OSTBAYERISCHE
TECHNISCHE HOCHSCHULE
REGENSBURG

Fakultät
Informatik und Mathematik

Projektbericht

zum HSP1 im Wintersemester 2019/20

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

Das Ziel des vorliegenden Projektes ist es das Spiel „Reversi“ mithilfe von Methoden aus dem Bereich der künstlichen Intelligenz zu implementieren. Dazu wird der entsprechende Reinforcement-Learning-Ansatz von AlphaZero beziehungsweise AlphaGo Zero angewendet, der ein neuronales Netz in Kombination mit dem Monte-Carlo-Suchverfahren nutzt.

1.1 Reversi

In diesem Abschnitt werden die grundlegenden Spielregeln[1] von Reversi, so wie sie ebenfalls im vorliegenden Projekt Anwendung finden, vorgestellt.

Reversi ist ein kompetitives Spiel, bei dem die Teilnehmer in einer vorgegebenen Reihenfolge Züge machen dürfen. Bei zwei Spielern bedeutet das, dass sich diese abwechseln, bei mehr als zwei geht es reihum. Jeder Spieler hat eine Farbe beziehungsweise ein Symbol für seine gelegten Steine. Ein Zug besteht darin, den eigenen Spielstein angrenzend an einen gegnerischen Stein so auf dem Spielfeld zu platzieren, dass mindestens ein gegnerischer eingeschlossen wird, wobei es stattdessen außerdem möglich ist auszusetzen. Sobald der eigene Spielstein positioniert ist, werden alle gegnerischen, die zwischen dem neuen und bereits vorhandenen Steinen liegen, so umgekehrt, dass sie die Farbe oder das Symbol des derzeitigen Spielers annehmen.

Üblicherweise ist das Spielfeld ein Quadrat der Größe 8x8, wobei in Abwandlungen andere Dimensionen grundsätzlich möglich sind. Ein typischer Startzustand zeigt dich darin, dass bereits zwei Steine der beiden Spieler, also ingesamt vier, in der Mitte des Feldes in Form einer 2x2 Anordnung platziert sind. Die Steine der jeweiligen Teilnehmer sind dabei diagonal angeordnet.

Ziel ist es, möglichst viele Steine der eigenen Farbe oder mit dem eigenen Symbol auf dem Feld zu haben. Das Spiel ist beendet, sobald beide Spieler direkt hintereinander passen beziehungsweise beide keine Züge mehr machen können. Gewonnen hat entsprechend derjenige, der mehr Spielsteine auf dem Feld liegen hat. Falls beide Teilnehmer die gleiche Anzahl haben, so ist der Spielausgang unentschieden.

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

[1] $https://de.wikipedia.org/wiki/Othello_(Spiel)$

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 [SAH+19].

2 Theoretische Grundlagen

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].

Zur Visualisierung dieses Vorgehens dienen Abbildung 1, sowie Abbildung 2. Als erster Schritt erfolgt die Auswahl eines Knotens analog des „Selection“-Schrittes in Abbildung 1.

Falls der aktuelle Spielzustand noch nicht als Knoten existiert, wird der Baum zunächst expandiert [CBSS08]. Dies kann in Abbildung 1 im Schritt „Expand“eingesehen werden und resultiert darin, dass die nächstmöglichen Aktionen als Kinderknoten an den aktuell betrachteten Knoten angehängt werden.

Um nun die beste Aktion zu ermitteln, werden Spiele ausgehend vom aktuellen Zustand bis hin zum Spielende simuliert. Dabei werden valide Spielzüge zufällig ausgewählt [CBSS08], wie „Simulation“in Abbildung 1 veranschaulicht. An dieser Stelle werden nicht alle vorhandenen Folgepositionen eines Knotens berücksichtigt, sondern lediglich eine davon. Der letzte Knoten repräsentiert den finalen Spielstand.

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].

