero, n_uct, n_duels:* nat, *perfect_win_rate:* float

zero: MCTS.Zero ← *Zero(iterations = n_zero)*
load_curr_model_if_exists(zero) ▷ Alternativ kann *best_model* geladen werden
uct: MCTS.UCT ← *UCT(iterations = n_uct)*
buffer: List<Features, Labels> ← *load_buffer_if_exists()*
best_stats: Stats ← {*win_rate* = 0, . . . } ▷ Andere Statistiken werden als . . . abgekürzt
$i \leftarrow 0$
while $i \leq$ *n_batches* **do**
    buffer ← *buffer* ∪ *one_game_data()*
    batch ← *sample(buffer, batch_size)*
    (loss, metrics) ← *fit(zero, batch)*
    if *checkpoint(i)* **then**
        save_curr_model(zero)
        stats ← *eval(zero, uct, n_duels)*
        if *stats.win_rate* > *best_stats.win_rate* **then**
            save_best_model(zero)
        end if
        if *stats.win_rate* ≥ *perfect_win_rate* **then**
            upgrade(uct, n_uct)
        else if *stats.win_rate* ≤ 1 − *perfect_win_rate* **then**
            downgrade(uct, n_uct)
        end if
    end if
    $i \leftarrow i + 1$
end while
save_buffer(buffer)

Zusammenfassung: Erstens werden die Modelle *Zero* (gegebenenfalls mit prätrainierten Modellparametern) und *UCT* initialisiert und die Hilfsvariablen wie z. B. *buffer* (Puffer zum Speichern von Batches) und *best_stats* definiert. Anschließend gerät der Algorithmus in eine **while**-Schleife, die (höchstens) *n_batches*-mal iteriert. Eine Iteration der **while**-Schleife besteht aus folgenden Schritten:

- Nachdem neue Spieldaten ins Puffer eingelegt werden, wird das AlphaZero-Modell um ein Batch trainiert und die Trainingsergebnissen werden für weitere Datenanalysen in eine Log-Datei geschrieben.

- Wird ein Evaluierung-Checkpoint erreicht, werden einige Schritte zum Vergleich des neuen Modells mit dem besten Zero-Modell ausgeführt:
  - Wenn das neue Modell eine höhere Gewinnrate erzielt als dem aktuell besten Modell, ist das beste Modell durch das Neue ersetzt.
  - Wenn das neue Modell sogar eine nahezu perfekte Gewinnrate erzielt, wird die Anzahl der Iterationen von dem UCT-Spieler und somit dessen Schwierigkeitsgrad erhöht.
  - Analog gilt für umso perfekte Verlustrate eine dementsprechende Reduzierung von der Anzahl der Iterationen.
- Schließlich wird das Puffer geschrieben, damit das Training irgendwann später fortgeführt werden kann.

Der folgende Teil befasst sich mit den Hyperparametern des Programms, die jeweils eine wichtige Rolle bei der Leistung des Modells spielen.

2.4.1. Hyperparameter

Zur Kenntnis nehmend, dass einige Hyperparameter je nach Spielgröße unterschiedliche Werte einnehmen können, sind die gewählten Spielgrößen unten aufgelistet:

- **S**mall: 6 x 6 Brettgröße, 4 Gewinnlänge
- **M**edium: 8 x 8 Brettgröße, 5 Gewinnlänge
- **L**arge: 10 x 10 Brettgröße, 5 Gewinnlänge

Unten werden alle wichtigen Hyperparameter fürs Training (gegebenenfalls in Abhängigkeit von Spielgrößen Small, Medium und Large aufgelistet:

- *lr*: Dies ist die Lernrate des zugrundeliegenden CNN-Modells, die zwischen 0 und 1 liegt, wobei der Standardwert 0.002 beträgt.
- *temp*: Die Policy-Temperatur bei der Evaluierung, die ebenfalls zwischen 0 und 1 variiert. Ein Wert nahe 0 erhöht die Wahrscheinlichkeiten der bestbewerteten Aktionen, während ein Wert nahe 1 mehr Exploration fördert.
- *epsilon*: Epsilon repräsentiert die Störungsrate der Policy-Wahrscheinlichkeiten, mit Werten im Bereich von 0 bis 1. Hierfür ist der Standardwert 0.25.
- *k_ucb*: Diese Größe steht für die UCB-Explorationsrate in der Selektionsphase. Sie ist eine positive Zahl und hat einen Standardwert von 5.
- *kl_threshold*: Diese Größe, die aus dem für dieses Projekt angewandten Algorithmus zur Lernratenanpassung stammt [37], stellt die untere bzw. obere Schranke der KL-Divergenz dar, um die Lernrate zu steigern oder zu senken. Der Standardwert ist auf 0.008 festgelegt, wobei ein höherer Wert eine verstärkte Lernrate zur Folge hat.

- *n_batch_reps*: Hiermit wird die Anzahl der Durchläufe über einen einzigen Batch bestimmt, standardmäßig auf 5 gesetzt.
- *n_epochs*: Damit wird die Anzahl der Trainingsspiele gemeint. Standardmäßig wird es auf 1000 gesetzt.
- *n_zero*: Dies beschreibt die Anzahl der Zero-Iterationen, wobei unterschiedliche Werte für verschiedene Größen vorgesehen sind: 400 für Small, 500 für Medium und 600 für Large-Spielgrößen.
- *n_uct*: Hiermit wird die Anzahl der MCTS-Iterationen festgelegt, wiederum mit unterschiedlichen Standardwerten für verschiedene Größen.
- *gamma*: Der Discount-Faktor γ ist eine Zahl im Bereich von 0 bis 1, die angibt, inwiefern der Wert des nächsten Zustands bei der Berechnung von Value-Loss-Funktion eine wichtige Rolle spielt. Dabei wird die Value-Loss-Funktion als $R + \gamma \cdot V_{t+1} - V_t$ gewählt, die der Bellman-Gleichung in den meisten ADP-Modellen entspricht. In der aktuellen Implementierung wird es standardmäßig auf 0 gesetzt und die Value-Loss-Funktion als $R - V_t$ modifiziert.

Die aktuellen AlphaZero-Modelle werden mit den standardmäßigen Werten der oben genannten Hyperparametern feinabgestimmt, die größtenteils von einem AlphaZero-Projekt [1] übernommen wurden. Die Vermutung liegt nahe, dass die Modelle mit der standardmäßigen Wahl der Hyperparameter plausible Ergebnisse in der Evaluierung und Benchmarking erzielen.

2.5. Benchmarks

In diesem Teil werden die wichtigsten Kenngrößen für die Trainingsergebnisse sowie die Evaluierungsergebnisse vorgestellt. Die Trainingsergebnisse können direkt aus zwischengespeicherten Log-Dateien entnommen werden. Jedoch werden die Evaluierungsergebnisse erst nach dem Training gesammelt.

2.5.1. Fehler

loss:
Gegeben ein Batch der Größe k in einem $M \times N$-Spiel, wird der Value-Fehler durch $MSE(V, \hat{V}) = \frac{1}{k}\sum_{i=1}^{k}(V_i - \hat{V}_i)^2$ gemessen, also die mittlere quadratische Abweichung zwischen den tatsächlichen Bewertungen $(V_i)_{1\leq i\leq k}$ und den Bewertungsvorhersagen $(\hat{V}_i)_{1\leq i\leq k}$.

Der Policy-Fehler eines einzelnen Spielzustandes, der die Diskrepanz zwischen den tatsächlichen und den vorhergesagten Policies quantifiziert, wird durch das negative Skalarprodukt der geschätzten logarithmierten Policy-Wahrscheinlichkeiten $(\log p_j)_{1\leq j\leq M\cdot N}$ und den tatsächlichen MCTS-Wahrscheinlichkeiten $(q_j)_{1\leq j\leq M\cdot N}$ berechnet. Aus dem Durchschnitt der Policy-Fehler der einzelnen Zuständen ergibt sich ein mittlerer Policy-Fehler, der im Training eines Batches verwendet werden kann.

Der zu minimierende Hauptfehler besteht nämlich aus der Summe des Policy- und Value-Fehlers. Ein hoher Wert weist auf die Divergenz des Modells hin, also darauf, dass die Ergebnisse des Modells noch signifikant von den Zielwerten abweichen. Ein erfolgreiches Training ist durch eine kontinuierliche Regression des Fehlers erkennbar. Wenn das jedoch nicht der Fall ist, liegt es möglicherweise an einen Fehler in der Modell-Architektur oder ein problematisches Hyperparameter wie z. B. eine hohe Lernrate.

Ist im Laufe des Trainings ein Sättigungspunkt erreicht, sodass das Modell ab einem gewissen Punkt keine Lernfortschritte mehr macht, kann das Training vorzeitig gestoppt werden.

2.5.2. KL-Divergenz

kl: Die KL-Divergenz [7], formal beschrieben als $\mathrm{KLDiv}(p, q) = \sum_i p_i \log \frac{p_i}{q_i}$, quantifiziert, wie stark sich eine neue Wahrscheinlichkeitsverteilung von einer vorherigen unterscheidet. Sie wird verwendet, um den Effekt eines Batch-Trainings auf die Policy-Wahrscheinlichkeiten zu bewerten, indem die neuen Policy-Wahrscheinlichkeiten $(p_i)_i$ mit den alten $(q_i)_i$ verglichen werden.

Eine hohe KL-Divergenz weist auf relativ große Änderungen hin und darauf, dass die Lernrate um ein festgelegtes Vielfaches (z. B. $lr\_step = 1.5$) reduziert werden sollte. In der aktuellen Pipeline werden die Lernraten in Abhängigkeit von KL-Divergenz angepasst [37].

2.5.3. Entropie

entropy: Die Entropie [5], formal beschrieben als $\mathrm{H}(p) = -\sum_i p_i \log p_i$, misst die Unvorhersehbarkeit der Policy-Wahrscheinlichkeiten.

Eine hohe Entropie deutet auf eine hohe Explorationsrate der Policy hin und impliziert, dass es noch nicht konvergiert ist. Erwünscht ist ein stets sinkender Entropie-Trend im Laufe des Trainings, um die Konvergenz der Policy sicherzustellen.

2.5.4. Erklärte Varianz

expl_var: Die erklärte Varianz, bezeichnet als $\mathrm{EVar}(p, q) = 1 - \frac{V(p-q)}{V(p)}$, zeigt auf, wie hoch der Anteil einer abhängigen Variablen ist, der durch eine unabhängige Variable erklärt wird. Sie gibt an, inwiefern die Reward-Vorhersagen des Modells mit den tatsächlichen Rewards übereinstimmen.

Erstrebenswert ist eine Zunahme der erklärten Varianz im Trainingsverlauf, die darauf hinweist, dass das Modell signifikante Lernfortschritte macht. Im sonstigen Fall mag das an ein problematisches Hyperparameter oder das Modell-Design liegen, was dazu führt, dass das Modell früher gelernte Informationen verliert.

d_expl_var: Diese Metrik beinhaltet die Änderungsrate von Vorhersagengenauigkeit im Laufe eines Batch-Trainings, und kann für eine genauere Analyse der Training-Effektivität benutzt werden. Ist es nahezu stets positiv, kann davon ausgegangen werden, dass das Modell lernfähig ist.

