OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Fakultät
Informatik und Mathematik

Projektbericht

zum HSP1 im Wintersemester 2019/20 2020

Implementierung von Reversi mit dem AlphaZero-Ansatz

Autoren: simon1.hofmeister@st.oth-regensburg.de
nadiia1.matsko@st.oth-regensburg.de
monika.silber@st.oth-regensburg.de
simon.wasserburger@st.oth-regensburg.de

Leiter: Prof. Dr. rer. nat. Carsten Kern

Abgabedatum: 15.03.2020

Inhaltsverzeichnis

1 Einleitung **1**
1.1 Reversi . . . 1
1.2 ReversiXT . . . 1
1.3 AlphaGo . . . 1
1.4 AlphaZero . . . 2

2 Related Work **4**
2.1 Monte Carlo Tree Search . . . 4
2.2 Neuronales Netz . . . 6
2.3 Reinforcement Learning . . . 6
2.4 Zusammenspiel von MCTS und NN . . . 6

3 Implementierung **7**
3.1 Monte Carlo Tree Search . . . 7

4 Organisation **9**
4.1 Team und Aufgabenverteilung . . . 9
4.2 Kommunikation . . . 9
4.3 Versionskontrolle . . . 9
4.4 OS, IDE und Programmiersprache . . . 9
4.5 Testumgebungen . . . 9
4.6 Projekt-Dokumentation . . . 9

5 Fazit **10**

Anhang **I**

1 Einleitung

Das Ziel des Projektes ist es das Reversi Spiel mit Ansatz von künstlicher Intelligenz zu implementieren, indem AlphaZero Framework und Montecarlo Tree Search Verfahren angewendet werden.

1.1 Reversi

1.2 ReversiXT

1.3 AlphaGo

AlphaGo ist ein Computerprogramm, das das Brettspiel Go spielt[New16]. Es wurde von DeepMind Technologies entwickelt, das später von Google übernommen wurde. AlphaGo hatte drei weitaus mächtigere Nachfolger, genannt AlphaGo Master, AlphaGo Zero und AlphaZero [Dee].

Im März 2016 schlug AlphaGo Lee Sedol in einem Fünf-Spiel-Match, das erste Mal, dass ein Computer-Go-Programm einen 9-Dan-Profi ohne Handicap besiegte [Dee16]. Obwohl es im vierten Spiel gegen Lee Sedol verlor, trat Lee im Endspiel zurück und gab im Endergebnis 4 zu 1 zu Gunsten von AlphaGo. In Anerkennung des Sieges wurde AlphaGo von der Korea Baduk Association mit einem Ehren-9-Dan ausgezeichnet [Str16]. Der Vorsprung und das Herausforderungsspiel mit Lee Sedol wurden in einem Dokumentarfilm mit dem Titel AlphaGo [Koh17] unter der Regie von Greg Kohs dokumentiert. Er wurde am 22. Dezember 2016 von Science als einer der zweiten Durchbruch des Jahres gewählt [Sta16].

AlphaGo und seine Nachfolger verwenden einen Monte Carlo Baumsuch-Algorithmus, um seine Züge auf der Grundlage von Wissen zu finden, das zuvor durch maschinelles Lernen „gelernt “ wurde, insbesondere durch ein künstliches neuronales Netz (eine deep learning Methode) durch ausführliches Training, sowohl durch menschliches als auch durch Computerspiel [SHM+16]. Ein neuronales Netz wird trainiert, um AlphaGos eigene Zugauswahlen und auch die Partien der Gewinner vorherzusagen. Dieses neuronale Netz verbessert die Stärke der Baumsuche, was zu einer höheren Qualität der Zugauswahl und einem stärkeren Selbstspiel in der nächsten Iteration führt.

AlphaGo Spielstil

Im Spiel gegen einen Top-Go-Spieler, hat das künstliche Intelligenzprogramm AlphaGo die Kommentatoren mit Zügen verwirrt, die oft als „schön“ beschrieben werden, aber nicht in den üblichen menschlichen Spielstil passen [Rib16].