Dieser Prozess ist in Abbildung 2 dargestellt. in dieser Grafik wurde der Baum um numerische Werte der Form „X|Y“ergänzt, wobei X für die Anzahl an Besuchen steht und Y für die Anzahl, wie oft das Spiel ausgehend vom betrachteten Knoten gewonnen wurden. Die Zahl „0“signalisiert dabei das Verlieren, die Zahl „1“das Gewinnen des simulierten Spiels. Bevor der Simulationsdurchlauf gestartet wird hat der ausgewählte Knoten den Wert „1|1“. Am Ende angekommen, wird der Spielausgang ermittelt, wodurch der entsprechende Knoten zur Veranschaulichung mit dem entsprechenden Wert ergänzt wird (siehe „1“ - „1“ - „0“). Daraufhin werden die Resultate unten angefangen bis hin zum Wurzelknoten durch alle involvierten Nodes zurückgeleitet. Der anfangs betrachtete Knoten hat nun den Wert „4|3“, da er dreimal öfter

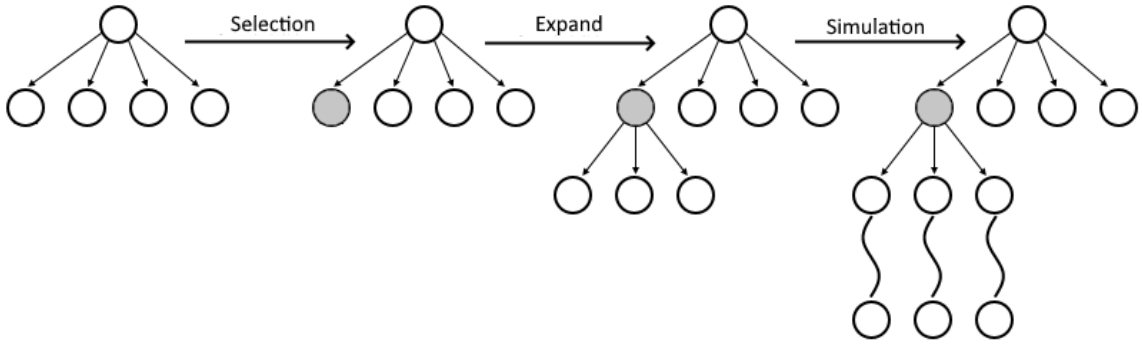

Abbildung 1: Auswahl eines Knotens mit nachfolgender Expansion und Simulation, eigene Abbildung

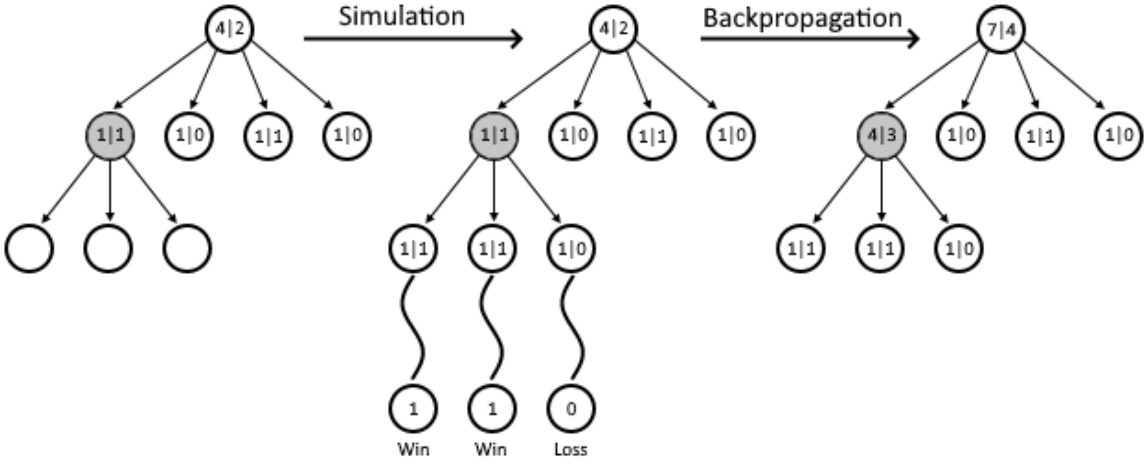

Abbildung 2: Simulation und anschließende Backpropagation der entsprechenden Werte, eigene Abbildung