2.5.5. Spielstärke

strength: Die Spielstärke ist eine paarweise Evaluationsmetrik zur Ermittlung der Gewinnchancen für den ersten Spieler gegenüber dem zweiten Spieler. In diesem Zusammenhang wird sie wie folgt berechnet:

$$S(n, k, m, w, t) := \frac{n}{k} \cdot m + w + \frac{t}{2} \tag{2.1}$$

wobei w und t die Anzahl der Gewinne bzw. Unentschieden in m Evaluierungsspielen sind und n die aktuelle Iterationsanzahl des Gegners, ausgedrückt als ein Vielfaches von k (die Iterationsanzahl zu Beginn des Trainings).

Aus der Spielstärke können teilweise Spielinformationen wiederhergestellt werden. Aus $r := s \mod m$ folgt, dass $w + \frac{t}{2} := r$ und $n := \frac{s-r}{m} \cdot k$ ist. Dies hat nämlich zur Folge, dass das Einsetzen von Spielstärke anstelle tatsächlicher Spielergebnisse die Effizienz der Datenspeicherung deutlich erhöht, und zwar ohne großen Informationsverlust.

2.6. Evaluierung

Um die relative Spielstärke von AlphaZero-Varianten genauer auszuwerten, wird eine Funktion für paarweise Vergleiche von Spielern benötigt.

Sei $\text{comp}_\varepsilon(\mathbf{A}_1, \ldots, \mathbf{A}_n)$ eine abstrakte Form der Methode (befindlich in "src/comp.py"), die für alle $i < j$ Zweikämpfe von $\mathbf{A}_i$ und $\mathbf{A}_j$ durchführt. Standardmäßig wird ε, der *epsilon*-Wert jedes Spielers, auf 0 gesetzt.

Um die Leistung von Spielern in verschiedenen Spielszenarien zu entdecken, wird es empfohlen, dass in einem beliebigen Zweikampf mindestens einer der Spieler nichtdeterministisch ist; sonst würden die Spieler nahezu immer den gleichen Zug in einem gegebenen Zustand durchführen. Um dieses Problem zu beheben, wird in der aktuellen Implementierung ein beliebiger Spielzustand als Startzustand eingeführt (bspw. mit $k = 2$ rein zufällig ausgewählten Zügen).

Alternativ kann die Policy-Temperatur *temp* erhöht oder eine geeignete Störungsrate $\varepsilon > 0$ gewählt werden. Hierbei muss zur Kenntnis genommen werden, dass eine höhere *temp* oder ε eine verstärkte Zufälligkeit im Verhalten der betreffenden Spieler mit sich bringt. Dies könnte die Fairness gegenüber deterministischen bzw. nichtdeterministischen Spielern potenziell beeinträchtigen.

3. Ergebnisse

In diesem Kapitel werden bspw. die Trainings- sowie Evaluierungsergebnisse für drei verschiedene Spielkonstellationen 'Small', 'Medium', 'Large' vorgestellt.

3.1. Umgebung und Ablauf des Trainings

Damit die rechenintensiven Trainingsprozesse ununterbrochen und ohne äußere Umgebungsfaktoren laufen können, werden alle Trainings sowie Evaluierungen in Hydra durchgeführt, einem internen Cluster, der mittels SSH-Verbindungen fernbedient werden kann und aktuell von verschiedenen Forschungsgruppen von TU Berlin benutzt wird, wie z. B. ML und MLSEC.

Die Trainings werden ausschließlich in einer Partition namens "cpu-7d" ausgeführt, in der die von den Benutzern übermittelten Jobs eine Woche ohne Unterbrechung laufen können. Die Nutzung von CPU-Ressourcen wird von der dazugehörigen Head-Gruppe "head001-020" geplant, in der den jeweiligen Heads "head001" bis "head020" Zugriff gewährt wird. Diese Head-Gruppe verfügt über zwei CPUs mit jeweils 16 Rechenkernen und ein RAM mit einer Speichergröße von 512 GB.

In der gegebenen Umgebung dauert ein Training mit jeweils $n = 1000$ Epochen im Durchschnitt 6, 18 bzw. 36 Stunden für Small, Medium bzw. Large. Das Large-Modell wird fünfmal trainiert ($N_L = 5000$), wobei das Training von dem Small-Modell ab drittem abgeschlossenen Training früh angehalten worden ist ($N_S = 3000$). Im Gegensatz dazu wird das Medium-Modell um eine Trainingseinheit länger trainiert ($N_M = 6000$).

3.2. Trainingsergebnisse

Die Analyse von Trainingsergebnissen, die während des Trainings jeweils in "<Spielgröße>/train.log" zwischengespeichert worden sind, geben Aufschluss über die Leistung des Trainings der zugrundeliegenden Modelle **Zero_Net** verschiedener Spielgröße. Im Folgenden werden diese Ergebnisse visualisiert und vergleichsweise erklärt.

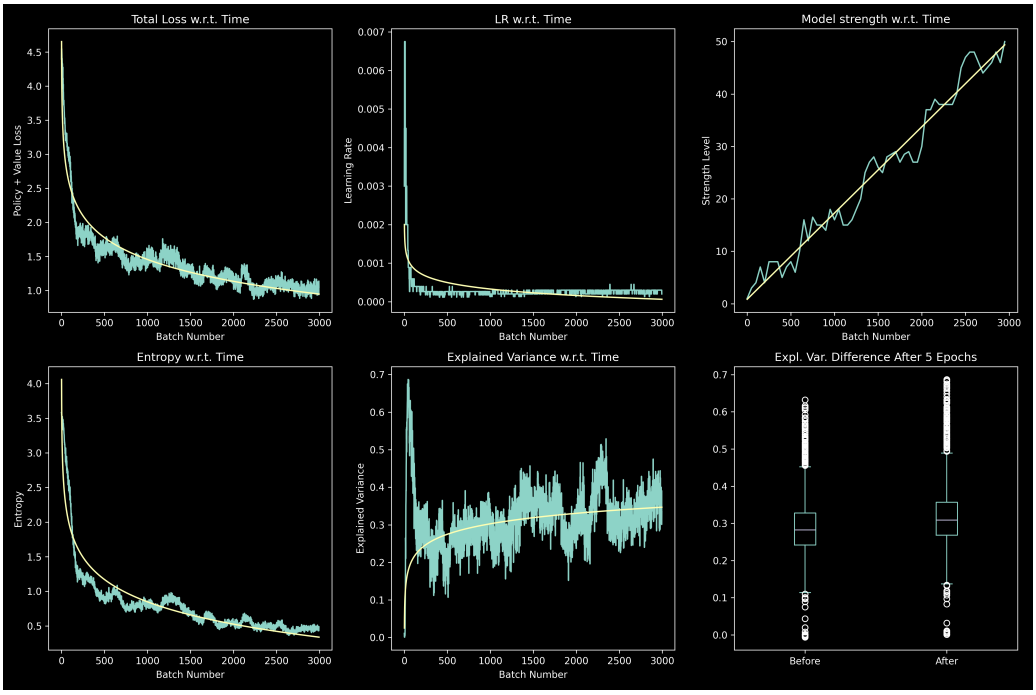


Abbildung 3.1.: Trainingsergebnisse für Small

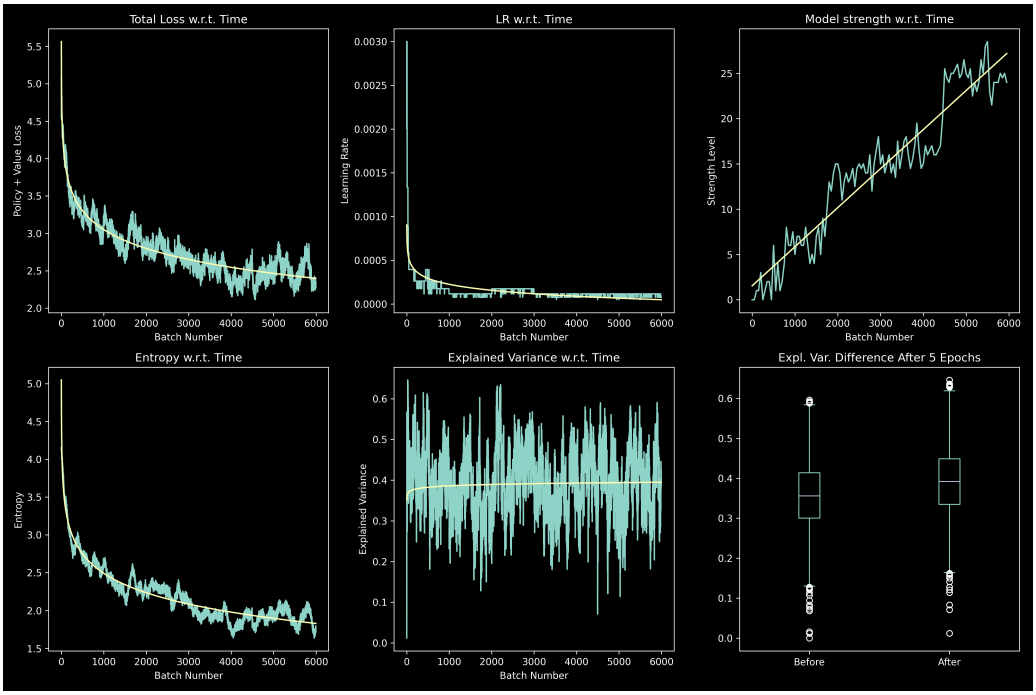


Abbildung 3.2.: Trainingsergebnisse für Medium

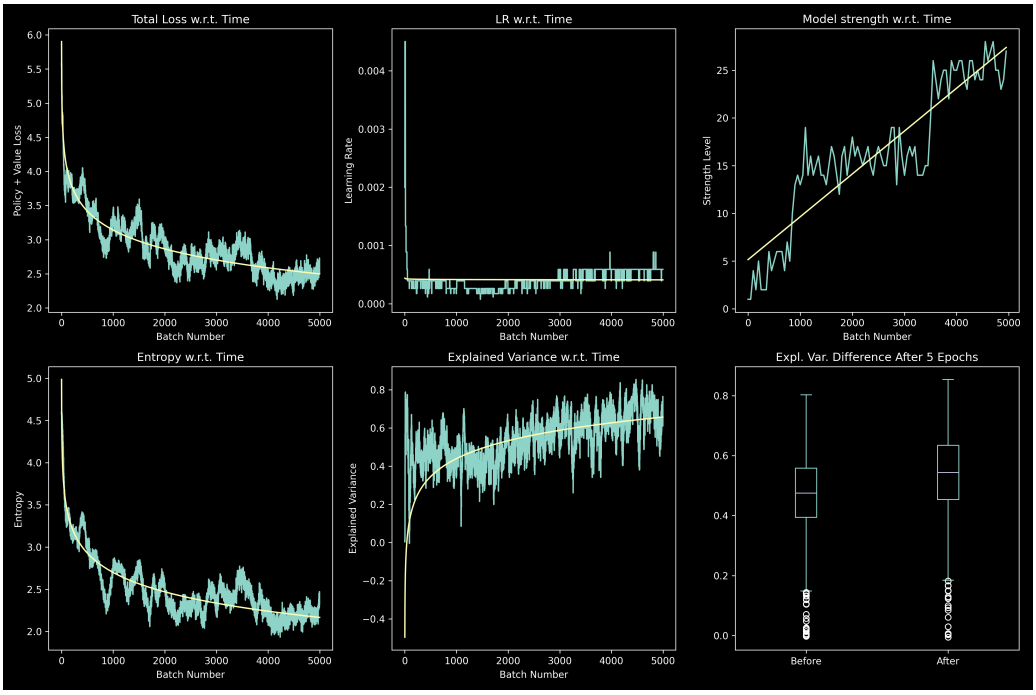


Abbildung 3.3.: Trainingsergebnisse für Large

Jede Abbildung besteht aus folgenden Teil-Diagrammen:

- Links, oben: Trainingsfehler im zeitlichen Verlauf
- Mitte, oben: Lernrate im zeitlichen Verlauf
- Links, unten: Entropie im zeitlichen Verlauf
- Mitte, unten: Erklärte Varianz im zeitlichen Verlauf
- Rechts, unten: Änderungsrate von Erklärter Varianz
- Rechts, oben: Geschätzte Spielstärke im Laufe des Trainings

3.2.1. Trainingsfehler und Lernrate

In der Abbildung 3.3 werden die Trainingsfehler des Modells (für Large) im Laufe der 5000 Epochen dargestellt. Die Kurve lässt sich durch eine streng monoton fallende logarithmische Funktion beschreiben. Analog sinkt der Trainingsfehler in 3.2 und 3.3 kontinuierlich.

