Westfälische Hochschule

University of Applied Sciences
Gelsenkirchen Bocholt Recklinghausen

Bachelorarbeit

Titel der Arbeit // Title of Thesis

Entwicklung eines Agenten für Vier gewinnt mit Monte-Carlo-Baumsuche und neuronalen Netzen

---

Akademischer Abschlussgrad: Grad, Fachrichtung (Abkürzung) // Degree

Bachelor of Science (B.Sc.)

Autorenname, Geburtsort // Name, Place of Birth

Stefan Göppert, Kassel

Studiengang // Course of Study

Medieninformatik

Fachbereich // Department

Informatik und Kommunikation

Erstprüferin/Erstprüfer // First Examiner

Prof. Dr. Wolfram Conen

Zweitprüferin/Zweitprüfer // Second Examiner

Prof. Dr. Marcel Luis

Abgabedatum // Date of Submission

Eidesstattliche Versicherung

Göppert, Stefan
Name, Vorname // Name, First Name

Ich versichere hiermit an Eides statt, dass ich die vorliegende Abschlussarbeit mit dem Titel

Entwicklung eines Agenten für Vier gewinnt mit Monte-Carlo-Baumsuche und neuronalen Netzen

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate kenntlich gemacht. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Essen, 23.11.2020
Ort, Datum, Unterschrift // Place, Date, Signature

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Im Oktober 2015 wurde der Europameister in Brettspiel Go, Fan Hui, erstmals von einem Computerprogramm geschlagen. Er unterlag dem von DeepMind entwickelten AlphaGo fünf zu null. DeepMind hat damit die Welt des Compter-Go revolutioniert, da Go aufgrund seiner hohen Komplexität sehr lange eine große Herausforderung für Computerprogramme dargestellt hat. Gerade einmal ein halbes Jahr später wurde auch der 18-fache Weltmeister, Lee Sedol, vier zu eins von AlphaGo besiegt.[1]

Der von AlphaGo verwendete Algorithmus, die Monte-Carlo-Baumsuche, wird bereits seit 2006 eingesetzt, um Computer-Go zu spielen. Er ist umfangreich erforscht und es existieren viele Varianten und Verbesserungen. Die neuartige Kombination mit neuronalen Netzen durch das DeepMind-Team hat allerdings dazu geführt, dass es erstmals gelang, die besten Spieler der Welt unter Turnierbedingungen zu besiegen.

Klassische Brettspiele wie Backgammon, Schach und Go sind im Bereich der künstlichen Intelligenz von großem Interesse. Sie verfügen über einfache, klar definierte Regeln, aus denen sich eine sehr große Komplexität entwickeln kann und dennoch können sie problemlos von Menschen verstanden werden. Durch diese leichte Verständlichkeit sind sie ein idealer Anfangspunkt für die Entwicklung von neuen Algorithmen und das Erlernen von neuen Techniken, bevor diese Algorithmen auf Probleme der realen Welt angewandt werden.

Aktuell steht Go im Fokus dieser Entwicklungen, aufgrund der Limitierungen einer Bachelorarbeit wird allerdings das einfachere Spiel VierGewinnt betrachtet.

Das Ziel dieser Arbeit ist es, einen Agent für VierGewinnt auf Basis der Monte-Carlo-Baumsuche (MCTS) zu entwickeln und dabei zu untersuchen, welche Auswirkungen verschiedene Verbesserungen und der Einsatz von neuronalen Netzen auf die Spielstärke des Agenten haben.

Die Grundlage dieser Arbeit bildet das Paper „A Survey of Monte Carlo Tree Search Methods“[2] von Browne u.a., in dem die Autoren den Forschungsstand zum Zeitpunkt der Veröffentlichung (2011) zusammenfassen und eine Vielzahl von Verbesserungen vorstellen. Sie geben außerdem einen Überblick darüber, auf welche Problemstellungen die Verbesserungen bereits angewandt wurden. Ausgehend von diesem Paper werden die Verbesserung durch Transpositionen, vorgestellt von Childs u.a.,[3] die All Moves as First (AMAF)-Heuristik - speziell die α-AMAF und Rapid Action Value Estimation (RAVE)-Technik auf

1. „AlphaGo: The story so far“, besucht am 12. August 2020, https://deepmind.com/research/case-studies/alphago-the-story-so-far.
2. Cameron B. Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, *IEEE Transactions on Computational Intelligence and AI in Games* 4, Nr. 1 (März 2012): 1–43.
3. Benjamin E. Childs, James H. Brodeur und Levente Kocsis, „Transpositions and Move Groups in Monte Carlo Tree Search“, in *2008 IEEE Symposium On Computational Intelligence and Games* (Dezember 2008), 389–395.

Basis der Arbeiten von Helmbold und Parker-Wood,[4] sowie von Gelly und Silver[5] - und die Verbesserung durch die Last Good Reply Policy, vorgestellt von Drake[6] und verbessert von Baier und Drake,[7] betrachtet.

1.1 Das Spiel Vier Gewinnt

VierGewinnt ist ein zwei Spieler Brettspiel, das auf einem senkrecht stehenden, hohlen, rechteckigen Spielbrett gespielt wird. Das klassische Spielbrett hat sieben Spalten und sechs Zeilen. Beide Spieler verfügen zu Beginn des Spiels über 21 Spielsteine einer Farbe, klassisch rot und gelb. Abwechselnd setzen beide Spieler einen Spielstein in eine freie Spalte und lassen den Spielstein so auf das unterste freie Feld fallen. Eine Spalte ist frei, solange sich darin weniger als sechs Spielsteine befinden.

Es ist nicht in den Regeln festgelegt, welche Farbe beginnt, die Abbildungen in dieser Arbeit gehen aber davon aus, dass der gelbe Spieler den ersten Stein setzt.

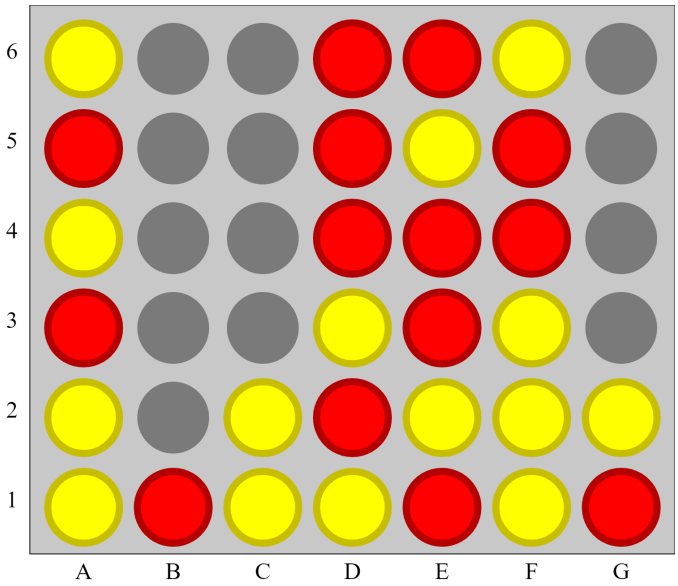


Abbildung 1: Eine mögliche Spielposition in VierGewinnt, Rot ist am Zug.

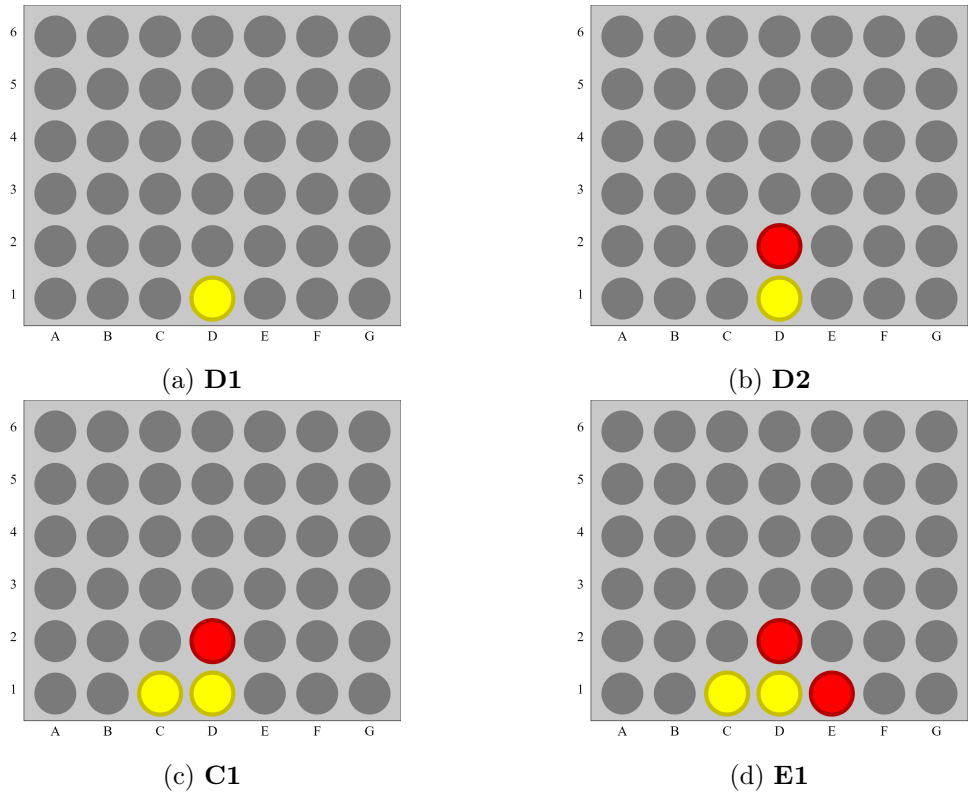
Ein Spieler hat das Spiel gewonnen, wenn es ihm gelingt, eine Viererreihe von Spielsteinen seiner eigenen Farbe zu bilden. Viererreihen können vertikal, horizontal oder diagonal gebildet werden.

4. David P Helmbold und Aleatha Parker-Wood, „All-Moves-As-First Heuristics in Monte-Carlo Go“:6.
5. Sylvain Gelly und David Silver, „Combining Online and Offline Knowledge in UCT“, in *Proceedings of the 24th International Conference on Machine Learning*, ICML '07 (Corvalis, Oregon, USA: Association for Computing Machinery, 20. Juni 2007), 273–280.
6. Peter Drake, „The Last-Good-Reply Policy for Monte-Carlo Go“, *ICGA Journal* 32, Nr. 4 (1. Dezember 2009): 221–227.
7. Hendrik Baier und Peter D. Drake, „The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go“, *IEEE Transactions on Computational Intelligence and AI in Games* 2, Nr. 4 (Dezember 2010): 303–309.

Gelingt es keinem Spieler eine Viererreihe zu bilden, bevor alle Felder mit Spielsteinen gefüllt sind, im klassischen Spiel nach 42 Spielsteinen, so endet das Spiel unentschieden.

Das Setzen eines Spielsteins wird in dieser Arbeit als **Zug** bezeichnet, die Spieler wechseln sich nach jedem Zug ab. In der englischen Literatur gibt es hierfür unterschiedliche Namen wie „ply“ oder „half-ply“, dabei ist in der Regel ein „ply“ die Kombination der Züge beider Spieler, und ein „half-ply“ ist der Zug eines einzelnen Spielers.

Ein Zug kann wie in Schach durch die gespielte Spalte und die Zeile, in der der Stein liegen bleibt, eindeutig definiert werden. In der Spielsituation in Abbildung 1 muss Rot den nächsten Stein setzen - mögliche Felder sind **B2**, **C3** und **G3**. Unabhängig davon, wo Rot seinen Stein setzt, kann Gelb mit dem nächsten Stein gewinnen, denn sowohl **B3** als auch **C4** und **G4** vervollständigen eine Viererreihe für Gelb.



(a) **D1** (b) **D2**

(c) **C1** (d) **E1**

Abbildung 2: Die ersten vier Spielzüge des Spiels aus Abb. 1. Diese Position kann als Abfolge der gemachten Züge **D1,D2;C1,E1** beschrieben werden.

Das Spielbrett ist achsensymmetrisch zur Spalte **D**, es kann also horizontal gespiegelt werden, ohne die Spielposition zu verändern. Diese Eigenschaft kann von Computerprogrammen ausgenutzt werden, um gespiegelte Zustände nicht doppelt zu untersuchen (siehe Kapitel 2.4.1).

VierGewinnt gehört zur Gruppe der kombinatorischen Spiele. Kombinatorische Spiele zeichnen sich dadurch aus, dass sie deterministisch sind, es keine verborgenen Informationen gibt, abwechselnd gezogen wird und das Spiel nach einer endlichen Anzahl an Zügen zu

Ende ist.[8] Als zufallsfreies Spiel mit perfekter Information ist VierGewinnt lösbar und wurde bereits 1988 von Victor Allis[9] und unabhängig davon im selben Jahr von James D. Allen schwach gelöst.[10]

Ein Spiel gilt als schwach oder stark gelöst, wenn ein realisierbarer Algorithmus existiert, mit dem für jede Startposition, bei beidseitig perfektem Spiel, eine optimale Spielweise bestimmt werden kann (schwach) oder wenn in jedem Spielzustand, auch solchen die nur durch fehlerhaftes Spiel erreicht werden, der optimale Zug bestimmt werden kann (stark).[11]

Victor Allis hat mithilfe seines Computerprogramms „VICTOR“ gezeigt, dass der erste Spieler bei beidseitig perfektem Spiel immer gewinnt, wenn er den ersten Stein in Spalte **D** setzt, das Spiel mindestens unentschieden endet, wenn er in die Spalten **C** oder **E** setzt, und verliert, wenn er in einer der anderen Spalten beginnt. Somit hat der erste Spieler einen entscheidenden Vorteil.