besucht wurde, wobei das Spiel letztendlich zweimal davon gewonnen wurde. Im Wurzelknoten auf oberster Ebene kann entsprechend eine Änderung der Werte von „4|2“ auf „7|4“ festgestellt werden.

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 abge-

laufener 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 wird eine Variante des Upper Confidence Bound (UCB) zur Auswahl der Knoten 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]. Für Abbildung 1 bedeutet das, dass dies im „Selection“-Schritt der grau hinterlegte Knoten wäre.

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]. Im Vergleich zu Abbildung 2 bedeutet das, dass zur Expansion des jeweiligen Knotens nicht wie bei der Simulation weiter in die Tiefe gegangen wird, um letztendlich die Endewerte „1“- „1“- „0“zu ermitteln, sondern die vom NN berechneten Werte zur Verfügung stehen. Die Backpropagation bleibt hierbei gleich.

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

Ein Neuronales Netz (NN) wird durch eine Vielzahl einzelner miteinander verknüpfter Neuronen beschrieben. Dabei erhalten sie Signale von anderen Neuronen als Input, verarbeiten diese und geben sie als Output an andere weiter. Diese Eingangswerte werden als Vektor X $= \{x_1, x_2, ..., x_n\}$ repräsentiert. Um adäquate Ausgangswerte zu erhalten werden die einzelnen Input-Elemente durch die entsprechenden Komponenten im Vektor W $= \{w_1, w_2, ..., w_n\}$ gewichtet. Des Weiteren kann es zu jedem Neuron einen Bias b geben. Zur Verarbeitung der Eingangswerte wird eine Aktivierungsfunktion herangezogen, die Z $= \{w_1x_1, w_2x_2, ..., w_nx_n + b(W^TX+b)\}$ als Input erhält. Das sich hieraus ergebende Resultat entspricht $\hat{y}$. Im weiteren Verlauf wird versucht die Diskrepanz zwischen dem tatsächlichen Wert y und dem geschätzten Wert $\hat{y}$ durch Anwendung einer Verlustfunktion zu minimieren. Dieser Vorgang wird als „Backpropagation“bezeichnet. Dabei werden Fehler von dem letzten bis hin zum ersten layer zurückgeleitet, um alle beteiligten Gewichte in die entsprechende Richtung anzupassen. Dies wird dadurch erreicht, dass man das Minimum der Fehlerfunktion lokalisiert, das durch Annäherung an den Gradienten der Funktion ermittelt werden kann [Sew19, S. 75-79].

Neuronale Netze bestehen aus verschiedenen Schichten. Dabei existieren stets der input layer, ein oder mehrere hidden layer und der output layer. Während der input layer über genau so viele Neuronen, wie es Eingangswerte gibt, verfügt, weist der output layer so viele Neuronen auf, wie Ausgangsmerkmale vorhanden sind. Die Neuronenanzahl in den versteckten Zwischenschichten kann variieren. Sobald mehr als ein hidden layer vorliegt, spricht man von einem tiefen neuronalen Netz [Sew19, S. 77].

Hinsichtlich der Fehlerfunktion spricht man von einem „L1 loss“ beziehungsweise einem „L2 loss“, wenn der absolute respektive der quadratische Fehler berücksichtigt wird [Sew19, S. 82].

Wenn in einer Schicht alle Neuronen mit allen Elementen aus dem vorangegangen, sowie dem nachfolgenden layer verknüpft sind, wird diese als fully connected bezeichnet [Sew19, S. 79].