Ähnlicherweise nimmt die Lernrate des Modells im Laufe der Zeit stets ab, wie den Abbildungen 3.1, 3.2 und 3.3 zu entnehmen ist. Dies ist ein starker Hinweis, dass die Änderungsrate der gelernten Modellparameter im Laufe der Zeit stets reduziert wird, sowie dass, das Modell konvergiert.

3.2.2. Entropie

In den Abbildungen 3.1, 3.2 und 3.3 ist es zu sehen, dass die Entropie von Policy-Wahrscheinlichkeiten im zeitlichen Verlauf nahezu perfekt durch eine stets sinkende logarithmische Funktion approximiert wird. Dieses Erkenntnis weist auf die Konvergenz von der Policy-Wahrscheinlichkeitsverteilung hin.

Eine andere Beobachtung ist es, dass die untere Schranke von Entropie je nach Größe des Spielfelds unterschiedlich ist. Die Entropie sinkt in Small bis auf 1, wobei im Fall von Large ein Tiefpunkt von 2 erreicht wird. Dementsprechend können diese Werte normalisiert werden, um die Entropie-Werte verschiedener Spielgrößen untereinander vergleichbar zu machen.

3.2.3. Erklärte Varianz

Die erklärte Varianz ist im Laufe des Trainings in allen Abbildungen tendenziell logarithmisch gestiegen, woraus folgt, dass der Fehler von Reward kontinuierlich minimiert wird. Merkwürdig ist jedoch die besonders hohe erklärte Varianz am Anfang des Trainings, welche durch eine ebenso hohe Lernrate und dadurch begründet werden, dass das Modell am Anfang nur wenig Spielzustände zu lernen hat. In Small wird dieses Phänomen im Vergleich zu Medium und Large besonders stark ausgeprägt.

In denselben Bildern ist es auch zu erkennen, dass die erklärte Varianz zufolge eines Batch-Trainings im hohen Ausmaß erhöht wird (bspw. im Falle von Small um $avg\_d\_expl\_var = 0.05$). Unter der Annahme, dass das Modell die erlernten Rewards nicht vergisst, sollten dann die Reward-Vorhersagen des Modells im Laufe des Trainings effektiv gegen die wahren Rewards konvergieren.

3.2.4. Spielstärke

Als Letztes werden die Spielstärken der Modelle im Laufe des Trainings gezeigt. In allen Graphen ist ein linearer Wachstumstrend im zeitlichen Verlauf zu erkennen. Hierbei könnte die Pearson-Korrelationsanalyse mit der folgenden Formel einwandfrei angewendet werden, wobei X dem Trainingskontrollpunkt bzw. Y der Spielstärke entspricht:

$$\rho(X,Y) = \frac{\sum_{i=1}^{n}(X_i - \bar{X})(Y_i - \bar{Y})}{\sqrt{\sum_{i=1}^{n}(X_i - \bar{X})^2 \sum_{i=1}^{n}(Y_i - \bar{Y})^2}}$$

In allen drei Fällen ist eine hohe Pearson-Korrelation ρ im Bereich $[0.85, 0.95]$ zu erwarten. Im besonderen Fall von Small ist es sogar dem Modell in nur 3000 Epochen von Training gelungen, den stärksten Gegner mit $n\_uct = 5000$ in 10 von 10 Spielen zu besiegen und somit die Maximalspielstärke zu erreichen.

3.3. Evaluierungsergebnisse

Sei $\mathbf{Zero}_t$ bzw. $\mathbf{Flat}_t$ der Zero- bzw. Flat-Spieler mit dem besten Modell nach den ersten t Trainings. Ferner sei $\mathbf{ZeroX}_t$ der Zero-Spieler mit MCTS-Wiederverwendung (siehe Unterabschnitt 2.3.4), wo-

bei der Suffix -*X* für *eXtra* steht. Sei $\mathbf{UCT}^k$ der UCT-Spieler mit $1000 \cdot k$ MCTS-Iterationen, also mit einem Schwierigkeitsgrad von k.

Um die Leistung von allen erwähnten Zero-Varianten zu testen, werden die Zweikampfsdaten in Form von comp(...) benötigt. Unten sind die sogenannten Zweikampfmatrizen gegeben, die die Spielstärke (siehe Unterabschnitt 2.5.5) von dem ersten Spieler gegen den zweiten Spieler jeweils in $n = 50$ Spielen in kompakter Form darstellen. Dabei sind einige Zweikämpfe ausgeschlossen (und dementsprechend in den Zweikampfmatrizen maskiert), nämlich die Kämpfe eines Spielers gegen sich selbst, die Kämpfe zweier UCT-Spieler und die Kämpfe von zwei verschiedenen Zero-Varianten, die sich im Trainingskontrollpunkt unterscheiden.

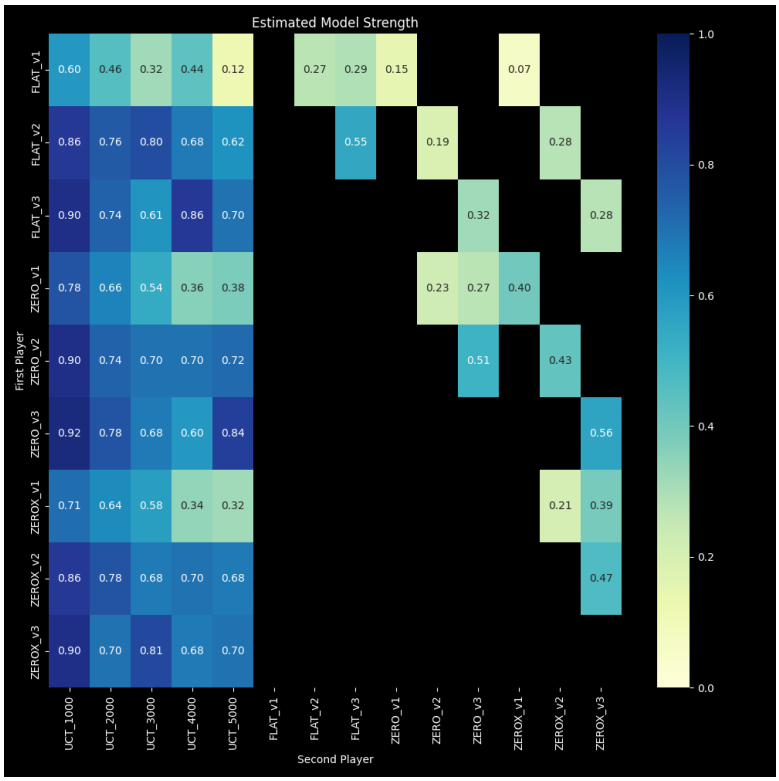


Abbildung 3.4.: Zweikampfsresultate in Small

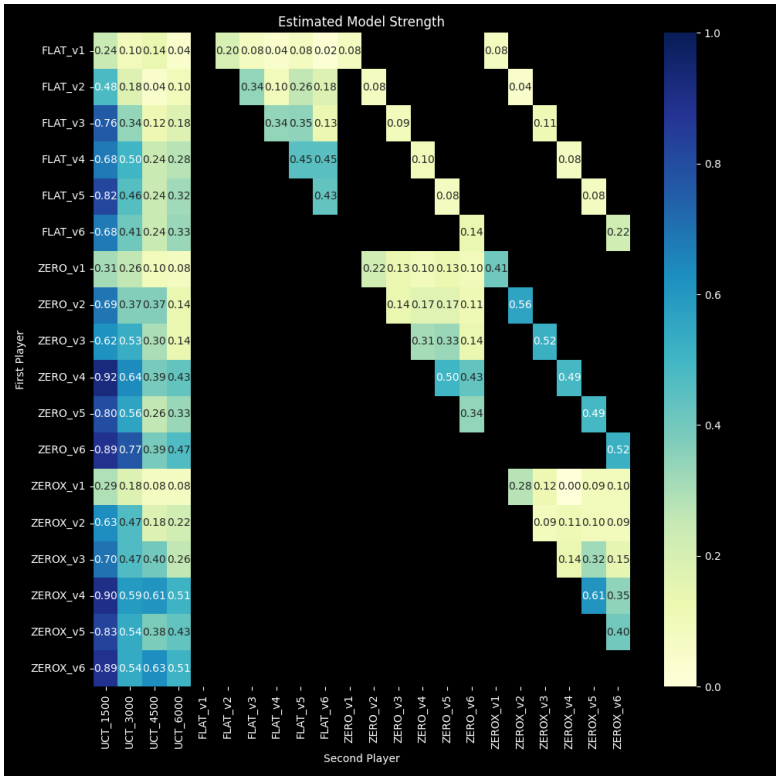


Abbildung 3.5.: Zweikampfsresultate in Medium

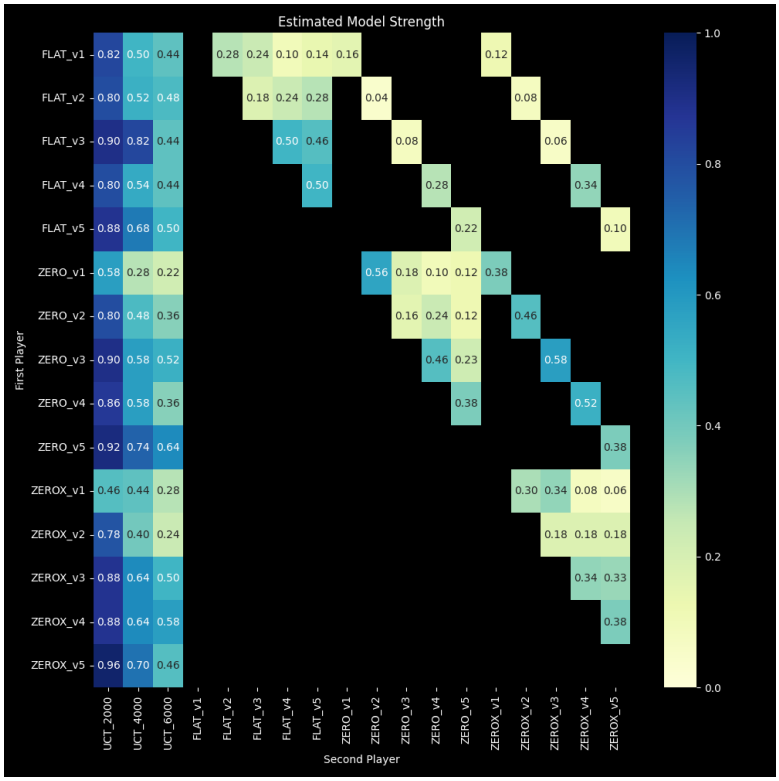
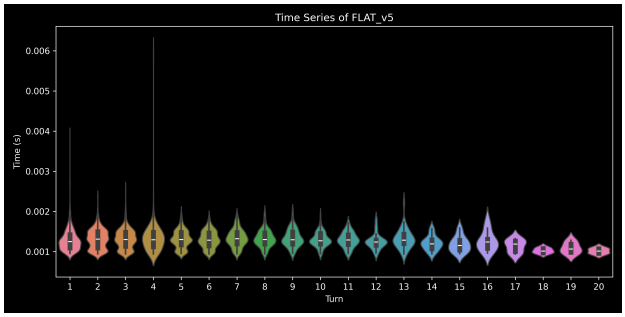