1.2 Definition eines Agenten

Wikipedia definiert einen Agenten wie folgt:

> Als Software-Agent (auch **Agent** oder Softbot) bezeichnet man ein Computerprogramm, das zu gewissem (wohl spezifiziertem) eigenständigem und eigendynamischem (**autonomem**) Verhalten fähig ist. Das bedeutet, dass abhängig von verschiedenen **Zuständen** (Status) ein bestimmter **Verarbeitungsvorgang** abläuft, ohne dass von außen ein weiteres Startsignal gegeben wird oder während des Vorgangs ein äußerer Steuerungseingriff erfolgt.[12]

Es handelt sich also um ein Computerprogramm, das abhängig von verschiedenen Zuständen ohne äußeres Einwirken (durch einen Benutzer) Entscheidungen trifft. Im Reinforcement Learning, einem Teilgebiet des maschinellen Lernens, lernt ein Agent durch Interaktion mit seiner Umgebung. Die Umgebung liefert den Zustand an den Agenten, welcher ausgehend von diesem Zustand eine Aktion wählt. Diese Aktion wiederum wird von der Umgebung verarbeitet, um einen neuen Folgezustand zu generieren.

8. *Kombinatorische Spieltheorie*, in *Wikipedia* (20. Dezember 2019), besucht am 14. August 2020, https://de.wikipedia.org/w/index.php?title=Kombinatorische_Spieltheorie&oldid=195080333.
9. Victor Allis, „A Knowledge-Based Approach of Connect-Four: The Game Is Solved: White Wins“ (Vrije Universiteit, 1. Dezember 1988).
10. James D. Allen, „Expert Play in Connect-Four“, besucht am 13. August 2020, http://tromp.github.io/c4.html.
11. *Gelöste Spiele*, in *Wikipedia* (19. März 2019), besucht am 14. August 2020, https://de.wikipedia.org/w/index.php?title=Gel%C3%B6ste_Spiele&oldid=186759431.
12. *Software-Agent*, in *Wikipedia* (6. Juli 2019), besucht am 14. August 2020, https://de.wikipedia.org/w/index.php?title=Software-Agent&oldid=190180915.

1.3 Kaggle

Die Inspiration für diese Arbeit kam durch den Anfang 2020 auf Kaggle gestarteten Wettbewerb „Connect X“.

> „Kaggle ist eine Online-Plattform für den Wissensaustausch und Wettbewerbe rund um die Datenanalyse, Machine Learning (ML), Data Mining und Big Data. Zielgruppe der Plattform sind Datenwissenschaftler sowie Unternehmen und Organisationen aus unterschiedlichsten Branchen. Die Mitglieder entwickeln Modelle, Daten nach bestimmten Vorgaben zu analysieren. Für die besten Lösungen sind in der Regel hohe Geldpreise ausgeschrieben.“[13]

Üblicherweise geht es in einem Wettbewerb auf Kaggle darum, eine gegebene Menge von Daten zu analysieren, um ein bestimmtes Problem zu lösen. „Connect X“[14] ist eine neue Art von Wettbewerb, bei dem sich die eingereichten Programme der Teilnehmer direkt miteinander messen, anstatt eine bestimmte Bewertungsmetrik zu erfüllen.

Kaggle stellt eine Python-Bibliothek mit einer Spielumgebung (Environment) zur Verfügung, mit der die Agenten entwickelt werden können und welche für die Evaluation im Wettbewerb eingesetzt wird.

Die Bewertung der eingereichten Agenten läuft kontinuierlich. Jeder Agent beginnt mit 600 Punkten. In regelmäßigen Abständen spielen zwei Agenten mit ähnlicher Punktzahl gegeneinander. Der Sieger dieses Spiels gewinnt Punkte abhängig davon, wie viel stärker der Gegner eingeschätzt wurde und der Verlierer verliert die selbe Anzahl Punkte. Dieses Bewertungssystem ist vergleichbar mit dem Elo-System, das für die Bewertung von Schach und Go Spielern benutzt wird.[15]

Der für diesen Wettbewerb eingereichte Programmcode muss in der Programmiersprache Python vorliegen und kann gängige Datascience Bibliotheken wie Numpy, Scipy und Deep-Learning-Bibliotheken[16] wie Pytorch, Tensorflow und Keras verwenden.

1.4 Aufbau der Arbeit

In den nachfolgenden Kapiteln wird zunächst in Kapitel 2 die Monte-Carlo-Baumsuche und die untersuchten Verbesserungen vorgestellt, welche danach in Kapitel 3 implementiert werden. Danach werden in Kapitel 4 die Grundlagen von künstlichen neuronalen Netzen

13. Stefan Luber, „Was ist Kaggle?“, 6. August 2020, besucht am 7. August 2020, https://www.bigdata-insider.de/was-ist-kaggle-a-951812/.
14. „Connect X“, besucht am 17. November 2020, https://kaggle.com/c/connectx/overview/evaluation.
15. Siehe *Elo-Zahl*, in *Wikipedia* (7. November 2020), besucht am 17. November 2020, https://de.wikipedia.org/w/index.php?title=Elo-Zahl&oldid=205293641.
16. Deep Learning bezeichnet das Teilgebiet des maschinellen Lernens, das sich mit tiefen neuronalen Netzen befasst. Siehe Kapitel 4 zu neuronalen Netzen.

erklärt, bevor eigene neuronale Netze mit Hilfe der AutoKeras-Bibliothek in Kapitel 5 entwickelt und trainiert werden. Am Ende dieses Kapitels werden die besten, trainierten Netzwerke mit der Monte-Carlo-Baumsuche kombiniert.

Die in den Kapiteln 3 und 5 entwickelten Agenten werden in Kapitel 6 miteinander verglichen und Kapitel 7 fasst die Ergebnisse danach zusammen.

2 Die Monte-Carlo-Baumsuche

Die Monte-Carlo-Baumsuche (MCTS) wurde 2006 parallel von Rémi Coulom[17] und Kocsis und Szepesvári[18] entwickelt. Coulom hat dabei den Begriff Monte Carlo Tree Search eingeführt.

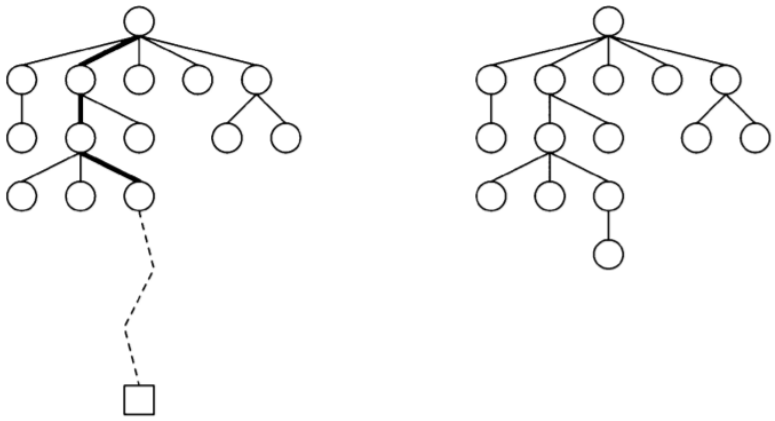


Abbildung 3: Vereinfachte Darstellung des MCTS-Algorithmus[19]

Die Monte-Carlo-Baumsuche erzeugt inkrementell einen asymmetrischen Baum, meistens einen Spielbaum (siehe Abb. 3). In jeder Iteration des Algorithmus wird eine **Tree Policy** benutzt, um den interessantesten Knoten im aktuellen Baum zu finden. Die **Tree Policy** muss dabei die Erkundung, also die Suche in weniger erforschten Teilen des Baumes, und die Ausnutzung der vielversprechendsten Knoten gegeneinander abwägen. Vom ausgewählten Knoten wird dann eine **Simulation** gestartet. Es werden so lange Aktionen, gesteuert durch die **Default Policy**, ausgeführt, bis ein terminaler Zustand erreicht ist. Das Ergebnis dieser Simulation wird benutzt, um den Suchbaum zu aktualisieren. Es wird ein neuer Kindknoten entsprechend der gewählten Aktion im Startknoten hinzugefügt und seine Statistik, sowie die seiner Vorfahren, aktualisiert.[20]

2.1 Hintergrund

Bevor die Monte-Carlo-Baumsuche erklärt werden kann, wird zunächst erklärt, was ein Spielbaum ist und was man unter Monte-Carlo-Methoden versteht.

17. Rémi Coulom, „Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search“, in *Computers and Games*, hrsg. H. Jaap van den Herik, Paolo Ciancarini und H. H. L. M. Donkers, bearb. David Hutchison u. a., Bd. 4630, Lecture Notes in Computer Science (Berlin, Heidelberg: Springer Berlin Heidelberg, 2007), 72–83, https://doi.org/10.1007/978-3-540-75538-8_7.

18. Levente Kocsis und Csaba Szepesvári, „Bandit Based Monte-Carlo Planning“, in *Machine Learning: ECML 2006*, hrsg. Johannes Fürnkranz, Tobias Scheffer und Myra Spiliopoulou, Lecture Notes in Computer Science (Berlin, Heidelberg: Springer, 2006), 282–293.

19. Baier und Drake, „The Power of Forgetting“

20. Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, S. 1 f.

2.1.1 Spielbäume

Spielbäume sind Datenstrukturen, die den Ablauf eines Spiels abbilden. Die Wurzel des Spielbaumes ist der initiale Zustand des Spiels, zum Beispiel das leere Spielfeld zu Beginn des Spiels.

Unter dem Zustand eines Spiels versteht man die vollständige Beschreibung der aktuellen Spielsituation. Dazu gehören beispielsweise die Anzahl und Position der Spielsteine oder Spielfiguren auf dem Spielfeld, die gesammelten Punkte der Spieler, die möglichen Aktionen, die in dieser Position ausgeführt werden können und welcher Spieler an der Reihe ist.

Ein Spiel kann formell mit den folgenden Elementen definiert werden:[21]

- s_0: Der **Ausgangszustand** des Spiels, zum Beispiel das leere Spielfeld
- PLAYER(s): Definiert, welcher Spieler im Zustand s am Zug ist
- ACTIONS(s): Definiert die Menge der legalen Spielzüge im Zustand s
- TRANSITION(s, a): Die **Übergangsfunktion** definiert den Folgezustand s', wenn Aktion a im Zustand s gewählt wurde
- TERMINAL-TEST(s): Ein Test, ob das Spiel im Zustand s vorüber ist oder nicht
- UTILITY(s, p): Eine Bewertungsfunktion, die einen numerischen Wert für einen terminalen Zustand s und einen Spieler p liefert. Dieser Wert ist typischerweise -1, 0 oder +1 für eine Niederlage, Unentschieden oder einen Sieg.

Mit dem Ausgangszustand, der ACTIONS Funktion und der TRANSITION Funktion kann ein Spielbaum definiert werden. Jeder Knoten in einem Spielbaum (siehe Abb. 4) repräsentiert einen Zustand des Spiels. Die Kanten von einem Knoten zu seinen Kindern repräsentieren die legalen Aktionen, die in diesem Zustand ausgeführt werden können, die jeweiligen Kindknoten sind die resultierenden Zustände, wenn die entsprechende Aktion im Zustand gewählt wurde. Gibt es in einem Zustand keine legalen Züge mehr, so handelt es sich um einen terminalen Zustand. Diese Zustände haben keine Kindknoten und können ausgewertet werden, um ein Spielergebnis zu erhalten. Sie bilden die Blätter des Baumes.

21. Stuart Russell und Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3 edition (Upper Saddle River: Pearson, 11. Dezember 2009), S. 162.

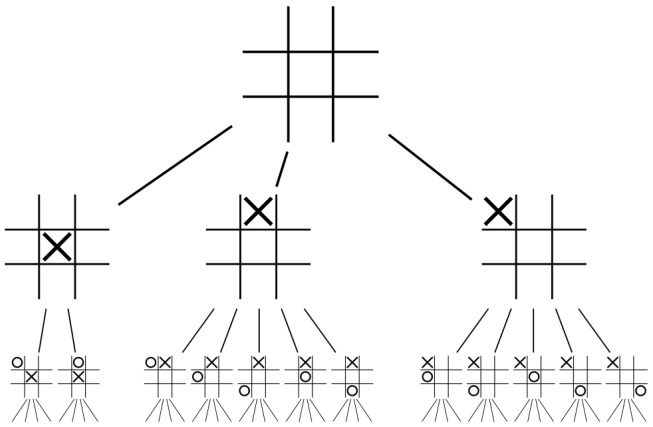


Abbildung 4: Ein Spielbaum mit den ersten zwei Zügen für Tic-Tac-Toe[22]

Abbildung 4 zeigt einen vereinfachten Spielbaum für das Spiel Tic-Tac-Toe. Spielzustände, die durch Rotation oder Spiegelung des Spielfeldes gebildet werden können, wurden ausgelassen. Durch die Symmetrie sind alle vier Zustände, in denen das erste Kreuz in einer Ecke respektive am Rand gesetzt wurde, identisch.