Convolutional Neuronal Networks (CNNs) beschreiben tiefe Netze, die für die Bildverarbeitung ausgelegt sind. Dabei wird die Grafik typischerweise dreidimensional dargestellt, mit der Bildhöhe und -breite, sowie den Farbkanälen in der Tiefe. Hierbei kann eine Schicht als convolutional layer eingebunden werden, die entsprechend eine Faltungsoperation anhand eines vorgegebenen Kerns auf das vorliegende Bild anwendet [Sew19, S. 85].

Ein weiteres wichtiges Hilfsmittel findet sich in der Batch Normalisation, mithilfe der Mittelwerte und Varianzen in den Eingabedaten der einzelnen Schichten normalisiert werden können. Das ist sinnvoll, da sich die Schichten sonst laufend an die neue Verteilung anpassen müssen. Der ausschlaggebende Vorteil der Normalisierung ist, dass das Modell deutlich effizienter trainiert werden kann, da höhere Lernraten angewendet werden können [IS15].

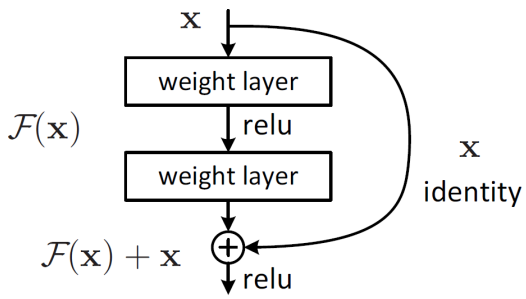

Abbildung 3: Darstellung einer möglichen Abkürzung in residual nets, Abbildung entnommen aus [HZRS16]

Tiefe NN sind schwer zu trainieren. Zur Vereinfachung dieser Problematik besteht die Möglichkeit Abkürzungen zu nutzen. Die Vorgehensweise wird in Abbildung 3 deutlich. Dabei können Daten (siehe „x“) bestimmte Schichten (vergleiche „weight layer“) überspringen und werden somit in diesen layern nicht verarbeitet, sondern gehen direkt weiter zu einem vorgegebenen Punkt im NN. Diese Methode wird „residual learning“ genannt und optimiert den Umgang mit tiefen Netzen [HZRS16].

Das in AlphaGo Zero verwendete NN zeigt nachfolgenden von [SSS+17] beschriebenen Aufbau. Zunächst werden die Eingabedaten in einem *residual block* verarbeitet. Dieser besteht aus einem *convolutional block* und nachfolgend 19 oder 39 residualen Blöcken. Erst genannter führt eine Faltungsoperation mit 256 Filtern mit einem 3x3 Faltungskern und einer Schrittweite von 1 durch. Anschließend folgt die *Batch Normalisation*, sowie die Aktivierung mithilfe der ReLU. Die Restblöcke verfügen ebenfalls über die drei genannten Komponenten – Faltung, Normalisierung und Aktivierung, sowie zusätzlich darauf anschließend erneut eine Faltung mit den gleichen Eigenschaften, gefolgt von der *Batch Normalisation* und der Endposition der Abkürzungsmöglichkeit der Eingabedaten. Abschließend findet sich erneut die ReLU. Die resultierenden Daten werden an den den sogenannten „Policy Head“, sowie den „Value Head“ weitergereicht. Diese beiden Komponenten sind für das Berechnen der *Policy* beziehungsweise der Spielbewertung zuständig. Der *Policy Head* setzt sich aus einer Faltung mit zwei Filtern mit einem 1x1 Faltungskern und einer Schrittweite von 1, einer Normalisierung, der ReLU und einem *fully connected layer* zusammen. Letzterer gibt einen Vektor zurück, der die Logit-Wahrscheinlichkeiten für alle realisierbaren Aktionen beinhaltet. Die ersten drei Schritte sind gleichermaßen im *Value Head* wiederzufinden. Allerdings folgt hier einem *fully connected layer* mit einer versteckten Schicht. Des Weiteren durchlaufen die Daten die ReLU und erneut eine vollständig verknüpfte Schicht, die in einem Skalar resultiert. Dieser wird abschließend durch die Aktivierungsfunktion tanh auf einen Punkt im Wertebereich zwischen [-1, 1] abgebildet [SSS+17].