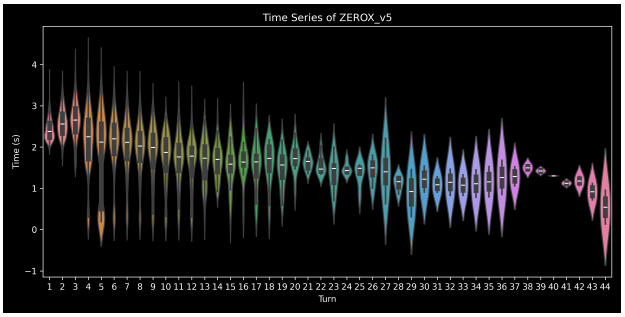
Abbildung 3.6.: Zweikampfsresultate in Large

Für eine feinere Untersuchung der Spielstärken kann zudem direkt auf die Ergebnisse einzelner Zweikämpfe in der ”competition/”-Ordner der jeweiligen Spielgröße zugegriffen werden. Jede CSV-Tabelle mit dem Namen ”competition/..._lengths.csv” repräsentiert die Länge der Endspiele aller gespielten Partien zweier Spieler, wobei die Spalten aus Kombinationen des ersten Spielers und des Endergebnisses des Spiels bestehen. Jede CSV-Datei mit dem Namen ”competition/..._counts.csv” stellt die Anzahl dieser Ereignisse fest. Die weiterführenden Analysen sind im Kapitel 4 zu finden.

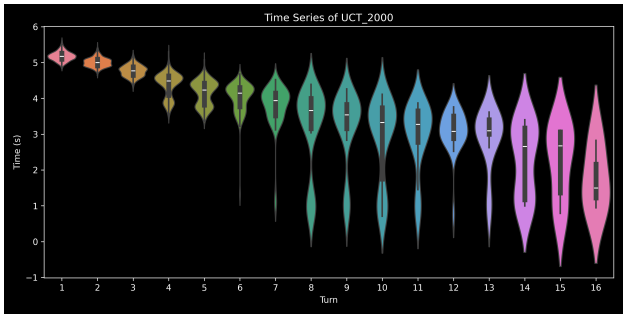
Analog befindet sich unter den nach Spielgrößen gruppierten Ordnern ”timeseries/”-Ordner, die die Zeitdauer aller Züge eines Spielers - ermittelt von gespielten Partien - in der jeweiligen CSV-Datei enthält, wobei die Spalten die Spielrunden repräsentieren. Beispielsweise wird ein Teil der Ergebnisse in Large als eine Reihe von Violindiagrammen visualisiert:



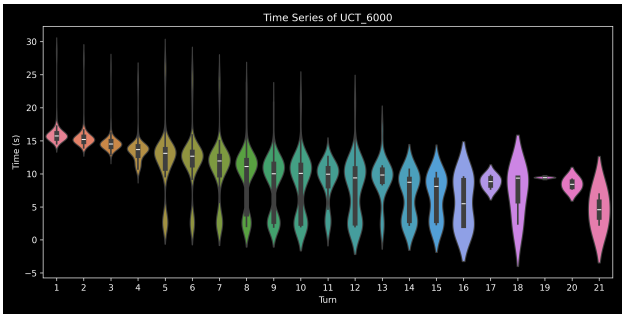
(a) Zugdauer von Flat



(b) Zugdauer von ZeroX



(c) Zugdauer von UCT_2000



(d) Zugdauer von UCT_6000

Die X- bzw. Y-Achse beschriftet die Spielrunden bzw. die Zeitdauer in Sekunden, wobei jeder Spielrunde ein Violindiagramm zugeordnet ist, die zur Schätzung der Verteilung von der Zugdauer dient.

Es ist offensichtlich, dass ausschließlich der Algorithmus **Flat** eine konstante Ausführungszeit aufweist, was in Abschnitt 2.3.5 erläutert wird, mit einer geschätzten Obergrenze von 2 Millisekunden. Bei den anderen Algorithmen zeigt sich jedoch eine durchgehend lineare Abnahme in Abhängigkeit von der Größe des Suchraums. Die Algorithmen **Zero** und **UCT**_2000 begrenzen die maximale Zugdauer auf 3 Sekunden, während bei **UCT**_6000 die Zugdauer auf bis zu 15 Sekunden ansteigen kann.

4. Diskussion

In diesem Kapitel werden die Ergebnisse von vorherigen Kapiteln ausführlich interpretiert und anhand systematische Vergleiche mit bestehendem Literatur diskutiert.

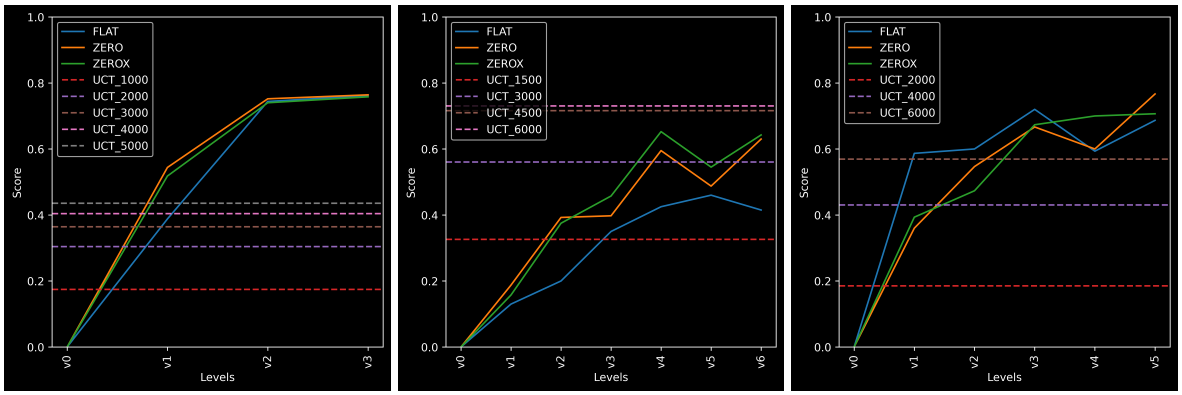
4.1. Fragen zur Leistung von AlphaZero

In diesem Abschnitt werden - anhand Evaluierungsdaten und Vergleich mit bestehendem Literatur - die wichtigsten Hypothesen und Fragen mit Fokus auf Leistung der Zero-Varianten beantwortet.

4.1.1. [Q1] Sind Zero-Varianten stärker als UCT?

Gebraucht werden die Zweikämpfe von allen Zero-Varianten gegen UCT-Spieler: comp($\mathbf{Z}_t, \mathbf{UCT}^k$) für alle $\mathbf{Z}_t \in \{\mathbf{Flat}_t, \mathbf{Zero}_t, \mathbf{ZeroX}_t\}$, alle Schwierigkeitsgrade k von UCT-Spieler und alle Trainingskontrollpunkte t.

Die relevanten Ergebnisse sind in Form von oberen Dreiecksmatrizen in den jeweiligen Zweikampfmatrizen zu finden. Da in allen drei Fällen die Anzahl der paarweisen Vergleiche von Modellen mit steigender Anzahl von Spielern rasant steigt, wird zur Vereinfachung eine allgemeingültige durchschnittliche Spielstärke von allen Spielern anhand der Spieldaten berechnet, die zur Bildung der folgenden totalen Rangordnungen dient.



(a) Q1-Ergebnisse von Small (b) Q1-Ergebnisse von Medium (c) Q1-Ergebnisse von Large

Abbildung 4.1.: Q1-Ergebnisse

In den Kurvendiagrammen 4.1a, 4.1b bzw. 4.1c werden die geschätzten Spielstärken von Zero-Spielern im zeitlichen Verlauf im Vergleich zu UCT-Spielstärken gezeigt.

Aus dem ersten Blick ist es in allen drei Diagrammen zu erkennen, dass sich alle Varianten mit der Zeit stets verbessern oder gegebenenfalls ihre Spitzenleistung aufrechterhalten. In Fällen von Small und Large ist es sogar den finalen Modellen gelungen, eine höhere Spielstärke als alle UCT-Varianten zu erzielen.

In Medium liegen jedoch die ermittelten Niveaus von Zero-Spielern noch weit unterhalb von UCT-Grundlinien. Die Knickpunkte (wie z. B. in v4 im Diagramm 4.1c) zeigen außerdem, dass der Lernprozess nicht stets perfekt linear abläuft.

Der beobachtete Wachstumstrend von Spielstärke von Zero-Spielern stimmt mit den Ergebnissen einer anderen Forschung [26] überein, die den AlphaZero-Algorithmus in Gomoku anwendet.

4.1.2. [Q2] Welche Zero-Variante ist zum Zeitpunkt t am stärksten?

Angefragt werden die Evaluierungsergebnisse von: comp($\mathbf{Flat}_t, \mathbf{Zero}_t, \mathbf{ZeroX}_t$). Aufgrund der niedrigen Policy-Temperatur von allen Zero-Varianten handelt es sich hierbei annäherungsweise um einen deterministischen Fall. Um dies zu verhindern, werden zufällige Startzustände ausgewählt wie im Abschnitt 2.6 erwähnt.

Diese Evaluierungsergebnisse sind bereits in Form von Hauptdiagonalen in den jeweiligen Zweikampfsmatrizen 3.4, 3.5 bzw. 3.6 befindlich. Für eine feinere Untersuchung können die normalisierten Gewinnraten in den folgenden Tortendiagrammen visualisiert werden.