2.1.2 Minimax-Spielbäume

Minimax-Spielbäume sind spezielle Spielbäume für Zwei-Spieler-Nullsummenspiele. Der aktive Spieler im Ausgangszustand ist der Max-Spieler. Sein Ziel ist es, die maximal mögliche Punktzahl zu erreichen. Er wechselt sich mit seinem Gegenspieler, dem Min-Spieler, mit jedem Zug ab. Der Min-Spieler versucht die maximal erzielbare Punktzahl des Max-Spielers zu minimieren. In den meisten Spielbäumen haben nur die Blätter einen Wert und jeder Spieler versucht den Verlauf des Spiels so zu beeinflussen, dass sein Ziel erfüllt wird. Mit dem Wissen, dass beide Spieler optimal handeln und nicht von ihrer Strategie abweichen, kann der Wert eines Knotens rekursiv aus den Werten seiner Kindknoten bestimmt werden.

Wenn die Blätter des Baumes dem Max-Spieler gehören, so gehören die Elternknoten dieser Blätter dem Min-Spieler. Der Wert dieser Elternknoten ist also jeweils der minimale Wert der Kindknoten. Die Eltern dieser Knoten wiederum gehören dem Max-Spieler, der Wert der Eltern ist also das Maximum dieser Knoten. Diese Logik lässt sich bis zum Wurzelknoten fortsetzen um den Ausgang des Spiels vorherzusagen. Dieser Algorithmus wird **Minimax** genannt.

Ein Nachteil des Minimax-Algorithmus ist, dass der gesamte Spielbaum erforscht werden muss, also alle Blätter des Baumes besucht werden müssen. Kleine Spiele wie Tic-Tac-Toe können mit Minimax noch problemlos gelöst werden. VierGewinnt, Schach und Go haben einen zu großen Verzweigungsgrad, der verhindert, dass eine reine Minimax-Suche

22. Traced by User:Stannered en:User:Gdr original by, *First Two Ply of a Game Tree for Tic-Tac-Toe*, 1. April 2007, besucht am 14. August 2020, https://commons.wikimedia.org/w/index.php?curid=1877696

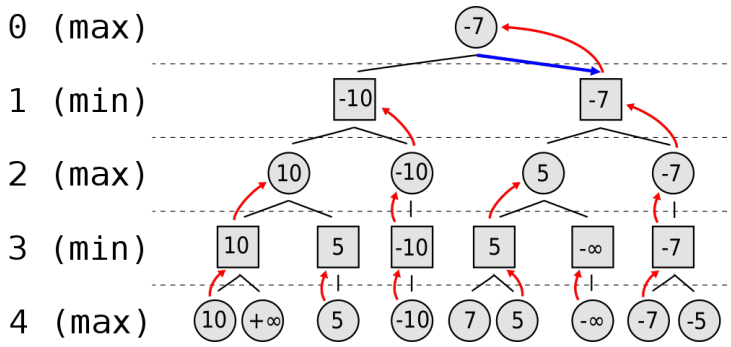


Abbildung 5: Ein Minimax-Spielbaum mit Kreisen für Max-Knoten und Quadraten für Min-Knoten. Die roten Pfeile sind die gewählten Aktionen in jedem Knoten und der blaue Pfeil ist die gewählte Aktion im Wurzelknoten. Die Zahlen in den Knoten stehen für den Wert der Knoten. [23]

in realistischer Zeit zu einem Ergebnis führt. Wenn eine Bewertungsfunktion für das Spiel bekannt ist, kann der Algorithmus auch nach einer maximalen Tiefe abbrechen und, anstatt weiter nach Blättern zu suchen, die Position direkt bewerten. Für nicht alle Spiele existiert eine gute Bewertungsfunktion.

2.1.3 Monte-Carlo-Methoden

Monte-Carlo-Methoden beschreiben Verfahren, bei denen ein analytisch schwer zu lösendes Problem, durch eine große Zahl gleichartiger Zufallsexperimente, numerisch approximiert wird.[24]

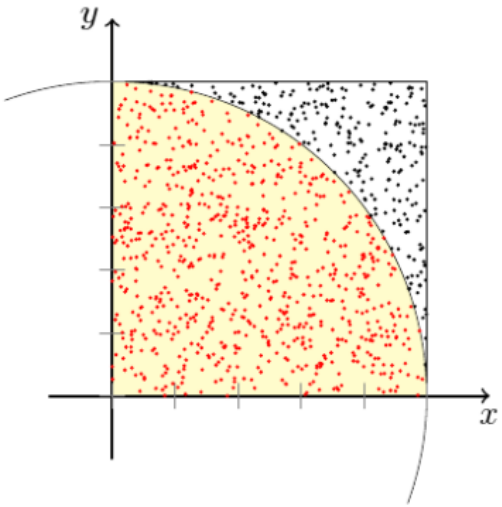


Abbildung 6: Numerische Berechnung der Kreiszahl Pi[25]

23. Nuno Nogueira, *Minimax Algorithm*, 4. Dezember 2006, besucht am 14. August 2020, https://commons.wikimedia.org/w/index.php?curid=2276653

24. *Monte-Carlo-Simulation*, in *Wikipedia* (23. September 2020), besucht am 13. Oktober 2020, https://de.wikipedia.org/w/index.php?title=Monte-Carlo-Simulation&oldid=203899528.

25. Springob at German Wikipedia, *Deutsch: Statistische Berechnung von Pi*, 24. November 2004, besucht am 13. Oktober 2020, https://commons.wikimedia.org/wiki/File:Pi_statistisch.png

Ein Beispiel für die Anwendung von Monte-Carlo-Methoden ist in Abb. 6 gegeben. Aus dem Verhältnis der Flächeninhalte des Quadrates und des Kreissektors kann Pi beliebig genau angenähert werden, indem zufällige Punkte im Quadrat gewählt werden und gezählt wird, wie viele Punkte davon im Kreissektor liegen.

Der Monte-Carlo-Teil der Monte-Carlo-Baumsuche beschreibt einen Schritt des Algorithmus, in dem wiederholt zufällige Spiele durchgeführt werden um somit die tatsächliche Gewinnchance aus einer Spielposition anzunähern.

2.2 Der MCTS-Algorithmus

Die Monte-Carlo-Baumsuche ist ein Best-First-Suchverfahren. Im Gegensatz zur Breitensuche, die den gesamten Baum gleichmäßig besucht und der Tiefensuche, die nach und nach zu jedem Blatt des Baumes vordringt, priorisiert MCTS die besten Teilbäume. Es wird Schritt für Schritt, beeinflusst durch die vorherige Erkundung des Baumes, ein partieller Spielbaum erzeugt. Dieser Spielbaum wird benutzt, um den Wert der Spielzüge abzuschätzen. Die Genauigkeit dieser Schätzung steigt mit jedem neuen besuchten Knoten.[26]

Der MCTS-Algorithmus vergrößert den Suchbaum so lange, bis ein Ressourcenlimit erreicht ist. Dies kann eine zeitliche Begrenzung, ein Speicherlimit oder eine feste Anzahl an Iterationen sein. Nach Erreichen dieses Limits wird die Suche beendet und der beste Kindknoten der Wurzel zurückgegeben.

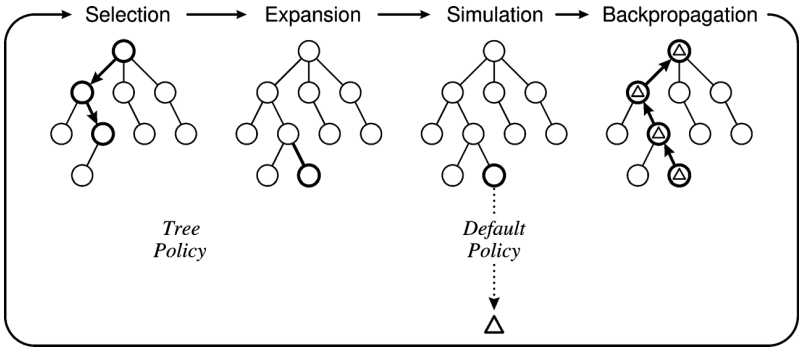


Abbildung 7: Eine Iteration des MCTS Algorithmus[27]

26. Siehe Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, S.. 5.
27. Browne u. a., S. 6

Eine Iteration der Baumsuche besteht aus vier Schritten (siehe Abb. 7):

1. **Selection:** Ausgehend von der Wurzel wird eine Auswahlregel (engl. policy), die in jedem Knoten das wichtigste zu erforschende Kind auswählt, rekursiv angewandt, bis ein terminaler oder ein unvollständiger Knoten erreicht ist. Ein Knoten ist terminal, wenn das Spiel in diesem Zustand vorüber ist und er ist unvollständig, wenn es noch potenzielle Kinder gibt, die von diesem Knoten aus noch nicht besucht wurden.

2. **Expansion:** Der Baum wird um einen (oder mehrere) Kindknoten, die durch einen unerforschten, legalen Zug erreichbar sind, erweitert. Ist der gewählte Knoten terminal, so wird kein neues Kind hinzugefügt.

3. **Simulation:** Von diesem neuen Knoten wird nun eine Simulation des Spiels, gesteuert durch die Default Policy, durchgeführt.

4. **Backpropagation:** Das Ergebnis der Simulation wird vom gewählten Knoten bis zur Wurzel entlang des durchsuchten Pfades propagiert. Alle dabei durchlaufenen Knoten aktualisieren ihre Statistiken mit dem Simulationsergebnis. Dieser Schritt wird im Folgenden auch als **Backup** oder **Update** bezeichnet.

Diese vier Schritte werden nach Browne u.a. in zwei Policies zusammengefasst:[28]

1. **Tree Policy:** Durchläuft den Baum und findet einen Knoten, der simuliert werden muss und erzeugt ihn, wenn nötig. Die Tree Policy kombiniert Selection und Expansion.

2. **Default Policy:** Steuert die Simulation des Spiels von einem nicht-terminalen Zustand bis zum Ende und erzeugt eine Bewertung des Endzustands. Im einfachsten Fall ist die Default Policy eine gleichverteilte Zufallsfunktion, die jeden Zug mit gleicher Wahrscheinlichkeit wählt.

Der Begriff Simulation ist in der Literatur nicht eindeutig. In manchen Fällen wird der gesamte Ablauf von der Wurzel bis zum Ende des zufälligen Spiels als Simulation bezeichnet. In anderen Fällen wird nicht von einer Simulation, sondern von einem Playout gesprochen. Diese Arbeit verwendet den Begriff Simulation für das zufallsgesteuerte Spiel nach dem Expandieren eines Knotens.

Der Algorithmus der Monte-Carlo-Baumsuche ist als Pseudocode in Algorithmus 1 zusammengefasst.

28. Siehe Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, S.. 6.

Algorithmus 1 Allgemeiner MCTS Algorithmus[29]

```
function MCTS(s_0)
    v_0 ← Knoten(s_0)
    while noch Zeit übrig do
        v_l ← TreePolicy(v_0)
        r ← SimuliereSpiel(s(v_l))
        Backup(v_l, r)
    return a(BestesKind(v_0))
```

Zunächst wird aus dem Ausgangszustand s_0 der Wurzelknoten v_0 erzeugt. So lange noch Rechenzeit verfügbar ist, wird zuerst die TreePolicy benutzt, um beginnend in der Wurzel ein unerforschtes Blatt v_l zu finden. Vom Spielzustand in diesem Blatt $s(v_l)$ wird dann ein Spiel simuliert und das Ergebnis r wird vom Blatt v_l den Baum bis zur Wurzel hinauf propagiert. Ist die Zeit abgelaufen, gibt der Algorithmus die Aktion, die zum besten Kindknoten der Wurzel führt $a(\text{BestesKind}(v_0))$, zurück. Die genaue Definition des „besten Kindes“ hängt von der Implementation ab, es ist aber üblich, dass das Kind mit der besten Bewertung oder den meisten Besuchen ausgewählt wird.

Für die Implementation der TreePolicy und insbesondere die Regel, wie in einem Knoten das beste Kind ermittel wird, gibt es verschiedene Ansätze. Die meistgenutzte Tree Policy ist **Upper Confidence Bounds for trees** (UCT) entwickelt von Kocsis und Szepesvári.[30]

2.3 Upper Confidence Bounds for Trees (UCT)

UCT gilt heute als beliebtester MCTS-Algorithmus. Kocsis und Szepesvári betrachten die Auswahl eines Kindknotens in dieser Variante als ein sogenanntes Banditenproblem.[31]

Banditenprobleme sind eine Klasse von Problemen im Reinforcement Learning, in denen der Agent zu jedem Zeitschritt t aus K Optionen optimal wählen muss, um eine kumulative Belohnung zu maximieren, indem die Aktion mit dem höchsten Wert so häufig wie möglich gewählt wird. Die Verteilung der Belohnungen jeder Aktion ist unbekannt aber statisch und die Belohnungen aus sukzessiven Aktionen sind voneinander unabhängig. Der Agent muss lernen, nur durch gesammelte Erfahrung seine erhaltene Belohnung auf lange Sicht zu maximieren. Verhält er sich gierig und wählt nur die Aktion mit der höchsten durchschnittlichen Belohnung, so werden alle Aktionen ignoriert, die im ersten Versuch keine Belohnung geliefert haben. Verbringt er dagegen zu viel Zeit damit, andere Aktionen als die derzeit beste zu erforschen, wählt er häufig suboptimale Aktionen.

29. Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, S. 6
30. Kocsis und Szepesvári, „Bandit Based Monte-Carlo Planning“.
31. Siehe Richard S. Sutton und Andrew G. Barto, *Reinforcement Learning: An Introduction*, Second edition, Adaptive Computation and Machine Learning Series (Cambridge, Massachusetts: The MIT Press, 2018), S. 25 ff.

Dieses Problem, auch bekannt als **exploration-exploitation-Dilemma**, ist ein Grundproblem vieler Algorithmen des Reinforcement Learnings. Die beste Strategie für das Banditenproblem ist die Upper Confidence Bound (UCB)-Regel **UCB1** von Auer u.a.[32]

Der Agent wählt die Aktion $j, 1 \leq j \leq K$, die die Gleichung

$$\bar{X}_j + \sqrt{\frac{2 \ln n}{n_j}} \tag{1}$$

maximiert. Wobei $\bar{X}_j$ die durchschnittliche Belohnung ist, die bisher durch das Spielen von Aktion j erhalten wurde, n_j ist die Anzahl der Spielzüge in denen j gewählt wurde und n ist die Anzahl der insgesamt gespielten Spielzüge. Diese Gleichung besteht aus einem exploitation-Teil $\bar{X}_j$ und einem exploration-Teil $\sqrt{(2 \ln n)/n_j}$. Der exploration-Teil repräsentiert die Unsicherheit in der Bewertung der Aktion j. Wenn die Aktion noch nicht oft gewählt wurde, n_j also im Vergleich zu n gering ist, so wird dieser Teil größer. Die Bewertung $\bar{X}_j$ ist also noch sehr unsicher. Wurde dagegen j sehr häufig gewählt, so wird der exploration-Teil kleiner und drückt eine hohe Sicherheit in der Schätzung des Wertes $\bar{X}_j$ aus.

Im UCT-Algorithmus wird die Auswahl des Kindknotens als ein solches Banditenproblem interpretiert. Dafür hat jeder Knoten v eine Statistik $Q(v)$, die durchschnittlich erhaltene Belohnung durch diesen Knoten, und $N(v)$, die Anzahl der Besuche des Knotens. Das gewählte Kind v' ist dann jenes, welches die Formel

$$Q(v') + 2C_p \sqrt{\frac{2 \ln N(v)}{N(v')}} \tag{2}$$

maximiert. C_p ist eine Konstante $C_p > 0$ für die Kocsis und Szepesvári gezeigt haben, dass bei Belohnungen im Bereich $[0, 1]$ der Wert $C_p = 1/\sqrt{2}$ optimal ist.[33] Für andere Belohnungen und für Veränderungen der Tree Policy muss der Parameter C_p individuell bestimmt werden.

Im Folgenden wird eine leichte Variation der UCT-Regel verwendet. Da mit VierGewinnt ein zwei Spieler Spiel mit Minimax-Spielbaum untersucht wird, liegen die erwarteten Belohnungen im Bereich $[-1, 1]$. Somit muss C_p für jede Implementierung experimentell bestimmt werden, weshalb $2\sqrt{2}$ aus Gleichung 2 zur Vereinfachung entfernt wurde.

Algorithmus 2 zeigt den vollen Pseudocode des UCT-Algorithmus.

Zusätzlich zu $Q(v)$ und $N(v)$ enthalten die Knoten noch Informationen über ihren Spiel-

32. Peter Auer, Nicolò Cesa-Bianchi und Paul Fischer, „Finite-Time Analysis of the Multiarmed Bandit Problem“, *Machine Learning* 47, Nr. 2 (1. Mai 2002): S. 237.
33. Siehe Kocsis und Szepesvári, „Bandit Based Monte-Carlo Planning“, S. 286 ff.

zustand $s(v)$ und die Aktion, die in den Knoten geführt hat $a(v)$, sowie einen Verweis auf den Elternknoten. TRANSITION(s, a) ist die Übergangsfunktion, die den Folgezustand s' erzeugt, der daraus resultiert dass in Zustand s die Aktion $a \in A(s)$ ausgewählt wurde. $A(s)$ ist die Menge aller legalen Spielzüge im Zustand s, analog dazu ist $A(v)$ die Menge aller legalen Spielzüge in Knoten v.

Algorithmus 2 Upper Confidence Bounds for Trees[34]

```
1:  function UCT(s_0)
2:      v_0 ← Knoten(s_0)
3:      while noch Zeit übrig do
4:          v_l ← TreePolicy(v_0)
5:          r ← SimuliereSpiel(s(v_l))
6:          Backup(v_l, r)
7:      return a(BestesKind(v_0, 0))
8:  function TreePolicy(v)
9:      while v ist nicht terminal do
10:         if v ist nicht vollständig expandiert then
11:             return Expandiere(v)
12:         else
13:             v ← BestesKind(v)
14:     return v
15: function Expandiere(v)
16:     wähle a ∈ noch nicht gewählte Aktionen aus A(s(v))
17:     füge ein neues Kind v' zu v hinzu
18:     mit s(v') = Transition(s(v), a)
19:     und a(v') = a
20:     return v'
21: function BestesKind(v, C_p)
22:     return argmax_{v' ∈ Kinder von v} Q(v') + C_p sqrt(ln N(v) / N(v'))
23: function SimuliereSpiel(s)
24:     while s ist nicht terminal do
25:         wähle a ∈ A(s) zufällig
26:         s ← Transition(s, a)
27:     return Belohnung für s
```

Die Backup-Regel verändert sich abhängig davon, ob das Spiel ein Ein-Spieler-Spiel oder ein Zwei-Spieler-Nullsummenspiel (und damit Minimax) ist. Da in einem Minimax-Baum ein Sieg von Spieler eins schlecht für Spieler zwei (und umgekehrt) ist, wird die Belohnung bei der Propagation durch den Baum mit jedem Knoten invertiert.

34. Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, S. 7 f.

Algorithmus 3 Backup-Regeln für Ein-Spieler und Zwei-Spieler Minimax

```
function BACKUP(v, r)                                   ▷ Ein-Spieler Backup
    while v ist nicht NULL do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + r
        v ← Elternknoten von v

function BACKUP(v, r)                                   ▷ Minimax Backup
    while v ist nicht NULL do
        N(v) ← N(v) + 1
        Q(v) ← Q(v) + r
        r ← −r
        v ← Elternknoten von v
```

Der UCT-Algorithmus, wie er von Kocsis und Szepesvári vorgestellt wurde, wird in Kapitel 3 implementiert und als Startpunkt für alle weiteren Verbesserungen verwendet.

2.4 Verbesserungen

Die Baumsuche kann um viele verschiedene Verbesserungen erweitert werden, durch die der Algorithmus für bestimmte Aufgaben besser geeignet ist. Diese Verbesserungen machen sich dabei gewisse Eigenschaften der Problemstellung zu Nutze. Generell wird zwischen Verbesserungen der Tree Policy und anderen Verbesserungen unterschieden. In diesem Kapitel werden zwei Verbesserungen der Tree Policy und eine Verbesserung der Simulation vorgestellt.

2.4.1 Transpositionen

Unter Transpositionen versteht man identische Spielzustände, die über unterschiedliche Zugkombinationen erreicht werden. Ein solcher Spielzustand wird zum Beispiel über die Zugfolge **D1,E1;C1,D2** erreicht (siehe Abb. 8).

Eine andere Zugfolge, die zum selben Spielzustand führt, ist **D1,D2;C1,E1**. Durch diese unterschiedlichen Pfade werden die beiden identischen Zustände normalerweise als separate Knoten mit separaten Statistiken gespeichert. Da es sich um den selben Zustand handelt, können diese Knoten kombiniert werden, denn der Weg, wie ein Zustand erreicht wird, hat keinen Einfluss darauf, welches die beste Aktion in diesem Zustand ist.

Zusätzlich können Symmetrie-Eigenschaften des Spiels ausgenutzt werden. VIERGEWINNT ist Achsensymmetrisch zur Spalte **D**, dadurch können beispielsweise die sieben möglichen Startpositionen auf vier einzigartige Zustände reduziert werden. Nach zwei Spielzügen existieren somit statt $7 \times 7 = 49$ nur noch $3 \times 7 + 4 = 25$ mögliche Spielzustände:

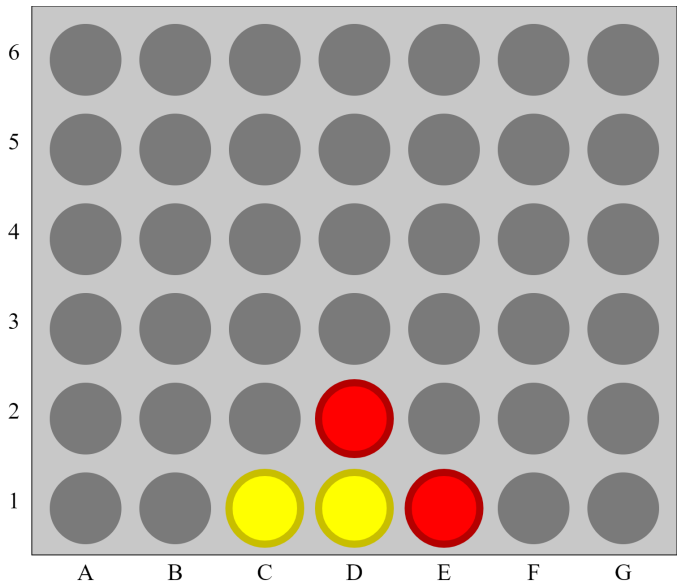


Abbildung 8: Diese Spielposition kann auf mehreren Wegen erreicht werden. Die Züge **D1,E1;C1,D2** sowie **D1,D2;C1,E1** und **C1,E1;D1,D2** führen zum gleichen Zustand.

Wenn Spalte **A**, **B** oder **C** von Gelb gespielt wurde, ist jede der 7 Antworten von Rot eine einzigartige Position. Hat Gelb Spalte **D** gespielt, so lassen sich die Antworten **A1**, **B1** und **C1** auf **E1**, **F1** und **G1** spiegeln. Zusammen mit **D2** gibt es in diesem Fall vier einzigartige Zustände.

Childs u.a. schlagen in ihrem Paper „Transpositions and Move Groups in Monte Carlo Tree Search“[35] drei Anpassungen des UCT-Algorithmus vor, um mit Transpositionen umzugehen.

Normale UCT-Algorithmen, die Transpositionen nicht berücksichtigen, werden im Paper als **UCT0** bezeichnet.

Wenn Transpositionen identifiziert und identische Knoten zusammengelegt werden, verwandelt sich der Baum in einen gerichteten azyklischen Graph. In diesem Graph kann jeder Knoten über einen oder mehrere Wege erreicht werden. Dadurch wird es wichtig, nicht nur die Bewertung der Knoten $Q(v)$ und $N(v)$ zu speichern, sondern auch die Statistiken wenn von einem Knoten v die Kindknoten $v_a, a \in A(v)$ besucht wurden. Diese Statistiken werden im Folgenden als $Q(v,a)$ und $N(v,a)$ ausgedrückt. Kann ein Knoten v' nur von einem Elternknoten v mit der Aktion a_i erreicht werden, es ist also keine Transposition, so sind die Statistiken $Q(v')$ und $Q(v,a_i)$ identisch.

Wenn Transpositionen erkannt werden, so können sich im einfachsten Fall die Knoten, unabhängig davon, wo sie sich im Baum befinden, ihre Statistiken teilen. Dies hat vor allem den Vorteil, dass der Baum kleiner wird. Die von Childs u.a. vorgeschlagene einfache

35. Childs, Brodeur und Kocsis, „Transpositions and Move Groups in Monte Carlo Tree Search“, S. 390 f.

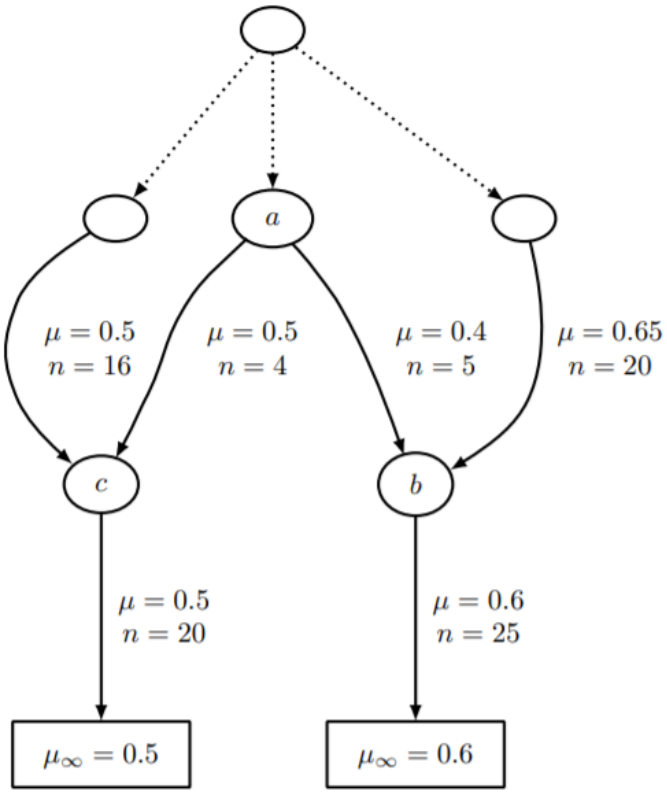


Abbildung 9: Ein Beispielgraph mit Transpositionen. μ und n entsprechend den Bewertungen $Q(v, a)$ und $N(v, a)$ der jeweiligen Kanten. Von außen betrachtet ist klar erkennbar, dass b der bessere Knoten ist, lokal ist dies aber in a nicht ersichtlich, hier scheint c mit $\mu = 0.5$ die bessere Wahl zu sein. Indem zusätzliche Informationen aus den anderen Wegen zu b und c wiederverwendet werden, kann auch in a der optimale Knoten gewählt werden.[36]

Auswahlregel **UCT1** ist:

$$UCT1 = \underset{a \in A(v)}{\operatorname{argmax}} Q(v, a) + C_p \sqrt{\frac{\ln N(v)}{N(v, a)}} \tag{3}$$

Diese unterscheidet zwischen den verschiedenen Wegen, die zum selben Knoten geführt haben können, kumuliert aber die Statistiken von Knoten mit identischen Zuständen.