Howard Yu,Professor für strategisches Management und Innovation an der IMD Business School meinte, dass AlphaGo eine Maschine darstellt, die nicht nur denkt, sondern auch lernen und Strategien entwickeln kann [Rib16].

Toby Manning, der Match-Schiedsrichter für AlphaGo vs. Fan Hui, hat den Stil des Programms als „konservativ “ beschrieben [Gib16]. Der Spielstil von AlphaGo begünstigt stark die größere

Wahrscheinlichkeit, mit weniger Punkten zu gewinnen, gegenüber der geringeren Wahrscheinlichkeit, mit mehr Punkten zu gewinnen [Rib16]. Seine Strategie, die Gewinnwahrscheinlichkeit zu maximieren, unterscheidet sich von dem, wozu menschliche Spieler neigen, nämlich territoriale Gewinne zu maximieren und erklärt einige seiner seltsam aussehenden Züge [Cho16].

1.4 AlphaZero

AlphaZero ist ein Computerprogramm, das von der Forschungsfirma *DeepMind* für künstliche Intelligenz entwickelt wurde, um die Spiele Schach, Shogi und Go zu meistern. Der Algorithmus verwendet einen ähnlichen Ansatz wie AlphaGo Zero [SHS+17].

Das neuronale Netz von AlphaGo Zero weiß nichts über das Spiel jenseits der Regeln. Im Gegensatz zu früheren Versionen von AlphaGo nahm AlphaGo Zero nur die Steine des Bretts wahr, anstatt einige seltene, vom Menschen programmierte Randfälle zu haben, die helfen, ungewöhnliche Go-Brettstellungen zu erkennen. Die KI beschäftigte sich mit dem Reinforcement Learning und spielte gegen sich selbst, bis sie ihre eigenen Züge und deren Auswirkungen auf den Ausgang des Spiels vorhersehen konnte [Gre17].

AlphaZero ersetzt das handgemachte Wissen und die domänenspezifischen Erweiterungen, die in traditionellen Spielprogrammen verwendet werden, durch tiefe neuronale Netze, einen universellen Reinforcement Learning-Algorithmus und einen universellen Baumsuch-Algorithmus [SHS+18]. Statt einer handgemachten Auswertungsfunktion und Heuristik für die Zugreihenfolge verwendet AlphaZero ein tiefes neuronales Netzwerk. Die Parameter des tiefen neuronalen Netzes in AlphaZero werden durch Reinforcement Learning trainiert, indem AlphaZero mit sich selber spielt und Parameter zufällig initialisiert. Statt einer Alpha-Beta-Suche mit domänenspezifischen Erweiterungen verwendet AlphaZero einen universellen Monte-Carlo-Baumsuch-Algorithmus [SHS+18]. AlphaZero ist eine verallgemeinerte Variante des AlphaGo Zero Algorithmus und kann neben Go auch Shogi und Schach spielen. Die Unterschiede zwischen AlphaZero und AlphaGo Zero sind: [SHS+18]

1. AZ hat fest programmierte Regeln für die Einstellung von Such-Hyperparametern.
2. Das neuronale Netz wird nun ständig aktualisiert.
3. Go ist (im Gegensatz zu Schach) unter bestimmten Reflexionen und Rotationen symmetrisch; AlphaGo Zero wurde programmiert, um diese Symmetrien auszunutzen. AlphaZero ist es nicht.
4. Schach kann im Gegensatz zu Go mit einem Unentschieden enden; daher kann AlphaZero die Möglichkeit einer unentschiedenen Partie in Betracht ziehen.

Im Jahr 2019 veröffentlichte DeepMind einen neuen Artikel über MuZero, einen neuen Algorithmus, der in der Lage ist, auf AlphaZero Arbeit zu verallgemeinern, indem er sowohl Atari

als auch Brettspiele ohne Kenntnis der Regeln oder Darstellungen des Spiels spielt [**?**].