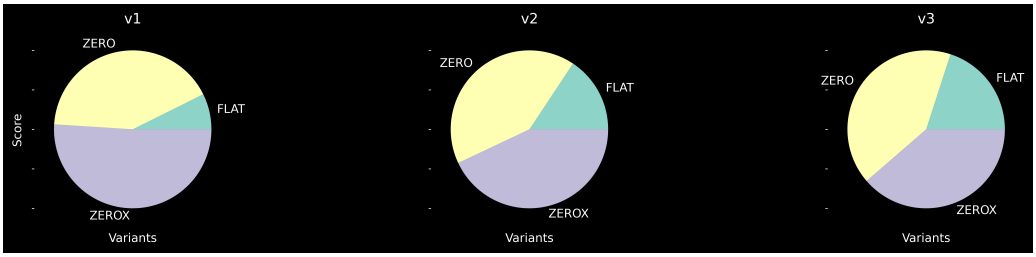


Abbildung 4.2.: Q2-Ergebnisse von Small

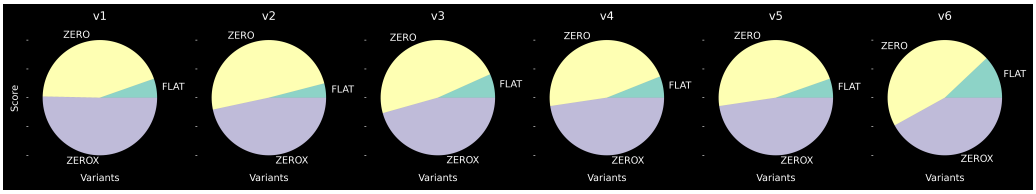


Abbildung 4.3.: Q2-Ergebnisse von Medium

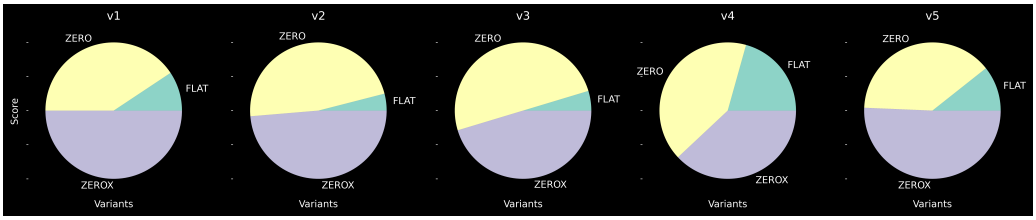


Abbildung 4.4.: Q2-Ergebnisse von Large

In den Tortendiagrammen von Medium und Large ist es ersichtlich, dass in den meisten Fällen **Flat** von **Zero** bzw. **ZeroX** besiegt worden ist. In einigen Ausnahmsfällen (wie z. B. v4 im Diagramm 4.4) hat **Flat** jedoch relativ hohe Gewinnraten erzielt, welches möglicherweise darauf hinweist, dass die erlernte Spielstrategie von **Flat** zu diesem Trainingszeitpunkt mit der von **Zero** und **ZeroX** übereinstimmt.

Eine andere Beobachtung folgt aus den Diagrammen, dass **Zero** ungefähr eine so hohe Leistung wie **ZeroX** erzielt. Vermutlich enstehen die geringfügigen Unterschiede in Gewinnraten von beiden Spielern durch die Randomisierung der Startzuständen.

Im Allgemeinen werden die Spielstärken von allen Zero-Varianten in Diagrammen 4.2, 4.3 und 4.4 mit steigender Anzahl von Trainingsepochen umso ausgeglichener. Dies kann durchaus bedeuten, dass eine hohe Anzahl der MCTS-Iterationen immer weniger Einfluss auf die Leistung des Modells hat, je ausreichender das Modell trainiert ist. Mit zunehmender Iterationszahl sind extreme Vorhersagen also immer unwahrscheinlicher, da eine höhere Zahl an Iterationen die Stabilität der Ergebnisse fördert.

4.1.3. [Q3] Verbessern sich Zero-Varianten im Laufe der Zeit?

Sei $Z \in \{\textbf{Flat}, \textbf{Zero}, \textbf{ZeroX}\}$ eine beliebig aber feste Zero-Variante. Dann werden die Zweikämpfe aller zwischengespeicherten Spieler benötigt: $\text{comp}(\mathbf{Z}_1, \ldots, \mathbf{Z}_T)$. Die relevanten Zweikampfsresultate befinden sich jeweils in dem linken Rechteck in den Zweikampfmatrizen 3.4, 3.5 und 3.6.

Um die paarweisen Vergleiche wegzuabstrahieren, wird analog zu Q1 eine totale Rangordnung aus den ermittelten Durchschnittsspielstärken gebildet und in den folgenden Säulendiagramme visualisiert.

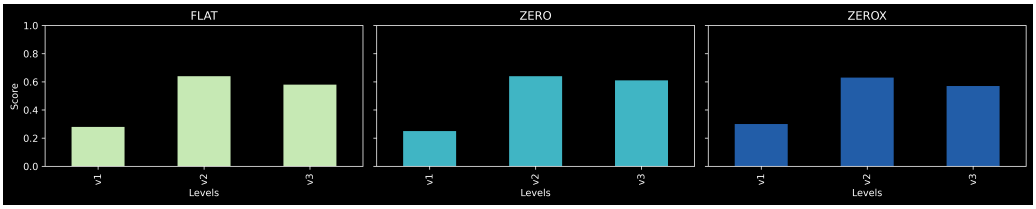


Abbildung 4.5.: Q3-Ergebnisse von Small

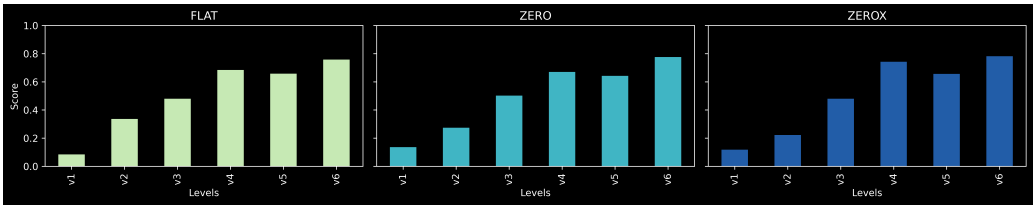


Abbildung 4.6.: Q3-Ergebnisse von Medium

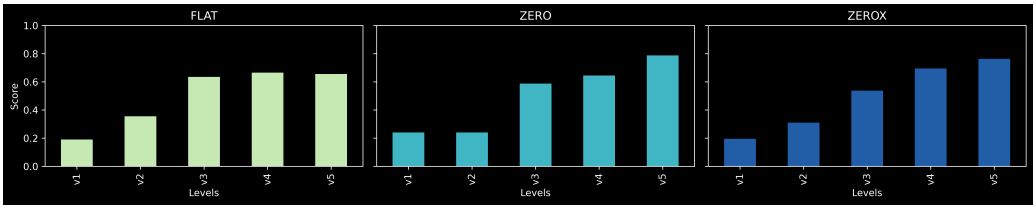


Abbildung 4.7.: Q3-Ergebnisse von Large

In den Diagrammen 4.6 und 4.7 ist es leicht zu erkennen, dass ein beliebiges nachtrainiertes Modell nahezu immer mindestens so stark wie seine Vorfahren ist.

Jedoch zeigt das Diagramm 4.5 überraschenderweise, dass das Weitertrainieren eines Modells um $n = 1000$ Batches ab $N = 2000$ die Spielstärke um eine Kleinigkeit verschlechtert. Dies kann bedeuten, dass die Modelle bis zum Zeitpunkt $N = 2000$ bereits eine Strategie entdeckt haben, mit der sie in den meisten zufällig ausgewählten Startzuständen mit $k = 2$ Zügen problemlos gewinnen können.

4.2. Weitere Diskussionen

Im Zusammenhang mit dem vorherigen Abschnitt 4.1 kommen einige Auffälligkeiten infrage, die im Folgenden ausführlich erklärt werden.

4.2.1. Entdeckung von Gewinnstrategien mithilfe AlphaZero

Dieser Teil beschäftigt sich hauptsächlich mit der Existenz von Gewinnstrategien in Small, und baut auf die Erkenntnisse von den bisherigen Abschnitten 4.1.1, 4.1.2 und 4.1.3 auf.

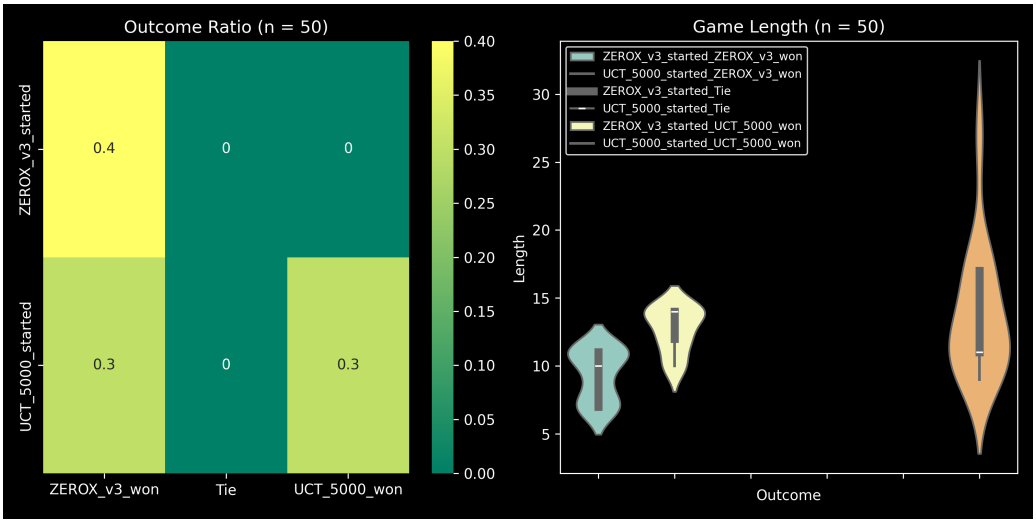


Abbildung 4.8.: ZEROX (3k Epochen) vs. UCT (5k Iterationen)

Dieses Bild besteht aus zwei Teildiagrammen. Das linke Diagramm ist eine Kontingenztabelle der Gewinnanzahl beider Spieler bezüglich der Wahl des ersten Spielers. Das rechte Diagramm visualisiert die Verteilung der Anzahl von Zügen in jeweils verschiedenen Fällen. Die Beobachtung, dass **ZeroX** keinen der $n = 50$ Zweikämpfe verliert, wenn der als erster Spieler anfängt, verstärkt die Hypothese, dass **ZeroX** eine Gewinnstrategie für den ersten Spieler gefunden hat.