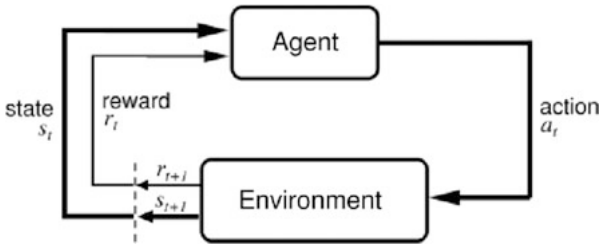

Abbildung 4: iterative Interaktion zwischen Agent und Environment, Abbildung entnommen aus [Sew19, S. 2]

2.3 Reinforcement Learning

Reinforcement Learning (RL), auch als bestärkendes Lernen bezeichnet, beschreibt die Interaktion eines Agenten mit einer konkreten Environment. Dabei stellt die Umgebung dem Agenten einen bestimmten Zustand bereit, anhand dessen er die bestmögliche Aktion auswählt, die im nächsten Schritt ausgeführt werden soll. Daraufhin wird der Zustand entsprechend modifiziert und eine Funktion berechnet, die dem Agenten eine Belohnung oder Bestrafung in Form eines positiven beziehungsweise negativen Rewards zurückliefert. Anhand iterativer Durchläufe von Lern- und Entscheidungsprozessen optimiert der Agent seine Aktionen. Dies ist in Abbildung 4 einzusehen. Dabei wählt der Agent zunächst Aktion a_t im Zustand s_t, was eine Änderung des Zustands der Umgebung zu s_{t+1} zur Folge hat. Außerdem berechnet die Environment den Reward r_t für die vom Agenten gewählte Aktion. Ziel in diesem Trainingsdurchlauf für den Agenten ist es, anhand der Belohnungsstrategie die Entscheidung bezüglich der besten nächsten Aktion zu stetig zu verbessern. Hierbei anzumerken ist, dass die Rewardfunktion sowohl den Zustand als auch die Aktion berücksichtigt, um einen Rückgabewert zu berechnen. Daraus folgt, dass die gleiche Aktion unterschiedlichen Ausgangszuständen in verschiedenen Belohnungswerten resultieren kann beziehungsweise sogar sollte [Sew19, S. 2].

Die Strategie zur Ermittlung der besten Aktion während des Lernprozesses wird als Policy π bezeichnet. Diese ist während eines gesamten Trainingsdurchlaufs gültig. Bezüglich der Notation gibt $\pi_{(s)}$ entsprechend die vielversprechendste Aktion im Zustand s an [Sew19, S. 16].

Ferner bringt das Aufstellen einer Rewardfunktion Schwierigkeiten mit sich, da sie versucht eine sinnvolle Vorhersage für die Bewertungen eines Zustandes zu ermitteln. Probleme können dabei beispielsweise darin liegen, dass Belohnungen womöglich erst im zukünftigen Verlauf zu Trage treten können oder dass sie zum gegenwärtigen Zeitpunkt schlichtweg noch ungewiss sind. Mögliche Lösungen hierfür sehen allerdings je nach Anwendungsdomäne verschieden aus [Sew19, S. 4-7].