Die zweite vorgeschlagene Auswahlregel **UCT2** macht Gebrauch von mehr Informationen aus den Transpositionen, indem statt der Bewertung der Aktion $Q(v, a)$ die Bewertung des Folgezustands $Q(v')$, wie in der originalen UCT Implementierung, verwendet wird (siehe Gleichung 4). v' ist der Knoten, der erreicht wird, wenn in Knoten v die Aktion $a, a \in A(v)$ gewählt wird. Für die Berechnung des Erkundungsfaktors wird weiterhin $N(v, a)$ anstatt

36. Tristan Cazenave, Jean Méhat und Abdallah Saffidine, „UCD : Upper Confidence Bound for Rooted Directed Acyclic Graphs“, *Knowledge-Based Systems* 34 (2012): S. 6

$N(v')$ im Nenner benutzt, da laut Childs die Auswahl sonst nicht mehr auf den korrekten Knoten konvergiert, wenn der Knoten v' sehr häufig über einen anderen Pfad besucht wurde.[37]

$$UCT2 = \underset{a \in A(v)}{\operatorname{argmax}} Q(v') + C_p \sqrt{\frac{\ln N(v)}{N(v,a)}} \tag{4}$$

Die letzte vorgeschlagene Variante **UCT3** berechnet den Wert der Kinder $Q(v')$ rekursiv als gewichteter Durchschnitt aller Kindeskinder. Dadurch, dass rekursiv alle Kinder eines Knotens betrachtet werden müssen, ist der Rechenaufwand in der Methode BESTESKIND sehr hoch. Um ihn zu reduzieren, können die Durchschnittswerte in den Knoten zwischengespeichert werden und müssen nur neu berechnet werden, wenn sich ein Kind verändert. Damit kann der Rechenaufwand aus dem teuren Selection-Schritt in den Backpropagation-Schritt verlagert werden. Die UCT3-Regel ist definiert als:

$$\begin{aligned} UCT3 &= \underset{a \in A(v)}{\operatorname{argmax}} Q^{UCT3}(v') + C_p \sqrt{\frac{\ln N(v)}{N(v,a)}} \\ Q^{UCT3}(v) &= \sum_{a \in A(v)} \frac{N(v,a)}{N(v)} Q^{UCT3}(v') \end{aligned} \tag{5}$$

In den meisten Implementierungen dieser Algorithmen werden die Statistiken nur in den Knoten gespeichert. Cazenave, Mehat und Saffidine[38] argumentieren dafür, die Statistiken nicht in den Knoten, sondern in den Kanten zu speichern. Dies kann allerdings die Implementierung verkomplizieren und wird deshalb bei der Anwendung von Transpositionen eher selten umgesetzt.

2.4.2 All Moves as First

Wenn die Simulation eines Spiels mit einem bestimmten Zug des ersten Spielers beginnt und zu einem Sieg führt, so deutet das darauf hin, dass dieser Zug gut war. All Moves as First (AMAF) weitet diese Erkenntnis auf alle in der Simulation gemachten Züge aus, indem alle am Sieg beteiligten Züge positiv bewertet werden. Umgekehrt werden alle Züge des Spielers, der verloren hat, negativ bewertet, egal wann sie in der Simulation gespielt wurden. Der Algorithmus aktualisiert dafür im Update-Schritt nicht nur die Statistik der durch die Tree Policy besuchten Knoten, sondern auch alle Nachbarknoten, die durch einen später gemachten Spielzug hätten erreicht werden können.[39]

37. Childs, Brodeur und Kocsis, „Transpositions and Move Groups in Monte Carlo Tree Search“, S. 390.
38. Cazenave, Méhat und Saffidine, „UCD“.
39. Helmbold und Parker-Wood, „All-Moves-As-First Heuristics in Monte-Carlo Go“.

Verschiedene AMAF-Varianten unterscheiden sich in der Art und Weise, wie diese zusätzlichen Informationen gespeichert und verarbeitet werden. So können beispielsweise die Statistiken $Q(v)$ und $N(v)$ direkt modifiziert werden oder jeder Knoten kann zusätzliche Statistiken $Q_{\mathrm{AMAF}}(v)$ und $N_{\mathrm{AMAF}}(v)$ speichern, welche dann in der Tree Policy kombiniert werden. Der AMAF-Wert ist nicht sehr genau, kann aber dabei helfen, die Suche in die richtige Richtung zu lenken, solange der Baum noch nicht gut genug erforscht ist, da auch Knoten, die nicht in der Iteration besucht wurden, aktualisiert werden. Man spricht vom **Bootstrapping** des Baumes.

In dieser Arbeit werden die zwei Varianten α-AMAF und Rapid Action Value Estimation (RAVE) untersucht.

α-AMAF: α-AMAF ist eine sehr einfache Variante. Der Wert eines Knotens v ist die Linearkombination aus dem UCT-Wert und dem AMAF-Wert

$$\alpha Q_{\mathrm{AMAF}}(v) + (1 - \alpha)Q(v) \tag{6}$$

wobei $0 < \alpha < 1$ eine Konstante ist. Ist $\alpha = 0$ so entspricht diese Variante dem normalen UCT-Algorithmus, ist $\alpha = 1$ so wird nur der AMAF-Wert benutzt.

RAVE: Die RAVE-Variante unterscheidet sich von α-AMAF dadurch, dass die Konstante α durch einen variablen Wert β ersetzt wird, der in jedem Knoten unterschiedlich ist und sich im Laufe der Suche verändert. Zu Beginn der Suche, wenn der Baum noch wenige Informationen enthält, ist $\beta = 1$. Je häufiger ein Knoten besucht wurde und damit je genauer die UCT-Statistik in diesem Knoten ist, umso geringer ist β.

Es gibt verschiedene Ansätze den Parameter β zu berechnen, in dieser Arbeit wird die Formel

$$\beta = \sqrt{\frac{k}{3N(s) + k}} \tag{7}$$

von Gelly und Silver[40] benutzt. k ist ein konfigurierbarer Parameter, der bestimmt, ab wie vielen Besuchen $N(v)$ der AMAF-Wert und die UCT-Statistik das gleiche Gewicht haben sollen, $\beta = 0.5$.

40. Siehe Sylvain Gelly und David Silver, „Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go“, *Artificial Intelligence* 175, Nr. 11 (1. Juli 2011): S. 1866.

2.4.3 Last-Good-Reply

Die vorigen Verbesserungen haben versucht, mehr Informationen aus der Baumstruktur zu ziehen oder zusätzliche Statistiken in den Knoten zu sammeln. Die Last-Good-Reply-Policy (LGR), vorgestellt von Drake[41] und später verbessert von Baier und Drake,[42] hat das Ziel, die Genauigkeit der Simulation zu verbessern.

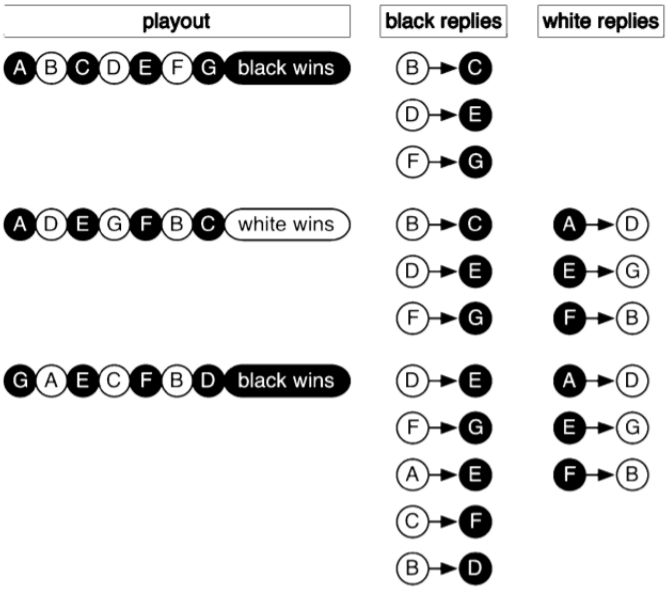


Abbildung 10: Aktualisierung der LGR-Tabelle über drei Simulationen[43]

Drake schlägt vor, jeden Zug in der Simulation als Antwort (engl. reply) auf den Zug des Gegenspielers zu betrachten. Haben die gemachten Züge zu einem Sieg in der Simulation geführt, so waren die Antworten gut und werden bei nachfolgenden Simulationen wiederverwendet. Haben sie zu einer Niederlage geführt, so werden diese Antworten nicht gespeichert. In späteren Simulationen, wenn der Gegenspieler einen Zug wählt, zu dem eine gute Antwort bekannt ist, wird die Antwort gespielt, sofern sie einen legalen Spielzug in der aktuellen Situation darstellt. Ist keine Antwort bekannt, wird die normale Default Policy benutzt und in der Regel ein zufälliger Zug gespielt.

LGR speichert zu jedem Spielzug, eindeutig definiert durch Zeile, Spalte und Farbe des Spielers, die Antwort des Gegenspielers. Zu jedem Zug wird nur eine Antwort gespeichert, alte Antworten werden dabei überschrieben.

Abbildung 10 zeigt die Aktualisierung der LGR-Tabelle über drei Simulationen (Baier und Drake bezeichnen Simulationen hier als Playouts) in einem hypothetischen Spiel. Schwarz gewinnt in der ersten Simulation, also werden alle Spielzüge von Weiß, auf die Schwarz geantwortet hat, mit ihrer Antwort gespeichert. In der zweiten Simulation benutzt Schwarz

41. Drake, „The Last-Good-Reply Policy for Monte-Carlo Go“.
42. Baier und Drake, „The Power of Forgetting“.
43. Baier und Drake, S. 305

die bekannten Antworten (D)→● E und (B)→● C, verliert aber. Weiß speichert seine guten Antworten. In der dritten Simulation spielen beide Spieler ihre bekannten Antworten (A)→● E sowie ● F→(B), zu anderen Zügen sind zwar gute Antworten bekannt, sie sind aber nicht legal und können deshalb nicht gespielt werden. Da Schwarz gewinnt, werden die Antworten wieder gespeichert, dieses Mal wird die Antwort auf (B) überschrieben.

Baier erweitert die Last-Good-Reply-Policy um ein Vergessen der Antworten, wenn sie zu einer Niederlage geführt haben.[44] Diese Ergänzung hat einen positiven Einfluss auf die Spielstärke des Algorithmus im Spiel Go und kann trivial implementiert werden.

44. Siehe Baier und Drake, „The Power of Forgetting“.

3 Implementierung der Monte-Carlo-Baumsuche

Dieses Kapitel geht auf die Implementierung der Baumsuche und ihrer Verbesserungen ein. Nach einem Überblick über die Struktur des Projektes wird zunächst die Implementierung des Spiels VierGewinnt in Kürze vorgestellt, bevor dann die Baumsuche und die, für die Verbesserungen jeweils gemachten, Veränderungen dargestellt werden.