2 Related Work

2.1 Monte Carlo Tree Search

Der MCTS findet als Alternative zum Alpha-Beta-Search Anwendung. Da der Alpha-Beta-Ansatz einen geringen Verzweigungsgrad und eine angemessene Bewertungsfunktion fordert, ist dessen in Anwendung in Brettspielen, die diese Bedingungen nicht erfüllen, ungeeignet. Der MCTS hingegen bewies sich im Umgang mit solchen Situation [CBSS08]. Basierend auf einer Bestensuche und stochastischer Simulation wählt der MCTS bei jedem Durchlauf den erfolgversprechendsten Knoten als nächsten Spielzug aus. Entsprechend wird ein Baum aufgebaut, in dem ein Knoten einen konkreten Spielzustand widerspiegelt[CBSS08]. Dabei werden die drei Schritte Expansion, Simulation und Backpropagation durchgeführt [CBSS08]. Falls der aktuelle Spielzustand noch nicht als Knoten existiert, wird der Baum zunächst expandiert. Um nun die beste Aktion zu ermitteln, werden Spiele ausgehend vom aktuellen Zustand bis hin zum Spielende simuliert. Dabei werden valide Spielzüge zufällig ausgewählt. Hierbei ist jedoch zu beachten, dass eine reine Zufallsauswahl, die impliziert, dass die Selektion aller Möglichkeiten gleich wahrscheinlich ist, in eher primitivem Spielverhalten resultiert. Mithilfe einer Heuristik können daher aussichtsreichere Spielzüge favorisiert werden. Innerhalb eines Playouts durchlaufene Nodes werden schließlich aktualisiert, indem vermerkt wird, dass sie einmal mehr besucht wurden und welches Spielergebnis sich ergeben hat [CBSS08].

Anzumerken ist, dass zwei verschiedene Policies genutzt werden. Für die Erweiterung des Baums wird eine Tree Policy angewendet, die besagt, dass entsprechende Blattknoten an bereits vorhandene, unbesuchte Knoten angefügt werden. Des Weiteren legt die Default Policy die Simulation fest. Hierbei wird in einem nichtterminalen Spielzustand, der gewöhnlich dem neu hinzugefügt Blattknoten entspricht, ein zufälliges Spiel durchlaufen, um ein ein Spielergebnis zu ermitteln [BPW+12].

Der MCTS bleibt so lange aktiv, bis er unterbrochen wird, beispielsweise aufgrund von abgelaufener Rechenzeit. Der zu diesem Zeitpunkt als am erfolgreichsten ermittelte Knoten beziehungsweise Spielzug steht daraufhin fest [BPW+12].

Zu einem Knoten gehören der entsprechende Spielzustand, den er widerspiegelt, der Spielzug aus dem er resultierte, sowie der aus Simulationen resultierende Reward und wie oft er besucht wurde [BPW+12].

Nachfolgend werden die konkreten Umsetzungsdetails des MCTS in AlphaGo Zero betrachtet, so wie sie von Silver et al. angewendet wurden [SSS+17]. In AlphaGo Zero besteht der Selection-Schritt daraus, ausgehend von einem Wurzelknoten eine Simulationen bis hin zu einem Blattknoten zu durchlaufen. Dafür wird in jedem Schritt der Folgeknoten ausgewählt, der den maximalen UCT-Wert enthält [SSS+17].

Entsprechend dem AlphaZero-Ansatz findet bei der Simulation keine zufällige Auswahl des Spielzuges statt, stattdessen wird eine Variante des Upper Confidence Bound (UCB) angewendet. Die konkrete Abwandlung ist der polynomiale UCB applied to Trees (UCT). Dieser errechnet sie wie folgt:

$$UCT = \frac{Q(n_i)}{N(n_i)} cP(n) \frac{\sqrt{N(n)}}{1 + N(n_i)} \tag{1}$$

Wobei c eine Konstante darstellt, die das Maß an Exploration festlegt und P(s,a) für die a-priori-Wahrscheinlichkeit für die Auswahl des jeweiligen Spielzugs steht. Letztlich wird stets der Spielzug ausgewählt, der den maximalen UCT-Wert darstellt [SSS+17].