Reinforcement Learning Modelle sind als „Markov Devision Process“ (MDP) anzusehen. Das hat zur Folge, dass aufgrund der zugrundeliegenden „Markov Property, sowie Markov Chain“ die Wahrscheinlichkeiten für die nächstmöglichen Aktionen lediglich vom derzeitigen Zustand

und nicht von weiteren vorangegangen abhhängen. Der MDP wendet diese Regeln auf den Entscheidungsprozess an, der im Fall von RL die Policy darstellt. Des Weiteren liefert der MDP die Wahrscheinlichkeiten für Zustandsübergange in der Form $P_a(s, s')$ an, wobei a für die mögliche Aktion, s für den aktuellen und s' für den resultierenden Zustand steht. Die Notation des entsprechenden Rewards lautet $R_a(s, s')$ [Sew19, S. 19f.].

Weitere Begriffe, die im Kontext von RL von Bedeutung sind, sind „Policy Evaluation“ und „Policy Iteration“. Erstere steht für die geschätzte Bewertung des Zustandes. Letztere beschreibt einen Iteration Prozess mit dem Ziel, dass die Policy konvergiert [Sew19, S. 27].

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 neuronalen 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

Als Vierer-Team bestand unsere Gruppe aus Simon Hofmeister, Nadiia Matsko, Monika Silber und Simon Wasserburger. Anzumerken ist, dass Simona Hofmeister und Simon Wasserburger bereits im Bachelorkurs Erfahrung mit dem Implementieren einer künstlichen Intelligenz für Reversi sammeln konnten. Aus diesen Gründen griffen wir für den Aufbau dieses Projektes auf das Grundgerüst des bereits im Bachelorkurs erstellen Programms von Simon Hofmeister zurück, der dieses entsprechend an die Anforderungen des vorliegenden Projektes anpasste.

Hinsichtlich der Verteilung der Aufgaben und des zugehörigen Aufwands ergab sich für die einzelnen Teammitglieder folgendes: Die weiteren Komponenten verteilten wir wie folgt:

Simon Hofmeister (150 h):

- Recherche
- Implementierung (Grundgerüst, NN)
- Dokumentation ()
- Organisation

Nadiia Matsko (150 h):

- Recherche
- Implementierung (NN)
- Dokumentation (AlphaGo, AlphaZero)
- Organisation

Monika Silber (150 h):

- Recherche
- Implementierung (MCTS)
- Dokumentation (Reversi, Theoretische Grundlagen, Implementierung MCTS, Organisation)
- Organisation

Simon Wasserburger (150 h):

- Recherche
- Implementierung (MCTS, Code-Refactoring Grundgerüst)

- Dokumentation (Grafiken MCTS)
- Organisation

4.2 Kommunikation

Die virtuelle Kommunikation fand hauptsächlich in unserer WhatsApp-Gruppe statt, wo wir einander auf Problemstellen aufmerksam machten, Lösungsansätze diskutierten, einander über Fortschritte informierten und Treffen ausmachten. Ebenfalls konnte durch Commits im Repository die Arbeit der Teamkollegen und entsprechend der Projektfortschritt verfolgt werden. Daneben verfügten wir über ein Trello-Board, in dem wir Aufgaben planten, dokumentierten und zuwiesen. In gemeinsamen Treffen besprachen und ergänzten wir dabei die offenen Punkte und kontrollierten den Projektablauf.

In den persönlichen Treffen, in denen meist alle Teammitglieder anwesend waren, konnten wir alle offenen Punkte konkretisieren und ausführlich diskutieren. Wir erarbeiteten gemeinsame Lösungsstrategien und halfen einander bei Verständnisproblemen. Abschließend trafen wir Absprachen zum weiteren Vorgehen, sowie zur Aufgabenverteilung.

4.3 Versionskontrolle

Als Versionskontrollsystem stand ein Gitlab-Repository der OTHR zur Verfügung. Zum Abrufen und Bearbeiten dessen wurde GitHub Desktop als GUI verwendet.