3.1 Überblick

Die Grundlage der Agenten bildet die abstrakte Klasse `players.base_players.Player` (siehe Listing 1). Diese Klasse definiert die Methode `get_move`, die von allen Implementierungen eines Spielers überschrieben werden muss. Diese Methode hat zwei Parameter `observation` und `configuration` und entspricht damit der vom Kaggle-Wettbewerb erwarteten Schnittstelle für einen Agent.

```
1 class Player(ABC):
2
3     @abstractmethod
4     def get_move(self, observation, configuration) -> int:
5         pass
```

Listing 1: Die Basisklasse aller Agenten

Die `observation` ist eine Repräsentation des Spielzustandes wie er durch die Kaggle-Spielumgebung erzeugt wird. Sie enthält das Spielfeld und eine Markierung, welcher Spieler am Zug ist. Das Spielfeld wird dabei als Liste dargestellt. Das erste Element der Liste ist die obere linke Zelle des Spielbretts und das letzte Element ist die untere rechte Zelle. Formatiert sieht die `observation` wie folgt aus:

```
{
    board: [
        0, 0, 0, 0, 0, 0, 0,
        0, 0, 2, 0, 0, 0, 2,
        0, 0, 2, 0, 0, 0, 1,
        0, 0, 1, 0, 0, 0, 2,
        1, 0, 2, 0, 0, 0, 1,
        1, 1, 2, 1, 0, 1, 2
        ],
    mark: 2
}
```

Die `configuration` enthält die Spielregeln. Sie definiert die Form des Spielsfelds (`rows`, `columns`) sowie die Anzahl der Steine, die in einer Reihe sein müssen, um zu gewinnen

(`inarow`). Außerdem enthält die `configuration` das maximale Zeitlimit in dem der Agent eine Entscheidung treffen muss (`actTimeout`).

Um Logik, die von allen MCTS-Spieler gemeinsam genutzt wird, auszulagern, basiert der normale MCTS-Spieler auf der Klasse `players.base_players.TreePlayer` und alle Verbesserungen erben von diesem MCTS-Spieler. Der `TreePlayer` kann den Spielbaum zwischen Zügen erhalten oder verwerfen und die maximale Ausführungszeit auf ein Zeitlimit oder eine Anzahl von Iterationen beschränken. Jeder MCTS-Spieler muss mindestens mit `init_root_node` definieren, wie der Wurzelknoten erstellt wird und in der Methode `perform_search` den Suchalgorithmus definieren.

```
class TreePlayer(Player):
    def has_resources(self) -> bool:
        if self.time_limit > 0:
            return time.time() - self.start_time < self.time_limit
        else:
            self.steps_taken += 1
            return self.steps_taken <= self.max_steps

    @staticmethod
    def determine_opponent_move(new_board, old_board, columns = 7):
        # gekürzt ...

    def _restore_root(self, observation, configuration):
        # gekürzt ...

    def _store_root(self, new_root):
        # gekürzt ...

    @abstractmethod
    def init_root_node(self, game_state):
        pass

    @abstractmethod
    def perform_search(self, root):
        pass
```

Listing 2: Eine Übersicht über den `TreePlayer`

Die Methode `get_move` des `TreePlayer` ist in Listing 3 gezeigt. Der `TreePlayer` versucht zu Beginn eines Zuges den aktuellen Zustand im gespeicherten Spielbaum zu finden. Ist kein Baum vorhanden oder kann der Zustand nicht gefunden werden, so wird eine neue Wurzel erstellt. Dieser (neue) Baum wird dann in die Methode `perform_search` gegeben, welche die eigentliche Baumsuche ausführt. Wenn die Suche fertig ist, wird der Baum für den nächsten Zug gespeichert und das Ergebnis der Suche zurückgegeben.

```
    def get_move(self, observation, conf) -> int:
        root = self._restore_root(observation, conf)
        if root is None:
            root_game = ConnectFour(columns=conf.columns,
                                    rows=conf.rows,
                                    inarow=conf.inarow,
                                    mark=observation.mark,
                                    board=observation.board)
            root = self.init_root_node(root_game)
        best = self.perform_search(root)
        self._store_root(root.children[best])
        return best
```

Listing 3: Die Methode `get_move` des `TreePlayers`

3.2 Vier Gewinnt

Die Monte-Carlo-Baumsuche benötigt eine Implementierung des Spiels für die Simulation. Die Geschwindigkeit dieser Implementierung ist entscheidend dafür, dass die Baumsuche bei einem strengen Zeitlimit erfolgreich ist. Je mehr Iterationen der Baumsuche ausgeführt werden können, umso genauer ist das Ergebnis. Die Implementierung von VierGewinnt orientiert sich stark an der aus dem Paket `kaggle-environments`, zur Verfügung gestellt durch Kaggle,[45] kapselt aber das Spiel in einer eigenen Klasse.

Die wichtigsten Methoden der Klasse `games.ConnectFour` sind

- `play_move(column)` Setzt einen Stein für den aktuellen Spieler in die angegebene Spalte
- `list_moves()` Listet alle legalen Spielzüge
- `is_terminal()` Gibt `True` zurück, wenn das Spiel vorbei ist, sonst `False`
- `get_reward(player)` Gibt die Belohnung für den angegebenen Spieler zurück. 1 bei einem Sieg, -1 bei einer Niederlage und sonst 0.

Zusätzlich kann sich ein Objekt der Klasse selbst kopieren und kann einen Hash des Spielzustandes erzeugen. Dieser Hash ist für die Transpositionstabelle (siehe Kapitel 3.4.1) nötig. Die eindeutige Bezeichnung einer Spielposition kann mit `get_move_name(col, played)` erzeugt werden. Dies ist eine numerische Kodierung, die eindeutig die Spalte, Zeile und den Spieler ausdrückt.

45. „Kaggle/Kaggle-Environments“, besucht am 13. Oktober 2020, https://github.com/Kaggle/kaggle-environments.

3.3 Monte-Carlo-Baumsuche

Die Implementierung der Monte-Carlo-Baumsuche besteht aus drei Komponenten. Der `players.mcts.MCTSPlayer` enthält den eigentlichen Suchbaum, die Knoten des Baumes sind Instanzen der Klasse `players.mcts.Node` und die Simulation der Spiele wird von `players.mcts.Evaluator` durchgeführt.

Evaluator Der `Evaluator` bestimmt zunächst, aus Sicht welches Spielers der Spielzustand simuliert wird. Danach führt er so lange zufällige Züge aus, bis der Spielzustand terminal ist und gibt die Belohnung des zuvor bestimmten Spielers zurück. Mit `reset` kann der interne Zustand des `Evaluator` zurückgesetzt werden. Dies ist insbesondere für die Last-Good-Reply-Verbesserung (siehe Kapitel 3.4.3) nützlich.

```
class Evaluator:
    def __call__(self, game_state: ConnectFour) -> float:
        game = game_state.copy()
        scoring = game.get_other_player(game.get_current_player())
        while not game.is_terminal():
            game.play_move(random.choice(game.list_moves()))

        return game.get_reward(scoring)

    def reset(self):
        pass
```

Listing 4: Der `Evaluator` implementiert die Simulation des Spiels

Node Jeder `Node` speichert die durchschnittliche Belohnung, die der Knoten erhalten hat, wie häufig er besucht wurde und Verweise auf die Kindknoten und den Elternknoten. Zusätzlich hält jeder Knoten noch eine Kopie des Spielzustandes in diesem Knoten und er weiß, welche Aktionen noch nicht im Knoten ausprobiert wurden.

```
class Node:
    def __init__(self, game_state: ConnectFour, parent=None):
        self.average_value = 0  # Q(v)
        self.number_visits = 0  # N(v)

        self.children = {}
        self.parent = parent

        self.game_state = game_state
        self.possible_moves = game_state.list_moves()
        self.expanded = False

    def Q(self) -> float:
        return self.average_value
```

Listing 5: Die gespeicherten Parameter in einem Knoten

In der Methode `best_child` ist die UCT-Auswahlregel (siehe Gleichung (2), S. 14) implementiert. Diese Regel wird erst benutzt, wenn jeder Kindknoten mindestens einmal besucht wurde, `child.number_visits` kann somit nie 0 sein.

```
15      def best_child(self, C_p = 1.0):
16          n_p = math.log(self.number_visits)
17
18          def UCT(child: Node):
19              return child.Q() + C_p * math.sqrt(n_p / child.number_visits)
20
21          _, c = max(self.children.items(),
22                     key=lambda entry: UCT(entry[1]))
23          return c
```

Listing 6: Implementierung der UCT-Auswahlregel

Der Expansionsschritt der Baumsuche findet in `expand_one_child` statt. Aus den noch verfügbaren Aktionen wird eine zufällig ausgewählt und der dazugehörige Zustand erzeugt. Danach wird ein Kindknoten mit diesem Zustand an den aktuellen Knoten angefügt und die gewählte Aktion aus der Liste der Möglichkeiten entfernt.

```
24      def increment_visit_and_add_reward(self, reward: float):
25          self.number_visits += 1
26          self.average_value += (reward - self.average_value) / self.
    number_visits
27
28      def expand_one_child(self):
29          node_class = type(self)
30
31          move = random.choice(self.possible_moves)
32          next_state = self.game_state.copy().play_move(move)
33          self.children[move] = node_class(
34              game_state=next_state,
35              parent=self
36          )
37          self.possible_moves.remove(move)
38          if len(self.possible_moves) == 0:
39              self.expanded = True
40
41          return self.children[move]
```

Listing 7: Expansion und Aktualisierung eines Knotens

MCTSPlayer Die `MCTSPlayer` Klasse implementiert den eigentlichen Algorithmus der Baumsuche in der Methode `perform_search` (vergleiche dazu Algorithmus 1).

```
class MCTSPlayer(TreePlayer):
    def __init__(
            self,
            exploration_constant: float = 1.0,
            **kwargs
    ):
        super(MCTSPlayer, self).__init__(**kwargs)
        self.exploration_constant = exploration_constant
        self.evaluate = Evaluator()

    def perform_search(self, root) -> int:
        while self.has_resources():
            leaf = self.tree_policy(root)
            reward = self.evaluate(leaf.game_state)
            self.backup(leaf, reward)
        return self.best_move(root)
```

Listing 8: Implementierung des UCT-Algorithmus im MCTSPlayer

Die `tree_policy` beginnt jeden Durchlauf in der Wurzel des Baumes. Solange kein terminaler Zustand erreicht wurde und der aktuelle Knoten vollständig expandiert ist, wird mit `best_child` der wichtigste Kindknoten gemäß der UCT-Auswahlregel gewählt und besucht. Die Suche wird aus diesem Kindknoten fortgesetzt. Wurde ein unvollständiger Knoten gefunden oder der Knoten ist terminal, so wird der gefundene Knoten zurückgegeben. Der zückgegebene Knoten ist ein Blattknoten des aktuellen Baumes (auch wenn dieser Knoten später zu einem inneren Knoten werden könnte) der simuliert und/oder bewertet werden kann.

```
    def tree_policy(self, root):
        current = root
        while not current.game_state.is_terminal():
            if current.is_expanded():
                current = current.best_child(self.exploration_constant)
            else:
                return current.expand_one_child()
        return current
```

Listing 9: Die allgemeine Tree Policy der Monte-Carlo-Baumsuche

Nachdem der Knoten simuliert wurde, wird das Spielergebnis in `backup` vom Blattknoten zur Wurzel propagiert. Da VierGewinnt ein Minimax-Spiel ist, wird die Belohnung mit jedem Schritt negiert. `best_move` wird ausgeführt, wenn die Suchzeit abgelaufen ist und eine Entscheidung getroffen werden muss. Diese Methode gibt den Zug zurück, der zum Knoten mit der besten Bewertung führt.

```
26      def backup(self, node, reward: float):
27          current = node
28          while current is not None:
29              current.increment_visit_and_add_reward(reward)
30              reward = -reward
31              current = current.parent
32
33      def best_move(self, node) -> int:
34          move, n = max(node.children.items(), key=lambda c: c[1].Q())
35          return move
36
37      def init_root_node(self, root_game) -> Node:
38          return Node(root_game)
```

Listing 10: Implementierung der Backup-Phase

Der `MCTSPlayer` hat einen konfigurierbaren Parameter `exploration_constant`. Dies ist die Konstante C_p, die das Gleichgewicht zwischen Erkunden und Ausnutzen steuert. Der optimale Parameter wird in Kapitel 6.2 bestimmt.

3.4 Verbesserungen

Die in diesem Unterkapitel vorgestellten Verbesserungen bauen auf dem oben vorgestellten `MCTSPlayer` auf und erweitern ihn um neue Funktionalität.