Ferner ist an dieser Stelle anzumerken, dass der UCB den Kompromiss zwischen Exploration und Exploitation widerspiegelt. Der erste Quotient der UCT-Gleichung steht für das Maß der Ausbeutung und lenkt den Algorithmus dahingehend vielversprechende Knoten weiter zu besuchen. Im Gegensatz dazu wird dazu angehalten Bereiche, die noch nicht oft aufgesucht wurden, verstärkt zu untersuchen. Dies wird als Erkundung bezeichnet und durch den letzten Term des UCT abgebildet. Wichtig hierbei ist es ein Gleichgewicht der beiden Komponenten zu finden [BPW+12].

Die klassische Simulation von zufälligen Spieldurchläufen entfällt im MCTS vollständig, da noch nicht expandierte Knoten an das NN weitergegeben und dort evaluiert wird. Sobald ein solcher Blattknoten erreicht ist, wird dieser dem NN übergeben, woraufhin die a-priori-Wahrscheinlichkeit und die Bewertung des Spielzugs ermittelt werden. Daraufhin ist der Blattknoten expandiert und alle ausgehenden Kanten beziehungsweise Kinderknoten werden mit initialen Werten belegt. Anschließend erfolgt die Backpropagation, indem in allen durchlaufenen Knoten die Gewinnwahrscheinlichkeit aktualisiert, sowie die Anzahl der Besuche um den Wert Eins inkrementiert wird [SSS+17].

Letztlich wird ein konkreter Spielzug ausgehend vom Wurzelknoten selektiert. Dabei wird der Knoten, der am häufigsten besucht wurde als der beste Spielzug aufgefasst. Der ausgewählte Kindknoten wird der neue Wurzelknoten. Der von ihm ausgehend aufgebaute Teilbaum wird beibehalten, während der der restliche Baum verworfen wird [SSS+17].

2.2 Neuronales Netz

2.3 Reinforcement Learning

2.4 Zusammenspiel von MCTS und NN

AlphaGoZero und AlphaZero basieren auf der Kombination des MCTS und des NN. Dabei gibt es zwei Schnittstellen zwischen diesen beiden Komponenten. Erstere liegt in der bereits erwähnten Bewertung von Blattknoten. Der MCTS durchläuft keinen simulierten Spielablauf, sondern überlässt die Evaluation des Knotens dem NN und arbeitet mit den zurückgegeben Daten weiter. Dies spiegelt die policy evaluation wider [SSS+17].

Eine weitere Verzweigung von MCTS und NN tritt bei dem Update der Parameter des NN auf. Das NN ermittelt zu jedem Spielstand die möglichen Züge und gibt deren Gewinnwahrscheinlichkeit, sowie die Wahrscheinlichkeit für die Auswahl des Zuges an. Für die Aktualisierung der Parameter werden die genannten Werte dahingehend angepasst, dass sie den im MCTS ermittelten Werten entsprechend beziehungsweise sich diesen annähern. Eine Anpassung in diese Richtung ist sinnvoll, da die Daten im MCTS als deutlich genauer gelten. Dieser Vorgang entspricht der policy improvement [SSS+17].

3 Implementierung

3.1 Monte Carlo Tree Search

Um den MCTS für Reversi zu realisieren wurden die Klassen MCTS und Node angelegt. Letztere enhält zwei überladene Konstruktoren zum Anlegen von Wurzel- und Kindknoten. Ein Node enthält zusätzliche Attribute. Sowohl der Elternknoten als auch eine ArrayList vom Typ Node, die die direkten Kinder enthält, werden abgespeichert. Die Anzahl, wie oft ein Knoten besucht wurde, wird in der Variable `numVisited` hinterlegt. Außerdem gespeichert wird das Ergebnis eines simulierten Spieldurchlaufs in `simulationReward`. Da der aktuelle Spielzustand durch das Environment, das ebenfalls die derzeitige Repräsentation des Playgrounds beinhaltet, definiert ist, wird dies ebenfalls im Node hinterlegt. Des Weiteren wird in dem Attribut `nextPlayer` festgehalten, welcher Spieler als Nächstes an der Reihe ist.