Es stellt sich heraus, dass die bestbewertete Zugfolge, aufgedeckt von dem ausreichend trainierten Zero-Modell, lautet: ”c3-d4-d2-b4-c4-c5-c2-c1-e2-f2-b2”. Die folgende Abbildung trägt zu einer feineren Untersuchung dieser Zugfolge bei:

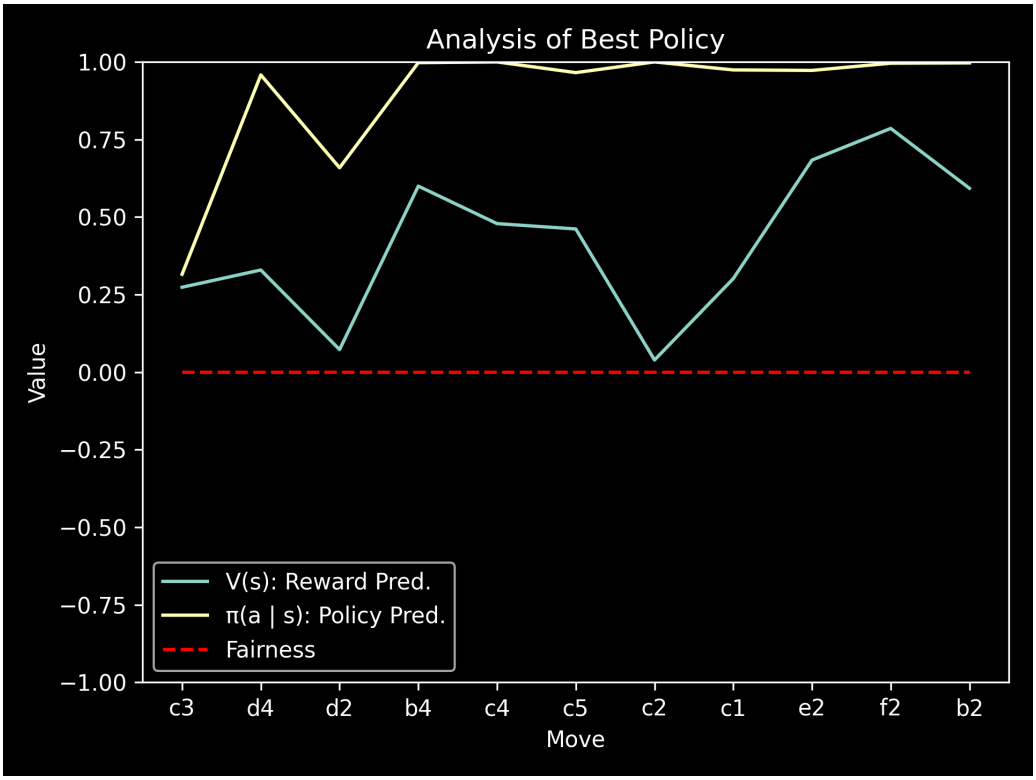


Abbildung 4.9.: Untersuchung von der bestbewerteten Zugfolge

In dieser Figur werden alle bestbewerteten Züge von dem ersten und zweiten Spieler auf der X-Achse aufgelistet. Die blaue Kurve stellt die Reward-Vorhersagen von dem zugrundeliegenden Modell dar, und die gelbe Kurve nämlich die vorhergesagte Policy-Wahrscheinlichkeit des bestbewerteten Zugs. Die beiden Kurven liegen offensichtlich weit über der rot gefärbten Fairness-Linie, woraus folgt, dass das Modell die Gewinnchancen des ersten Spielers überschätzt.

Vergleichsweise wurde in einer anderen AlphaZero-Forschung eine ähnliche Zugfolge der bestbewerteten Züge in Small-Spielen aufgedeckt, die unmittelbar zum Sieg des ersten Spielers führt: ”c3-d4-d2-b4-c4-c2-c5-c6-d3-b3-b5-b2-e2” (vgl. [26], *Appendix I*). Die beiden Strategien stimmen grundsätzlich nur bis zum fünften Zug überein. Ab sechstem Zug gefällt das Modell der anderen Forschung eine Zugentscheidung, die erwiesenermaßen das Spiel um zwei Züge verlängert, und die Gewinnchancen von dem zweiten Spieler geringfügig verbessert. Der Grund dafür mag sein, dass die Bewertungen eines Endzustandes unabhängig von der Länge des Spiels berechnet werden. Als eine Verbesserungsidee könnte die Bewertungsfunktion der *Gomoku*-Implementierung so modifiziert werden, dass die Länge des Spiels im Hinblick auf den Gewinner – in diesem Fall den ersten Spieler – bestraft wird. Vermutlich würde der zweite Spieler dann die Züge bevorzugen, die für ein längeres Endspiel sorgt.

Um sicherzustellen, dass eine Strategie in Gomoku auf theoretischer Sicht eine Gewinnstrategie ist, müssen alle möglichen rationalen Gegenzüge des zweiten Spielers untersucht werden. Ausgeschlossen sind bspw. die Züge außerhalb der Reichweite einer aktuellen Threat oder die irrationalen Züge im Falle eines Zugzwangs, die die aktuelle Drohung nicht verhindern.

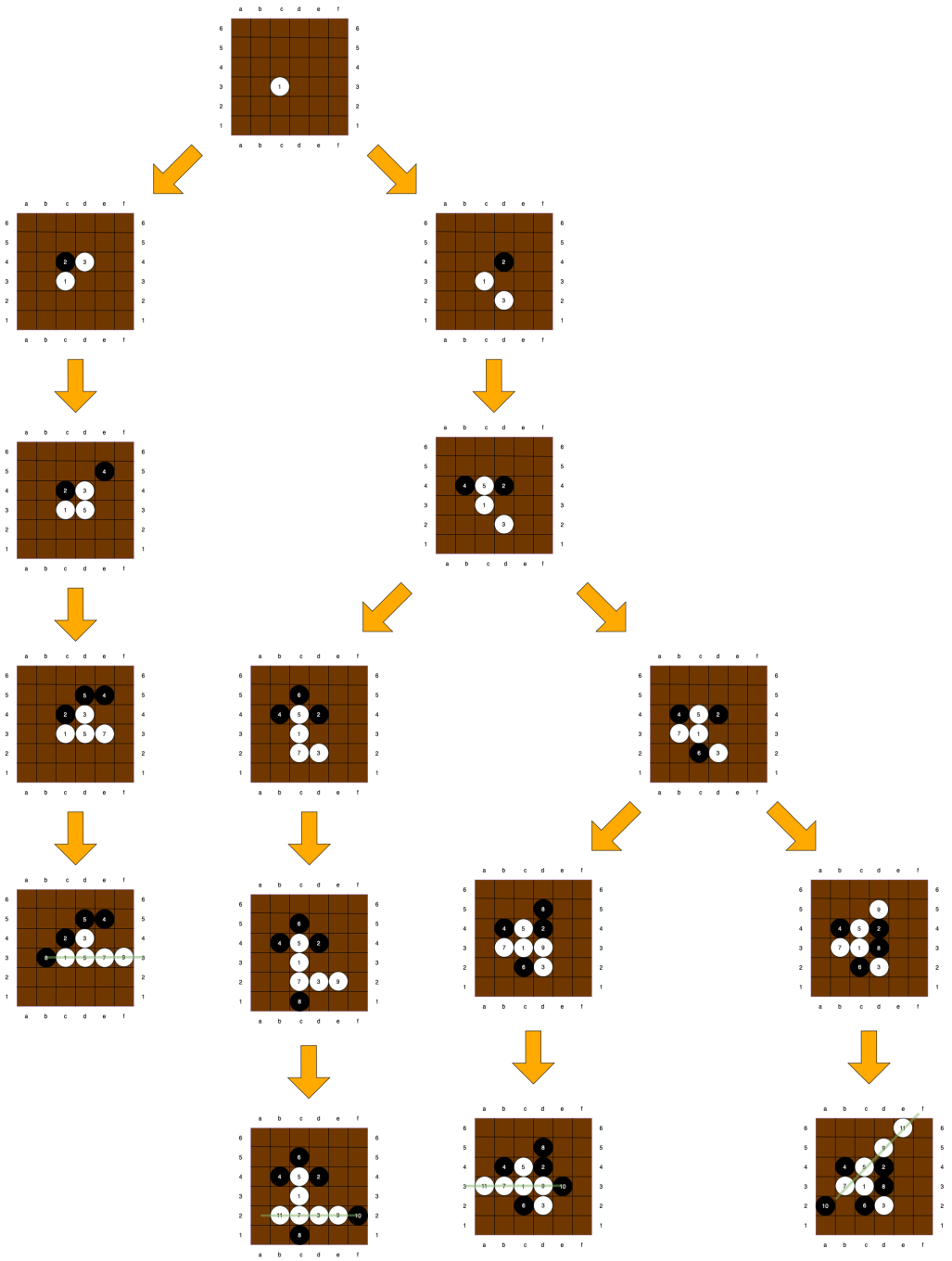


Abbildung 4.10.: (Vereinfachte) Gewinnstrategie vom ersten Spieler in Small

Die Abbildung 4.10 visualisiert die Gewinnstrategie des ersten Spielers ausführlich, und zeigt, welche Zugfolgen hundertprozentig zum Sieg des ersten Spielers führen. Dabei wird auch die bestbewertete Zugfolge vor Augen geführt, die bereits im Kurvendiagramm 4.9 auf der X-Achse zu finden ist.

Interessanterweise teilen alle gegebenen Gewinnstrategien des ersten Spielers einige Gemeinsamkeiten, insbesondere die Existenz einer VCT-Muster (Victory of Continuous Three). Im Falle einer ununterbrochenen Reihe dreier Steinen mit offenen Enden kann der Gegner höchstens ein Ende blockieren.

Somit bleibt immer noch ein Ende offen, und der aktuelle Spieler gewinnt unmittelbar, wenn er einen Zug auf dieses Spielfeld macht.

4.2.2. Verbesserung von Evaluierungsmethoden

Wie in den bisherigen Ergebnissen 4.1b und 4.6 ersichtlich ist, zeigen einige Ergebnisse auf temporäre Verschlechterungen in Modellleistung auf. Dies mag jedoch selber an den Sonderfällen der Spielstärke als Evaluierungsmetrik liegen, insbesondere direkt nach der Erhöhung des Niveaus.

Beispielsweise wird ein Modell A, das einen UCT-Spieler nur in einem von 10 Spielen besiegt, stärker angesehen als ein anderes Modell B, das in 10 von 10 Spielen gegen einen schwächeren UCT-Gegner gewonnen hat. Die totale Korrektheit der Spielstärke setzt voraus, dass das Modell B in der Lage ist, alle Spiele gegen alle schwächeren UCT-Gegner zu gewinnen, sowie dass das Modell A gegen alle stärkeren UCT-Gegner verliert. Dies ist aber eine strikte, schwer realisierbare Annahme und könnte eventuell zum Verlust der wahrhaft besten Modelle führen.

Wenn diese Verdachtsfälle relativ häufig auftreten würden, könnte an dieser Stelle eine ähnliche Rangordnungsmethode wie im Unterabschnitt 4.1.1 vorgeschlagen werden, die die ungewichtete Summe der Gewinnraten des aktuellen Modells gegen UCT-Spieler aller Schwierigkeitsgrade berechnet. Folglich stellt sich die offene Frage für die zukünftigen Forschungen, inwieweit dieser Ansatz die Leistung in gegebenen Sonderfällen verbessern bzw. verschlechtern könnte.

4.2.3. Verbesserung von Trainingsmethoden

Im Laufe dieses Teils werden die aktuellen Trainingsmethoden infrage gestellt, da sie möglicherweise sowohl zeitliche als auch leistungsmäßige Herausforderungen darstellen.