Hinsichtlich Implementierung der Komponenten wurden eigene Branches angelegt, die vom master-Branch geklont und nach Fertigstellung entsprechend dort wieder gemergt wurden. Dabei ergaben sich eigene Branches für den MCTS, das NN, sowie für den Prozess des Refactoring des ursprünglichen Codes. Die finale und alle Bestandteile umfassende Version befindet sich folglich im master-Branch.

4.4 OS, IDE und Programmiersprache

Als Entwicklungsumgebung wurde IntelliJ IDEA[2] von JetBrains genutzt. Da wir innerhalb der Gruppe bisher am meisten Erfahrung in Java hatten, wurde dies als Programmiersprache zur Implementierung des Projektes verwendet. Zur Erstellung des Programms kam maven[3] als Build-Management-Tool zum Einsatz.

Des Weiteren wurde zur Implementierung des NN die Library Eclipse deeplearning4j (DL4J) [DL4] zur Hilfe genommen. Hinsichtlich dieser Auswahl verschafften wir uns zunächst einen Überblick über vorhandene, gute Bibliotheken und recherchierten deren Vorteile und Anwendungsfälle, wobei die Entscheidung letztendlich auf die bereits genannte fiel.

[2] https://www.jetbrains.com/de-de/idea/
[3] https://maven.apache.org/

Entsprechend folgt nun die Darlegung der Eigenschaften von DL4J, wobei der Fokus auf der Ausarbeitung der für das vorliegende Projekte relevanten Aspekten liegt. Sämtliche Informationen sind, sofern nicht anders angegeben, der zugehörigen Website https://deeplearning4j.org/ entnommen.

Die Library hat ihren Ursprung bei Entwicklern der Firma Konduit [4] in San Francisco und wurde unter der Apache Software Foundation License 2.0 [5] veröffentlicht.

DL4J ist vollständig open source und in Java geschrieben, wobei für die zugrundeliegenden Berechnungen C, C++ und Cuda genutzt wurden. Des Weiteren unterstützt sie ND4J (N-Dimensional Arrays for Java) - eine Bibliothek für das wissenschaftliche Rechnen, die eine hohe Performanz gewährleistet [6]. Diese werden im vorliegenden Projekt anhand von `INDArrays` referenziert.

Außerdem werden zahlreiche, hilfreiche Features unterstützt, unter anderem Batch Normalisation, Residual Learning, CNNs. In Form eines `ComputationGraph` kann daher ein verhältnismäßig einfach ein komplexes neuronales Netz aufgebaut werden.

Weitere ausschlaggebende Vorteile liegen in der ausführlichen Dokumentation [7], sowie den zahlreichen Hilfsmitteln, wie Tutorials [8], Code-Beispiele [9] und Leitfäden zu sämtlichen Aspekten [10], die zudem alle schnell und unkompliziert über die eigene Website zu finden sind.

Ergänzend dazu sei auf mögliche Alternativen dazu verwiesen, die in lägen...

4.5 Testumgebungen

4.6 Projekt-Dokumentation

[4] https://konduit.ai/
[5] https://www.apache.org/licenses/LICENSE-2.0
[6] https://nd4j.org/
[7] https://deeplearning4j.org/api/latest/
[8] erreichbar über https://deeplearning4j.org/tutorials/setup
[9] https://github.com/eclipse/deeplearning4j-examples
[10] erreichbar über https://deeplearning4j.org/docs/latest/

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.

[DL4] Eclipse Deeplearning4j Development Team. Deeplearning4j: Open-source distributed deep learning for the JVM, Apache Software Foundation License 2.0. http://deeplearning4j.org.

[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.

[HZRS16] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *In Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[IS15] S. Ioffe and C. Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *In International Conference on Machine Learning*, page 448–456, 2015.

[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.

[SAH+19] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, Laurent Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy Lillicrap, and David Silver. Mastering atari, go, chess and shogi by planning with a learned model, 11 2019.

[Sew19] M. Sewak. *Deep Reinforcement Learning - Frontiers of Artificial Intelligence*. Springer, Singapore, 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