3.4.1 Transpositionen

Wenn Transpositionen berücksichtigt werden, wird der Baum zu einem gerichteten Graph. Das bedeutet, jeder Knoten kann mehrere Elternknoten haben. Zusätzlich muss jeder Knoten Statistiken darüber führen, wie häufig ein Kindknoten **aus diesem Knoten heraus** besucht wurde. Diese zusätzlichen Informationen werden in der Klasse `TranspositionNode` gespeichert. Der Zugriff auf die Kind-Statistiken geschieht mit den Funktionen `N(v,a)` und `Q(v,a)`.

```
def Q(v, a=None):
    if a is None:
        return v.average_value
    else:
        return v.child_values[a]

def N(v, a=None):
    if a is None:
        return v.number_visits
    else:
        return v.child_visits[a]
```

Listing 11: Funktionen für den Zugriff auf die Statistiken Q(v) und Q(v,a) bzw. N(v) und N(v,a) eines Knotens

Der TranspositionNode enthält nun nicht mehr nur seine eigene Statistik, sonder auch die seiner Kinder `child_values` und `child_visits`.

```
class TranspositionNode(Node):
    def __init__(self, *args, **kwargs):
        super(TranspositionNode, self).__init__(*args, **kwargs)
        self.parents = []

        self.child_values = defaultdict(float)
        self.child_visits = defaultdict(int)
        self.UCT3_val = 0
        self.sim_reward = 0
```

Listing 12: Erweiterung der Klasse Node um die Kind-Statistiken

In `best_child` sind die drei UCT-Auswahlregeln von Childs u.a., **UCT1**, **UCT2** und **UCT3**, implementiert. Für UCT3 ist es zusätzlich notwendig, dass jeder Knoten den Q^{UCT3}-Wert speichert, damit er nicht in jedem Durchlauf der Tree Policy neu berechnet werden muss. Durch den Minimax-Baum muss auch beim Berechnen von Q^{UCT3} in `update_QUCT3` der Wert der Kinder negiert werden.

```
    def best_child(self, C_p = 1.0, uct_method = "UCT") -> "Node":
        n_p = math.log(self.number_visits)
        v = self

        def UCT1(a, _):
            return Q(v, a) + C_p * math.sqrt(n_p / N(v, a))

        def UCT2(a, child):
            return Q(child) + C_p * math.sqrt(n_p / N(v, a))

        def UCT3(a, child):
            return child.UCT3_val + C_p * math.sqrt(n_p / N(v, a))

        def default(_, child):
            return Q(child) + C_p * math.sqrt(n_p / N(child))
```

```
36
37          selection_method = default
38          if uct_method == "UCT1":
39              selection_method = UCT1
40          elif uct_method == "UCT2":
41              selection_method = UCT2
42          elif uct_method == "UCT3":
43              selection_method = UCT3
44
45          _, c = max(self.children.items(),
46                     key=lambda ch: selection_method(*ch))
47          return c
48
49      def update_QUCT3(self):
50          if not self.children:
51              self.UCT3_val = self.sim_reward
52          else:
53              summed = 0
54              for a, c in self.children.items():
55                  c_n = N(self, a)
56                  c_v = -c.UCT3_val
57                  summed += c_n * c_v
58              self.UCT3_val = (self.sim_reward + summed) / N(self)
```

Listing 13: Anpassung der Methode `best_child` um die verschiedenen Transpositions-Algorithmen zu unterstützen

TranspositionPlayer Der Transpositionplayer wird um eine Transpositionstabelle erweitert. Jedes Mal, wenn ein neuer Knoten hinzugefügt werden soll, wird zunächst geprüft, ob der neue Zustand nicht bereits in der Tabelle vorhanden ist. Ist er vorhanden, wird der Knoten aus der Tabelle verwendet. Die `tree_policy` kann jetzt nicht mehr nur den Blattknoten zurückgeben, sie muss den gesamten Pfad durch den Baum zurückgeben, damit in `backup` die korrekten Knoten aktualisiert werden können.

```
1  class TranspositionPlayer(MCTSPlayer):
2      def __init__(self, uct_method = "UCT", **kwargs):
3          super(TranspositionPlayer, self).__init__(**kwargs)
4          self.transpositions = {}
5          self.uct_method = uct_method
6
7      def get_or_store_transposition(self, state):
8          if state in self.transpositions:
9              return self.transpositions[state]
10         tp_node = TranspositionNode(game_state=state)
11         self.transpositions[state] = tp_node
12         return tp_node
13
14     def tree_policy(self, root: TranspositionNode) -> List[]:
```

```
15          # gekürzt ...
```

Listing 14: Behandlung von Transpositionen im TranspositionPlayer

Die `backup` Methode aktualisiert zusätzlich zu den normalen Statistiken $Q(v)$ und $N(v)$ auch die Kindstatistiken $Q(v, a)$ und $N(v, a)$. `backup` durchläuft den Pfad, den die `tree_policy` erzeugt hat, rückwärts und aktualisiert dabei die Knoten. Wenn die **UCT3** Auswahlregel in `best_child` benutzt wird, so wird auch die Q^{UCT3} Statistik beginnend im Blattknoten Schicht für Schicht aktualisiert.

```
16      def backup(self, path: List[TranspositionNode], reward: float):
17          nodes_to_update = set()
18
19          leaf = path[-1]
20          leaf.sim_reward = reward
21
22          prev = None
23          for _node in reversed(path):
24              #...
25              if prev is not None:
26                  ms = _node.find_child_actions(prev)
27                  for m in ms:
28                      _node.child_visits[m] += 1
29                      _node.child_values[m] += (-reward - _node.child_values[
    m]) / _node.child_visits[m]
30
31              if self.uct_method == "UCT3":
32                  nodes_to_update.add(_node)
33                  new_updates = set()
34                  for node in nodes_to_update:
35                      node.update_QUCT3()
36                      if node.parents:
37                          new_updates.update(node.parents)
38                  nodes_to_update = new_updates
39
40              prev = _node
41              reward = -reward
42
43      def init_root_node(self, root_game):
44          return TranspositionNode(root_game)
```

Listing 15: Die Methode `backup` muss jetzt zusätzlich die Kind-Statistiken aktualisieren. Dies geschieht um eine Ebene im Baum versetzt.

Um den Code kürzer zu halten, wurde die Behandlung von Symmetrien ausgelassen. Der finale Agent mit Transpositionen kann, wie auch der `MCTSSpieler` mit dem Parameter C_p konfiguriert werden. Zusätzlich kann die Auswahlregel der Tree Policy gewählt werden und Symmetrien berücksichtigt werden. Wie sich diese Parameter auf die Leistung des Agenten auswirken, wird in Kapitel 6.3 untersucht.

3.4.2 All Moves as First

Beide All Moves as First Varianten wurden in einem Spieler umgesetzt. Wenn der Parameter α gesetzt ist, wird die α-AMAF Variante benutzt, sonst RAVE.

RaveEvaluator Der Evaluator hat nun einen internen Zustand. Er führt eine Liste mit allen eindeutigen Namen der Züge, die während einer Simulationen ausgewählt wurden.

```
class RaveEvaluator(Evaluator):
    def __init__(self):
        self.moves = []

    def __call__(self, game_state: ConnectFour):
        game = game_state.copy()
        scoring = game.get_other_player(game.get_current_player())
        while not game.is_terminal():
            m = random.choice(game.list_moves())
            move_name = game.get_move_name(m)
            self.moves.append(move_name)
            game.play_move(m)

        return game.get_reward(scoring)
```

Listing 16: Der `RaveEvaluator` speichert alle gemachten Züge in einer Liste

RaveNode Die Knoten im Baum des `RavePlayer` speichern die AMAF-Statistiken und einen zweiten Verweis auf die Kinder des Knotens. Dieser zweite Verweis erlaubt es, über den eindeutigen Zugnamen auf das Kind zuzugreifen. In `best_child` werden die AMAF- und UCT-Statistiken kombiniert. Wenn α gesetzt ist, dann wird dieser Wert als Gewicht benutzt, sonst wird β gemäß Gleichung (7) auf Seite 20 berechnet.

```
class RaveNode(Node):
    def __init__(self, *args, **kwargs):
        super(RaveNode, self).__init__(*args, **kwargs)
        self.rave_count = 0
        self.rave_score = 0
        self.rave_children = {}

    def beta(self, k=50) -> float:
        return math.sqrt(k / (3*self.number_visits + k))

    def best_child(self, C_p = 1.0, alpha = None, k=50)]:
        n_p = math.log(self.number_visits)

        def UCT_Rave(child: RaveNode):
            if alpha is not None:
                beta = alpha
            else:
```

```
                beta = child.beta(k)
            return (1 - beta) * child.Q() + beta * child.QRave() \
                   + C_p * math.sqrt(n_p / child.number_visits)

        m, c = max(self.children.items(), key=lambda c: UCT_Rave(c[1]))
        return c, m

    def expand_one_child(self):
        # ...
        child_state = self.game_state.copy().play_move(move)
        move_name = child_state.get_move_name(move, played=True)
        self.children[move] = nodeclass(
            game_state=child_state,
            parent=self
        )
        self.rave_children[move_name] = self.children[move]

        return self.children[move]
```

Listing 17: Der `RaveNode` kombiniert die AMAF-Statistik mit der UCT-Statistik. Bei der Expansion eines Kindknotens wird auch die Aktion, die in das Kind führt gespeichert.

RavePlayer Im `RavePlayer` aktualisiert die `backup` Methode zusätzlich die AMAF-Statistiken. Dabei wird in jedem Knoten auf dem Weg vom Blatt zur Wurzel geprüft, welche direkten Kinder über einen in der Simulation gemachten Zug erreicht werden können. Diese Kinder werden aktualisiert.

```
class RavePlayer(MCTSPlayer):
    name = "RavePlayer"

    def __init__(self, alpha: float = None, k:int = 50, *args, **kwargs):
        super(RavePlayer, self).__init__(*args, **kwargs)
        self.evaluate = RaveEvaluator()
        self.alpha = alpha
        self.k = k

    def backup(self, node: RaveNode, reward: float):
        move_set = set(self.evaluate.moves)
        current = node
        while current is not None:
            # ...
            for mov, child in current.rave_children.items():
                if mov in move_set:
                    child.increment_rave_visit_and_add_reward(-reward)

            reward = -reward
            current = current.parent

    def init_root_node(self, root_game):
```

```
23          return RaveNode(game_state=root_game)
```

Listing 18: In `backup` werden auch alle Knoten aktualisiert, die durch einen in der Simulation gemachten Zug erreicht werden können

3.4.3 Last-Good-Reply

Für die Implementierung der Last-Good-Reply-Policy muss nur der `Evaluator` angepasst werden. Der `AdaptiveEvaluator` wird um die Antwort-Tabelle `replies` erweitert, in der die guten Antworten gespeichert werden. Diese Tabelle kann zwischen zwei Spielzügen beibehalten werden. Der allgemeine Ablauf der Simulation bleibt der selbe, in jeder Simulation werden aber die Antworten auf gegnerische Züge zwischengespeichert. Am Ende der Simulation werden die erfolgreichen Antworten in die Antwort-Tabelle geschrieben.

```
1  class AdaptiveEvaluator(Evaluator):
2      def __init__(self, forgetting=False, keep_replies=False):
3          self.replies = {}
4          self.forgetting = forgetting
5          self.keep_replies = keep_replies
6
7      def __call__(self, game_state: ConnectFour) -> float:
8          game = game_state.copy()
9          simulating_player = game.get_current_player()
10         scoring = 3 - simulating_player
11
12         last_move = None
13         simulation_replies = {}
14
15         while not game.is_terminal():
16             move = self.get_next_move(game, last_move)
17             game.play_move(move)
18
19             move_name = game.get_move_name(move, played=True)
20             if last_move is not None:
21                 simulation_replies[last_move] = move_name
22
23             last_move = move_name
24
25         self.memorize(simulation_replies, game.winner)
26         return game.get_reward(scoring)
27
28     def reset(self):
29         if not self.keep_replies:
30             self.replies = {}
```

Listing 19: Der `AdaptiveEvaluator` erweitert den `Evaluator` um eine interne Antworten-Tabelle, die zwischen den Simulationen erhalten bleibt. Diese Tabelle wird in `get_next_move` verwendet, um den nächsten Zug der Simulation zu wählen.

Die Züge in der Simulation werden nicht mehr rein zufällig gewählt. Die Methode `get_next_move` sucht in der Antwort-Tabelle nach einer möglichen Antwort auf den Zug des Gegenspielers und prüft, ob diese Antwort legal ist. Hierfür wird nicht nur sichergestellt, dass die Spalte der Antwort spielbar ist, sonder auch, dass der Stein in der selben Zeile landen würde. Wurde kein legaler Zug gefunden, so wird ein zufälliger Zug gewählt. `memorize` sorgt dafür, dass die erfolgreichen Antworten in der nächsten Simulation wiederverwendet werden können. Wenn `forgetting` aktiviert ist, werden nicht erfolgreiche Antworten wieder aus der Tabelle entfernt.

```
31      def get_next_move(self, game: "ConnectFour", last_move: int) -> int:
32          move = None
33          legal_moves = game.list_moves()
34          if last_move is not None and last_move in self.replies:
35              reply = self.replies[last_move]
36              # prüfe ob die Antwort in diesem Zustand möglich ist
37              reply_move = (reply // 10) % game.cols
38              mname = game.get_move_name(reply_move, played=False)
39              if reply == mname:
40                  move = reply_move
41
42          if move not in legal_moves:
43              move = random.choice(game.list_moves())
44
45          return move
46
47      def memorize(self, replies, winner: int):
48          if winner is not None:
49              loser = 3 - winner
50              for move, reply in replies.items():
51                  replying_to = move % 10
52                  if replying_to == loser:
53                      self.replies[move] = reply
54                  elif self.forgetting and replying_to == winner:
55                      self.replies.pop(move, None)
```

Listing 20: `get_next_move` versucht eine Antwort aus der Antworten-Tabelle zu verwenden. Wenn keine legale Antwort vorhanden ist, wird ein zufälliger Zug benutzt. `memorize` speichert die guten Antworten des Spielers, der die Simulation gewonnen hat.

Die Genauigkeit der Simulation kann gesteigert werden, indem eine eigene Policy benutzt wird um die Simulation zu steuern. Last-Good-Reply lernt diese Policy automatisch, sie kann aber auch durch externes Wissen vorgegeben werden.