Der im Teil 3.1 gezeigte Wachstumstrend der Trainingslaufzeit in Abhängigkeit von der Spielgröße ist auf das quadratische Wachstum vom Branching-Faktor zurückzuführen. Der enorm hohe Zeitaufwand bei hohen Spielgrößen wie z. B. $19x19$ könnte den Trainingsprozess verlangsamen und dadurch die Leistungsfortschritte pro Zeiteinheit erheblich reduzieren. Um einen kleinen zeitlichen Vorsprung pro MCTS-Iteration zu schaffen, könnte eventuell ein schnellerer MCTS-Ansatz ausgesucht werden. Im Rückblick auf die bisherigen Ergebnisse ist es zu beachten, dass eine zunehmende Anzahl der MCTS-Iterationen im Laufe des Trainings immer weniger Leistungsverbesserung bewirkt.

In diesem Projekt werden verschiedene Hyperparameter und Designentscheidungen direkt von anderen Projekten übernommen oder gegebenenfalls durch Versuch-und-Irrtum und per Hand optimiert. Stattdessen könnten sie einem gewählten Algorithmus zur Hyperparametersuche [9] delegiert werden. Wünschenswert sind eine gut geeignete Zielfunktion zur Leistungsermittlung, eine gute Wahl des Wertenbereiches sowie Zugriff auf einem fernbedienten Cluster, die parallele Prozesse einwandfrei betreiben kann, damit die Gesamtzeit des Trainings nicht mit der Anzahl der Kombinationen hochskaliert.

Ferner könnte die Modellarchitektur des zugrundeliegenden CNNs optimiert werden, um noch höhere Informationskapazität zu erschaffen. Unten sind einige wichtige Aspekte aufgelistet:

- *Wahl der Aktivierungsfunktion*: In der aktuellen Konfiguration wird ReLU als die standardmäßige Aktivierungsfunktion festgelegt, die vermutlich zu einem Problem des verschwindenden Gradienten führen mag. An dieser Stelle könnte bspw. eine andere nichtlineare Funktion benutzt werden, die dieses Problem behandelt [20].

- *Wahl der individuellen Schichten*: Um die Leistung der Konvolutionellen Schichten zu verbessern, können einige empfehlenswerte Vorgehensweisen wie z. B. Batchnormalisierung [11] oder Max-Pooling ([15], Abschnitt 2.2.) vorgeschlagen werden.

- *Wahl des Lernratenanpassungsansatz*: Der aktuelle Algorithmus zur Lernratenanpassung benutzt ausschließlich die KL-Divergenz von den Policy-Wahrscheinlichkeiten vor und nach einer Trainingsepoche, ohne die Reward-Änderungsrate mitzuberücksichtigen. Dies kann ein Grund dafür sein, dass die erwartete erklärte Varianz bei jeder Spielgröße stark unterschiedlich ist: $[0.3, 0.5]$ bei Small (Abb. 3.1), $[0.2, 0.6]$ bei Medium (Abb. 3.2) und $[0.6, 0.8]$ bei Large (Abb. 3.3). Es wird vermutet, dass ein Lernratenanpassungsansatz mit Bezug auf Policy und Rewards dazu beitragen könnte, die Reward-Metriken über verschiedene Trainingsexperimente zu stabilisieren.

4.2.4. Limitationen von AlphaZero und deren Lösungsvorschläge

In den meisten ML-Anwendungen ist es praktikabel und günstig, ein ausreichend prätrainiertes Basismodell für spezifische Aufgaben feinabzustimmen, statt vom Anfang an zu erlernen. In diesem Projekt wird jedoch ein Gomoku-Modell ausschließlich für eine gegebene Spielgröße trainiert, da die Anzahl der Filter in der ersten Konvolutionen-Schicht direkt von der Spielgröße abhängt.

Aus dem erwähnten Grund ist der Informationsaustausch zweier Modelle verschiedener Spielgröße grundsätzlich verboten; es sei denn, dass ein prätrainiertes Modell mit nötigen Anpassungen in ein anderes Modell anderer Spielgröße integriert wird, wie z. B. die Übereinstimmung von Ein- und Ausgabedimensionen für die gegebene Spielgröße.

Um die Ein- und Ausgabeschichten variabler Größe zu definieren, ohne große Änderungen in der Modellarchitektur vornehmen zu müssen, könnte eine alternative Idee vorgeschlagen werden, komplexere NN-Ansätze anstelle CNNs zu verwenden, wie z. B. LSTM ([38], *Abschnitt 2.*) oder GNN ([45], *1. Introduction*).

Ein anderes Erkenntnis ist es, dass die Spielregeln über verschiedene Spielvarianten (wie z. B. Freestyle Gomoku, Gobang, Go) nur geringfügig voneinander unterschiedlich sind, welches bedeutet, dass ein bereits erworbenes Fachwissen in einer Spielvariante womöglich nützlich in anderen Spielvarianten sein könnte. In einem solchen Fall wird spielunabhängiges Fachwissen erwünscht, was aber im Falle von AlphaZero nicht möglich ist, weil die Spielregeln in den Methoden von *State* bereits vorgegeben sind. MuZero ist ein Algorithmus, der in der Lage ist, in verschiedenen kombinatorischen Spielen wie Schach, Shogi und Go eine Leistung zu erbringen, die über dem menschlichen Niveau liegt, ähnlich wie AlphaZero, und das ohne die Notwendigkeit, explizit die Spielregeln zu kennen (vgl.

[36]). Somit könnten vermutlich spielübergreifende Strategien zur Gewinnmaximierung entdeckt und in verschiedenen Spielen weiterverwendet werden.

5. Fazit

Diese Arbeit beschäftigt sich hauptsächlich mit der Frage, welche Vor- und Nachteile der AlphaZero-Algorithmus bei der Auswertung und Strategiefindung in Gomoku mit sich bringt. Ein AlphaZero-Modell ist dank Fortschritten in Bildverarbeitung und Reinforcement-Learning in der Lage, einen MCTS-Algorithmus mit deutlich mehr Zahl an Baumsucheniterationen zu besiegen, und zwar ausschließlich durch Trainings anhand Selbstspieldaten und ohne fachliches Vorwissen. Die Effektivität von AlphaZero-Algorithmen sollte sich besonders in relativ komplexen sequentiellen Entscheidungsprozessen zeigen, in denen nur wenig Expertenwissen vorhanden ist.

Die Evaluierungsdaten, gesammelt von Zweikämpfen paarweise verschiedener Spieler, sowie die Trainingsmetriken, zeigen, ein AlphaZero-Modell mit ausreichender Informationskapazität sei fähig dazu, schrittweise den zugrundeliegenden Markov-Entscheidungsprozess zu lernen, und weise eine stark positive Leistungskurve auf, sofern die Trainingsvariablen korrekt konfiguriert sind. Dieser Trend verdankt sich der Faustregel: Je größer sei der Trainingsbedarf und die Aufnahmefähigkeit eines Algorithmus, desto näher könne es dem zugrundeliegenden MDP kommen.

Diese Arbeit impliziert außerdem, der AlphaZero sei nicht nur in Brettspielen nützlich, sondern vermutlich auch in einer Vielzahl von Alltagsanwendungen wie z. B. Entwicklung leistungsstarkerer Spielerbots in Videospielen sowie KI-Unterstützung von finanziellen Marktentscheidungen. Vor allem in komplexen Domänen könnte die Entwicklung effizienter, lernfähiger Spielalgorithmen wie AlphaZero den hohen Bedarf an Computerressourcen erheblich reduzieren und folglich die Nachhaltigkeit fördern. Somit ermöglicht die moderne Technik nicht nur, umsonst wiederverwendbare Formen von künstlicher Intelligenz zu entwickeln, die weit über dem menschlichen Niveau liegen, sondern auch die Geschichte der Menschen mit dem nötigen Schwung vorwärtszutreiben und das Leben der Menschen zu revolutionieren.

Literaturverzeichnis

[1] Alphazero-gomoku. https://github.com/junxiaosong/AlphaZero_Gomoku/tree/master. Accessed: True.

[2] Bellman optimality principle. https://de.wikipedia.org/wiki/OptimalitÃd'tsprinzip_von_Bellman. Accessed: true.

[3] Chatgpt | openai. https://chat.openai.com. Accessed: True.

[4] Draw.io. http://www.draw.io. Accessed: True.

[5] Entropy (information theory). https://en.wikipedia.org/wiki/Entropy_(information_theory). Accessed: True.

[6] Gomoku by yamaceay. https://github.com/yamaceay/gomoku. Accessed: True.

[7] Kullback-leibler-divergenz. https://de.wikipedia.org/wiki/Kullback-Leibler-Divergenz. Accessed: True.

[8] Pytorch. https://pytorch.org. Accessed: True.

[9] Tuning the hyperparameters of an estimator. https://scikit-learn.org/stable/modules/grid_search.html. Accessed: True.

[10] Gopal Agrawal, Zalak Trivedi, and Seema Mahajan. Alphago–an ai in gaming.

[11] Katherina Babenkova. Varianten der monte-carlo tree search für das brettspiel gomoku unter berücksichtigung der sudden win/death problematik. https://doc.neuro.tu-berlin.de/bachelor/2023-BA-KatherinaBabenkova.pdf. Accessed: True.

[12] Nils Bjorck, Carla P Gomes, Bart Selman, and Kilian Q Weinberger. Understanding batch normalization. *Advances in neural information processing systems*, 31, 2018.

[13] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games*, 4(1):1–43, 2012.

[14] Xu Cao and Yanghao Lin. Uct-adp progressive bias algorithm for solving gomoku. In *2019 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 50–56. IEEE, 2019.

[15] Chih-Hung Chen, Shun-Shii Lin, and Yen-Chi Chen. An algorithmic design and implementation of outer-open gomoku. In *2017 2nd International Conference on Computer and Communication Systems (ICCCS)*, pages 26–30. IEEE, 2017.

[16] Vincent Christlein, Lukas Spranger, Mathias Seuret, Anguelos Nicolaou, Pavel Král, and Andreas Maier. Deep generalized max pooling. In *2019 International conference on document analysis and recognition (ICDAR)*, pages 1090–1096. IEEE, 2019.

[17] Peter Dayan and CJCH Watkins. Q-learning. *Machine learning*, 8(3):279–292, 1992.

[18] Janosch Deeg. Schlau, schlauer, am schlausten: Alphago zero. In *Künstliche Intelligenz: Vom Schachspieler zur Superintelligenz?*, pages 83–85. Springer, 2022.

[19] Frédérick Garcia and Emmanuel Rachelson. Markov decision processes. *Markov Decision Processes in Artificial Intelligence*, pages 1–38, 2013.

[20] Jürgen Groß. *Linear regression*, volume 175. Springer Science & Business Media, 2003.

[21] Zheng Hu, Jiaojiao Zhang, and Yun Ge. Handling vanishing gradient problem using artificial derivative. *IEEE Access*, 9:22371–22377, 2021.

[22] JH Kang and HJ Kim. Effective monte-carlo tree search strategies for gomoku ai. *International Journal of Circuit Theory and Applications*, 10:4841–4843, 2016.

[23] William T Katz and Son Pham. Experience-based learning experiments using go-moku. In *Proceedings of the 1991 IEEE International Conference on Systems, Man, and Cybernetics*, volume 2, pages 1405–1410, 1991.

[24] Donald E Knuth and Ronald W Moore. An analysis of alpha-beta pruning. *Artificial intelligence*, 6(4):293–326, 1975.

[25] Levente Kocsis and Csaba Szepesvári. Bandit based monte-carlo planning. In *European conference on machine learning*, pages 282–293. Springer, 2006.