Es gibt einen überladenen Konstruktor, der einerseits für das Anlegen einer neuen Wurzel und andererseits für das Erzeugen eines neuen Kinderknotens zuständig ist. Bei der Instanziierung durch die Konstruktoren werden sinnvolle initiale Werte vergeben. numVisited und simulationReward werden auf 0 beziehungsweise 0.0 gesetzt. Für den Wurzelknoten gilt, dass er keinen Parent besitzt, für alle weiteren Knoten wird der Parent übergeben und gesetzt. Die Children werden zunächst durch eine leere Liste initialisiert. Der `nextPlayer` wird ebenfalls übergeben und gesetzt. Außerdem zu erwähnen ist die Methode `calculateUCT()`, die den Upper Confidence Bound applied to Trees (UCT) für einen Knoten berechnet. Sie ermittelt zunächst die Exploitation-Komponente, indem sie den simulationReward durch die Anzahl an Besuchen dividiert. Die Exploration berechnet sich aus dem Verhältnis, wie oft der Parent besucht wurde, geteilt durch den inkrementierten Wert, wie oft der aktuelle Knoten besucht wurde. Aus dem Quotient wird anschließend die Wurzel gezogen. Um den Kompromiss zwischen diesen beiden Komponenten zu kontrollieren, wird die Exploration mit der A-priori-Wahrscheinlichkeit für einen Zug multipliziert. Diese wird im neuralen Netz trainiert.

Hinsichtlich der Klasse MCTS ist festzuhalten, dass diese in der Klasse Agent über den Konstruktoraufruf instanziiert wird. Dieser verlangt das Environment und den Player als Übergabeparameter und legt daraufhin einen neuen Wurzelknoten an, sowie eine leere ArrayList vom Typ Node, die die Blattknoten beinhaltet, die im späteren Verlauf simuliert werden müssen. Für die Simulation muss beachtet werden, dass das Environment geklont und somit eine tiefe Kopie erzeugt werden muss, damit der tatsächliche Spielzustand nicht unbeabsichtigt manipuliert wird. Dabei ist festzuhalten, dass dies zweimal stattfindet. Einmal, wenn ein neuer Baum aufgebaut wird, somit erhält der neue Wurzelknoten und ebenfalls jedes seiner Kinder jeweils einen eigenen Klon. Für die Kinderknoten gilt, dass diese ihre Environment-Instanz an ihre Kinder weitergeben und diese somit innerhalb derselben Instanz agieren. Um den MCTS zu starten, wird die Methode `searchBestTurn()` aufgerufen. Diese expandiert zunächst den Wurzelknoten, indem

sie alle im aktuellen Zustand möglichen validen Züge ermittelt und durch diese iteriert. Die Methode `getPossibleTurns()` gibt diese zurück. Sie iteriert über das gesamte Spielfeld und prüft dabei mithilfe der Methode `validateTurnPhase1()` im Environment, an welcher Stelle ein gültiger Zug gemacht werden kann. Für jeden dieser Züge wird in `expand()` der Kinderknoten angelegt, sowie als unbesuchter Blattknoten abspeichert. Außerdem wird ermittelt, welcher Spieler als Nächster einen Zug machen darf und der simulierte derzeitige Spielzustand anhand des Zuges aktualisiert. Nachdem ein Knoten expandiert wurde, wird er aus der Liste der Blattknoten wieder entfernt. Daraufhin werden die unbesuchten Blattknoten, die am Anfang den Kinderknoten der Wurzel entsprechen, in der Methode `traverse()` durchlaufen. Dabei wird in jedem dieser eine Simulation gestartet, die einen Spielverlauf bis zum Spielende anhand zufällig ausgewählter möglicher Züge durchspielt.

Mithilfe der Funktion `simulate()` erfolgt die Simulation eines Spiels. Ein Zufallszug wird durch eine Instanz der Java-Klasse Random generiert. Hierbei wird ein zufälliger Integer erzeugt, der durch eine Modulo-Operation auf den Größenbereich abgestimmt wird, der dem der Anzahl der möglichen Züge entspricht. Die resultierende Zahl gibt den auszuwählenden Zug innerhalb der ArrayList an.

Wenn keine weiteren validen Folgezüge ermittelt werden können, bedeutet das das Spielende und der Reward für die Spielausgang wird anhand der Funktion `rewardGameState()` berechnet. Diese erhält als Parameter das Environment, sowie den Spieler, für den der Reward kalkuliert werden soll. Indem der gesamte Playground durchlaufen und gezählt wird, wie viele Steine vom übergebenen Spieler enthalten sind, errechnet sich die Bewertung des Spiels. Abschließend werden die Werte für Anzahl Besuche und Reward ebenfalls im Wurzelknoten aktualisiert.

Nach Abschluss der Simulation wird der Reward zurückgegeben. Daraufhin wird in `traverse()` die Backpropagation der Ergebnisse durchgeführt, indem iterativ vom aktuellen Knoten bis hoch zur Wurzel die Anzahl an Besuchen inkrementiert und der Reward entsprechend erhöht wird.

4 Organisation

4.1 Team und Aufgabenverteilung
4.2 Kommunikation
4.3 Versionskontrolle
4.4 OS, IDE und Programmiersprache
4.5 Testumgebungen
4.6 Projekt-Dokumentation

5 Fazit

Literatur

[BPW+12] C. B. Browne, E. Powley, D. Whitehouse, S. M. Lucas, P. I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, Samthrakis S., and S. Colton. A survey of monte carlo tree search methods. In *IEEE Transactions on Computational Intelligence and AI in games*, volume 4(1), pages 1–43, März 2012.

[CBSS08] G. Chaslot, S. Bakkes, I. Szita, and P. Spronck. Monte-carlo tree search: A new framework for game ai. In *Proceedings of the Fourth Artificial Intelligence and Interactive Digital Entertainment Conference*, Oktober 2008.

[Cho16] Tanguy Chouard. The go files: Ai computer clinches victory against go champion, 03 2016. Retrieved: 25.12.2019.

[Dee] DeepMind. Deepmind. Retrieved: 25.12.2019.

[Dee16] DeepMind. Match 1 - google deepmind challenge match: Lee sedol vs alphago, 03 2016. Retrieved: 25.12.2019.

[Gib16] Elizabeth Gibney. Google ai algorithm masters ancient game of go, 12 2016. Retrieved: 25.12.2019.

[Gre17] Larry Greenemeier. Ai versus ai: Self-taught alphago zero vanquishes its predecessor, 10 2017. Retrieved: 25.12.2019.

[Koh17] Greg Kohs. Alphago, 2017. Retrieved: 25.12.2019.

[New16] BBC News. Artificial intelligence: Google's alphago beats go master lee se-dol, 03 2016. Retrieved: 25.12.2019.

[Rib16] John Ribeiro. Alphago's unusual moves prove its ai prowess, experts say, 03 2016. Retrieved: 25.12.2019.

[SHM+16] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, January 2016.

[SHS+17] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy P. Lillicrap, Karen Simonyan, and Demis Hassabis. Mastering chess and shogi by self-play with a general reinforcement learning algorithm. *CoRR*, abs/1712.01815, 2017.

[SHS+18] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[SSS+17] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis. Mastering the game of go without human knowledge. In *nature*, volume 550(7676), pages 354–359, März 2017.

[Sta16] Science News Staff. From ai to protein folding: Our breakthrough runners-up, 12 2016. Retrieved: 25.12.2019.

[Str16] The Straitstimes. Google's alphago gets 'divine' go ranking, 03 2016. Retrieved: 25.12.2019.

Anhang