[26] Levente Kocsis, Csaba Szepesvári, and Jan Willemson. Improved monte-carlo search. *Univ. Tartu, Estonia, Tech. Rep*, 1:1–22, 2006.

[27] Wen Liang, Chao Yu, Brian Whiteaker, Inyoung Huh, Hua Shao, and Youzhi Liang. Alphazero gomoku. *arXiv preprint arXiv:2309.01294*, 2023.

[28] Aditya Mahajan and Demosthenis Teneketzis. Multi-armed bandit problems. In *Foundations and applications of sensor management*, pages 121–151. Springer, 2008.

[29] Alessia Mammone, Marco Turchi, and Nello Cristianini. Support vector machines. *Wiley Interdisciplinary Reviews: Computational Statistics*, 1(3):283–289, 2009.

[30] Tom Michael Mitchell. *The discipline of machine learning*, volume 9. Carnegie Mellon University, School of Computer Science, Machine Learning . . . , 2006.

[31] Thomas M Moerland, Joost Broekens, Aske Plaat, Catholijn M Jonker, et al. Model-based reinforcement learning: A survey. *Foundations and Trends® in Machine Learning*, 16(1):1–118, 2023.

[32] Keiron O'Shea and Ryan Nash. An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*, 2015.

[33] Warren B Powell. *Approximate Dynamic Programming: Solving the curses of dimensionality*, volume 703. John Wiley & Sons, 2007.

[34] Stefan Reisch. Gobang ist pspace-vollständig. *Acta Informatica*, 13(1):59–66, 1980.

[35] Alfréd Rényi. On measures of entropy and information. In *Proceedings of the Fourth Berkeley Symposium on Mathematical Statistics and Probability, Volume 1: Contributions to the Theory of Statistics*, volume 4, pages 547–562. University of California Press, 1961.

[36] Jonathan Schaeffer. The history heuristic and alpha-beta search enhancements in practice. *IEEE transactions on pattern analysis and machine intelligence*, 11(11):1203–1212, 1989.

[37] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, et al. Mastering atari, go, chess and shogi by planning with a learned model. *Nature*, 588(7839):604–609, 2020.

[38] Antonio Serrano-Muñoz, Dimitrios Chrysostomou, Simon Bøgh, and Nestor Arana-Arexolaleiba. skrl: Modular and flexible library for reinforcement learning. *Journal of Machine Learning Research*, 24(254):1–9, 2023.

[39] Martin Sundermeyer, Ralf Schlüter, and Hermann Ney. Lstm neural networks for language modeling. In *Interspeech*, volume 2012, pages 194–197, 2012.

[40] János Szőts and István Harmati. Development of an incremental pattern extraction based gomoku agent. *Periodica Polytechnica Electrical Engineering and Computer Science*, 62(4):155–164, 2018.

[41] Zhentao Tang, Dongbin Zhao, Kun Shao, and LV Le. Adp with mcts algorithm for gomoku. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1–7. IEEE, 2016.

[42] Emanuel Todorov. Efficient computation of optimal actions. *Proceedings of the national academy of sciences*, 106(28):11478–11483, 2009.

[43] Daniel Whitehouse. *Monte Carlo tree search for games with hidden information and uncertainty*. PhD thesis, University of York, 2014.

[44] Hongming Zhang and Tianyang Yu. Alphazero. *Deep Reinforcement Learning: Fundamentals, Research and Applications*, pages 391–415, 2020.

[45] Dongbin Zhao, Zhen Zhang, and Yujie Dai. Self-teaching adaptive dynamic programming for gomoku. *Neurocomputing*, 78(1):23–29, 2012.

[46] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. Graph neural networks: A review of methods and applications. *AI open*, 1:57–81, 2020.

Anhänge

A. Komplexitätsanalyse von Gomoku

A.1. Klassifizierung von Gomoku anhand MDP-Eigenschaften

Die Aussagen über die Komplexität eines kombinatorischen Spiels wie Gomoku kann durch die Gemeinsamkeiten und Unterschiede argumentiert werden. Dabei ist es nützlich, das darunterliegende Markov-Entscheidungsprozess ("Markov Decision Process", abgekürzt als MDP) zu untersuchen.
Formal beschrieben ist ein MDP eine Tupel, das alle möglichen Zustände, Aktionen, Überführungswahrscheinlichkeiten und Bewertungen eines Spiels enkodiert (vgl. [18], S. 28). Anhand der folgenden Charakteristiken kann das darunterliegende MDP von Gomoku klassifiziert werden (vgl. [42], *2.1. Games with Hidden Information and Uncertainty*):

1. *1-Spieler-MDP vs. 2-Spieler-MDP*: In Markovketten mit nur einem Spieler wird ein existenzielles Suchproblem realisiert und die Bewertungen werden einfach an alle Vorgängerknoten propagiert. Jedoch wird in Markovketten mit zwei Spielern das Vorzeichen der propagierten Bewertung bei jeder Update negiert, so dass ein Maximierungsproblem in der Ebene t als ein Minimierungsproblem in der Ebene $t-1$ repräsentiert wird und vice versa. Gomoku ist ein 2-Spieler-Spiel und realisiert dabei ein Minimax-Problem.

2. *Endliche vs. Unendliche MDP*: Ein endliches Spiel lässt sich dadurch beschreiben, dass die Längen aller möglichen Markovketten eines Spiels nach oben hin begrenzt sind, gegeben eine endliche Spielkonfiguration. Ein unendliches Spiel besitzt jedoch unendlich lange Zugfolgen und üblicherweise existieren in solchen Spielen zyklische Spielzustände, die beliebig oft wiederholt werden können. Die Anzahl der möglichen Züge ist in Gomoku unmittelbar durch die gesamte Anzahl der Felder $M \cdot N$ beschränkt, somit ist das MDP stets endlich für eine beliebige (M, N, K)-Konfiguration.

3. *Deterministische vs. Stochastische MDP*: In deterministischen Fällen überführen die Aktionen von einem Zustand zu genau einem Zustand. In stochastischen Fällen wird aber zusätzlich die Wahrscheinlichkeitsverteilung der Überführungsfunktion gebraucht, um die Aktionen probabilistisch durchzuführen. In Gomoku werden die Züge unmittelbar durchgeführt, also ist das MDP deterministisch.

4. *Sequentielle vs. Parallele MDP*: In einem sequentiellen Spiel wird die Spielreihenfolge eindeutig festgelegt und somit ist sie eine lineare Ordnung, wohingegen ein paralleles Spiel mehrere Möglichkeiten zur Spielreihenfolge besitzt. In Gomoku spielen beide Spieler abwechselnd, somit ist das MDP sequentiell.

5. *MDP vs. POMDP*: In normalen Fällen kann jeder Spieler den Spielzustand vollständig beobachten. Aber in Fällen imperfekter Information muss als Ersatz dazu nur eine Liste von Beob-

achtungen (observations) bereitgestellt werden. In Gomoku müssen beide Spieler perfekt informiert über den aktuellen Spielzustand sein, also ist das MDP vollständig beobachtbar. Sonst würde bspw. die Regel verletzt, dass ein Feld höchstens nur einmal gesetzt werden kann.

A.2. Worst-Case-Laufzeit von Gomoku

Es sei ein Spiel mit einem Spielbrett der Größe $n := M \times N$, das bis zu einem gewählten Zeitpunkt $p \in 1, 2, \ldots, n$ gespielt wurde. Dann ist die Anzahl der möglichen Zugfolgen gleich $\frac{n!}{(n-p)!} = \prod_{i=0}^{p-1}(n-i)$. Wenn $p = n$ gewählt wird, ergibt sich eine Laufzeit-Komplexität von $O(n!) \subseteq O(n^n)$ analog zu [22]. Daher besitzt ein Gomoku-Spiel sowohl einen Branchfaktor von n als auch eine Suchtiefe von n.

A.3. Musterbasierte Enkodierung von Gomoku

In meisten Implementierungen von ADP Modellen wird der Spielzustand in Gomoku hauptsächlich in einem besonderen Binärformat enkodiert, jeweils 0 oder 1, vermutlich um die Anzahl der möglichen Eingabekombinationen zu beschränken.

Sei P die Anzahl der bekannten Spielmuster, durch deren Kombinationen sich einen Spielzustand als Ganzes beschreiben lässt. In einer Expertenanalyse [44] wurden $P = 20$ verschiedene Muster vorgegeben.

Eines von den Mustern ist Victory Continuous Four, also eine ununterbrochene Reihe von 4 Steinen gleicher Farbe, und ein typisches Spiel darf theoretisch höchstens nur eines je eine Farbe besitzen, weil es im Folgezug unmittelbar zu einem Sieg oder einem Verlust führt. Das heißt, eine Bit pro Farbe reicht vollkommen aus, Victory Continuous Four zu beschreiben.

Die restlichen Muster können jedoch mehrmals existieren. Die Anzahl eines Musters n wird (außer VCF) wie folgt dargestellt: 00000 wenn keines vorhanden ist, 10000 wenn nur eines, 11000 wenn zwei, 11100 wenn drei, 11110 wenn vier und $(1111(n-4)/2)$ wenn größer als vier.

Aus informationstheoretischer Sicht ist es zwar eine redundante Enkodierung, weil die Anzahl der auf 1 gesetzte Bits monoton mit steigender Anzahl des Vorkommens steigt. Allerdings werden effizientere Formen wie Binärkodierung nur seltener verwendet, es sei denn, das Modell verfügt über lernfähige Feature-Extraction-Layers selbst (z. B. konvolutionelle Layers). Eine mögliche Begründung ist, ML-Modelle wären dazu weniger fähig, effizientere Formen zu entschlüsseln.

Bemerkenswert ist es auch, dass die exakte Angabe ab einer bestimmten Anzahl vermieden wird (üblicherweise ”mindestens 5”). Es wird davon ausgegangen, dass die genaue Anzahl der Vorkommen ab einer bestimmten Anzahl nur einen geringfügigen Unterschied macht. Insgesamt ergibt sich dann $10 \cdot P - 8$.

Als Nächstes werden für jedes Muster zwei binäre Informationen hinzugefügt: 10 bzw. 01 wenn das Muster vorkommt und der erste bzw. zweite Spieler dran ist, sonst 00. Als Letztes wird die

Offensive/Defensive-Information wie folgendes hinzugefügt: 10 bzw. 01 wenn der erste bzw. zweite Spieler spielt. Insgesamt ergibt sich $14 \cdot P - 6$ Inputvariablen.
Die musterbasierte Enkodierung ist zwar eine gut informierte Heuristik, die nur von einem gewählten Gewinnlänge k abhängig ist und für ein Spielbrett beliebiger Größe angewendet werden kann. Jedoch bringt es weitere Probleme mit sich.

Beispielsweise ist ein solcher Ansatz nicht gut für Endspiele geeignet und die Spielmuster mit vollkommen besetzten Feldern seien uninteressant in den Augen des Gomoku-Spielers, da die Positionen nicht geändert oder rückgängig gemacht werden können.

Außerdem muss davon ausgegangen werden, dass das Expertenwissen dafür ausreicht, einen beliebigen Spielzustand vollständig anhand vorgegebenen Spielmustern zu beschreiben. Allerdings ist es im heutigen Wissensstand noch unbekannt, welche vorhandenen Spielmuster ”essenziell” sind.