Eine Alternative zu einer Verbesserung der Simulationsgenauigkeit, ist eine konkrete Bewertung des Spielzustands durch eine Heuristik. Diese Heuristiken sind aber schwierig akkurat zu definieren und für komplexe Spiele wie Go sind keine guten Heuristiken bekannt. Stattdessen können neuronale Netze eingesetzt werden, um aus einer großen Menge von

Daten eine Bewertungsfunktion zu lernen. Im nächsten Kapitel werden künstliche neuronale Netze vorgestellt und im darauf folgenden Kapitel werden eigene künstliche neuronale Netze entwickelt und trainiert, um eine solche Bewertungsfunktion für VierGewinnt zu lernen.

4 Künstliche neuronale Netze

Das Ziel dieser Arbeit ist es, einen MCTS-Agenten zu entwickeln und mit neuronalen Netzen zu kombinieren. Bevor die Herangehensweise bei der Entwicklung des neuronalen Netzes in Kapitel 5 vorgestellt werden kann, muss zunächst erklärt werden, was ein künstliches neuronales Netz ist und wie es funktioniert. Dieses Kapitel liefert einen kurzen Überblick über die Funktionsweise von künstlichen neuronalen Netzen und führt eine Reihe von Begriffen ein, die in Kapitel 5 benutzt werden, um neuronale Netze zu beschreiben.

Künstliche neuronale Netze (KNN), häufig auch nur neuronale Netze genannt, haben in den letzten Jahren das Forschungsfeld des maschinellen Lernens revolutioniert.

Die Ideen hinter KNNs stammen aber bereits aus den 1940er und 50er Jahren. McCulloch und Pitts haben 1943 ein Modell der Neuronen als Recheneinheit vorgestellt,[46] das die Funktionsweise von biologischen Neuronen imitieren sollte. Basierend auf dieser Arbeit hat Frank Rosenblatt 1957 das Perzeptron (von engl. *to percept*, etwas wahrnehmen) erfunden.[47] Das Perzeptron ist eine Maschine, die binäre Inputs verarbeitet und einen binären Output erzeugt. Es kann durch äußeres Einwirken dazu gebracht werden, seine Parameter anzupassen, wenn der erzeugte Output falsch ist - es kann lernen. Das Perzeptron ist auch heute noch die Basis für die Einheiten, aus denen künstliche neuronale Netze bestehen.

Ein Perzeptron alleine kann einfache aussagenlogische Verknüpfungen wie AND, OR und NAND abbilden, scheitert aber an XOR und anderen nicht linear separierbaren Problemen[48]. Dies haben Minsky und Papert in ihrem Buch Perceptrons 1969 gezeigt.[49]

Abhilfe schafft der sogenannte Multilayer Perceptron (MLP), ein mehrschichtiges Netzwerk aus Perzeptronen, allgemeiner einfach nur als Neuronen bezeichnet. Durch die Verknüpfung mehrerer Neuronen können auch komplexere Probleme wie das XOR Problem gelöst werden, doch das Training eines solchen Netzwerks hat lange für Schwierigkeiten gesorgt. Erst in den 80ern wurde es durch Anwendung des Backpropagation Algorithmus mit Hilfe von automatischer Differenzierung möglich auch tiefere neuronale Netze zu trainieren.

46. Warren S. McCulloch und Walter Pitts, „A Logical Calculus of the Ideas Immanent in Nervous Activity“, *The bulletin of mathematical biophysics* 5, Nr. 4 (1. Dezember 1943): 115–133.

47. Frank Rosenblatt, „The Perceptron: A Probabilistic Model for Information Storage and Organization“, *Psychologial Review* 65, Nr. 6 (November 1958): 386–408.

48. Zwei Mengen von Punkten A und B im 2-dimensionalen Raum sind linear separierbar, wenn eine Gerade so platziert werden kann, dass alle Punkte von A auf einer Seite der Linie liegen und alle Punkte von B auf der anderen.

49. Perceptrons *(Book)*, in *Wikipedia* (12. Mai 2020), besucht am 31. Juli 2020, https://en.wikipedia.org/w/index.php?title=Perceptrons_(book)&oldid=956342184.

4.1 Neuronen

Neuronen sind kleine Recheneinheiten mit einem oder mehreren Eingängen, die abhängig von den anliegenden Signalen eine Ausgabe erzeugen, man spricht von einer Aktivierung des Neurons. Jeder Eingang ist mit einem Gewicht assoziiert, das bestimmt, wie wichtig der jeweilige Eingang ist und eine Aktivierungsfunktion entscheidet dann, ob das anliegende Signal zu einer Aktivierung des Neurons führt. Ein Schwellenwert, auch Bias genannt, kann dazu verwendet werden, um diese Aktivierungsschwelle zu verschieben und somit die Aktivierung zu erleichtern oder zu erschweren.

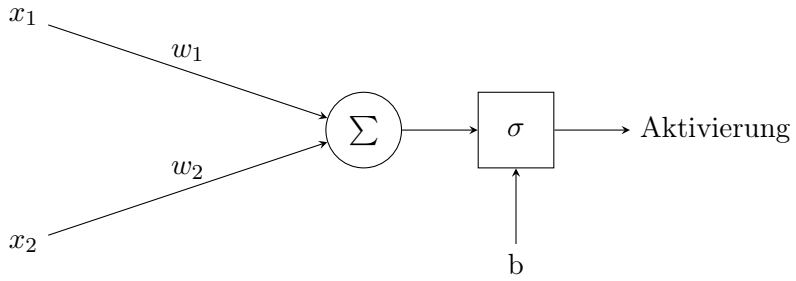


Abbildung 11: Ein Neuron mit zwei Eingängen x_1, x_2, zwei Gewichten w_1, w_2, Schwellenwert b und Aktivierungsfunktion σ

Der Wert des Neurons berechnet sich aus der Linearkombination der Eingänge mit den Gewichten plus dem Bias $z = w_0x_0 + w_1x_1 + b$. Auf diesen Wert wird eine **Aktivierungsfunktion** angewandt, um den Ausgabewert des Neurons zu erhalten. Im klassischen Perzeptron ist diese Aktivierungsfunktion die Stufenfunktion

$$STEP(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases} \tag{8}$$

.

Die Ausgabe ist 1, wenn die gewichtete Summe der Eingänge den Schwellenwert überschreitet, sonst 0. Diese Funktion hat einen großen Nachteil: Änderungen an den Gewichten haben keinen Einfluss auf die Ausgabe, bis der Schwellenwert durch diese Änderung über- bzw. unterschritten wird. Es ist somit schwierig, die Parameter genau anzupassen.

Heute werden viele verschiedene Aktivierungsfunktionen eingesetzt. Die häufigste ist die Sigmoid Funktion

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{9}$$

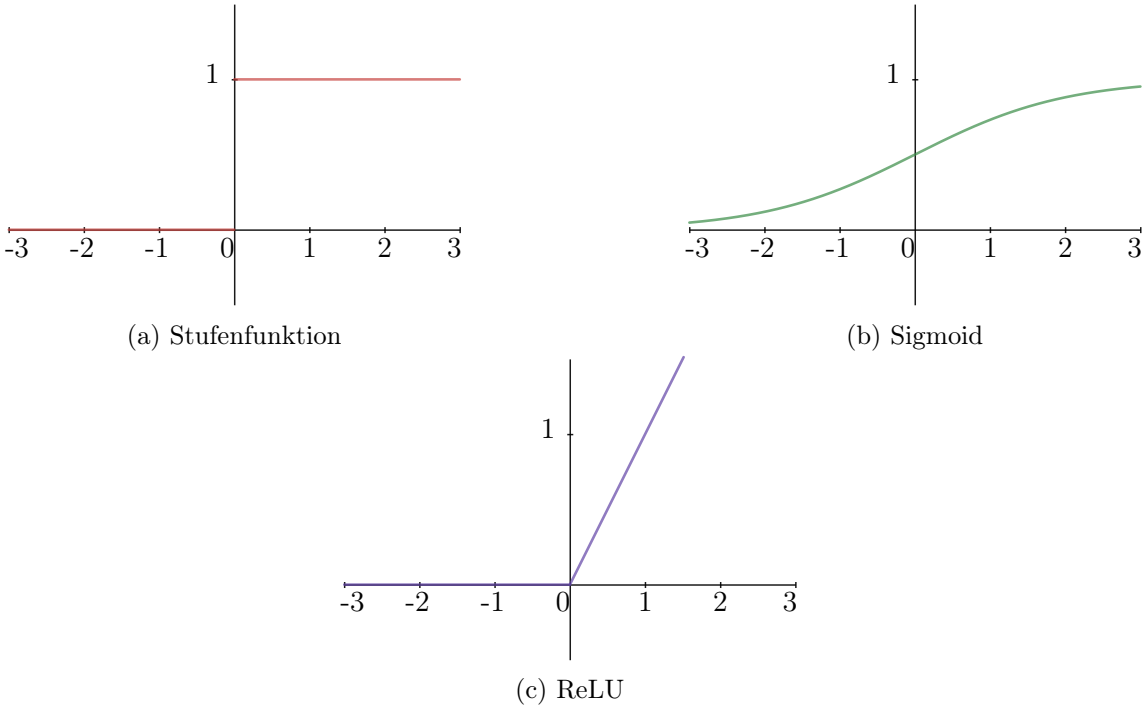
Sie wird, genauso wie die Stufenfunktion, für große positive Werte 1 und für große negative Werte 0, dazwischen steigt sie stetig an. Der S-Förmige Kurvenverlauf dieser Funktion bewirkt, dass kleine Veränderungen der Gewichte nur kleine Änderungen in der Ausgabe

erzeugen. Damit lassen sich Neuronen mit dieser Aktivierungsfunktion besser trainieren.[50]

Neben Sigmoid ist die ReLU Funktion eine weitere beliebte Aktivierungsfunktion. ReLU steht für **Re**ctified **L**inear **U**nit und ist eine sehr einfache Funktion:

$$ReLU(z) = max(0, z) \tag{10}$$

Der Wert von ReLU ist 0 für alle $z < 0$ und sonst z.



(a) Stufenfunktion

(b) Sigmoid

(c) ReLU

Abbildung 12: Graph der Aktivierungsfunktionen

Abb. 12 zeigt den Kurvenverlauf der vorgestellten Aktivierungsfunktionen.

Aktivierungsfunktionen sind nötig, um eine Nichtlinearität in das System einzubringen. Jedes Neuron berechnet eine Linearkombination seiner Inputs. Ein Netzwerk dieser Neuronen ohne Nichtlinearität führt dazu, dass der Output ebenfalls nur eine Linearkombination der Inputs des Netzwerks ist. Egal wieviele Schichten das Netzwerk hat, der Output verhält sich so, als gäbe es nur eine Schicht. Aktivierungsfunktionen sind daher nichtlinear, wodurch sich jede beliebige Funktion approximieren lässt.[51]

Die in diesem Kapitel vorgestellten Neuronen können zu einem Netzwerk zusammengeschaltet werden. Das nächste Kapitel stellt eine einfache Art eines neuronalen Netzes, ein vollständig verbundenes Netzwerk, vor.

50. Siehe Michael A. Nielsen, „Neural Networks and Deep Learning“, 2015, Kapitel 1, besucht am 30. Juli 2020, http://neuralnetworksanddeeplearning.com.
51. Siehe Nielsen, Kapitel 4.

4.2 Architektur eines neuronalen Netzes

Neuronale Netze bestehen aus mehreren Schichten mit jeweils einem oder mehreren Neuronen. Die erste Schicht ist der **Input layer** (Eingabeschicht). Häufig werden die Inputs als Neuronen dargestellt, mit einem Neuron pro Input-Feature. Die Neuronen der Eingabeschicht haben keine Aktivierungsfunktion und reichen ihren Input unverändert an die nächste Schicht weiter. Generell wird die Eingabeschicht bei der Zählung der Schichten eines Netzwerks nicht mitgezählt (siehe Abb. 13).

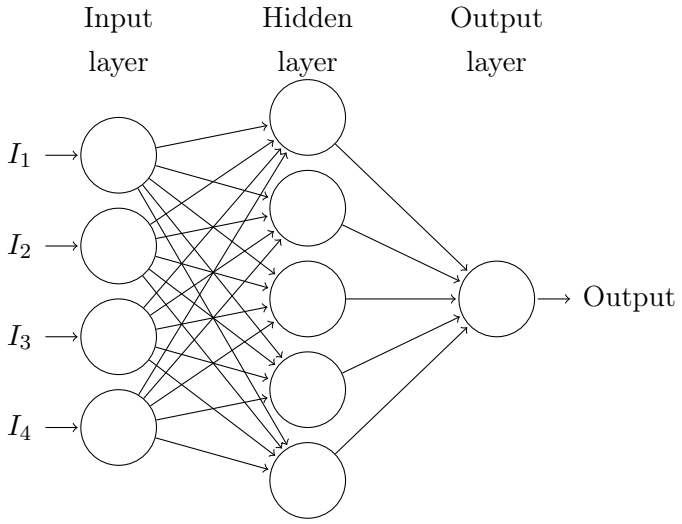


Abbildung 13: Ein neuronales Netz mit 2 Schichten, die Eingabeschicht wird nicht mitgezählt

Alle Schichten zwischen der ersten und letzten Schicht werden als **Hidden layer** (versteckte Schichten) bezeichnet. In einem einfachen neuronalen Netz sind alle Neuronen mit allen Neuronen der vorherigen Schicht verbunden, man spricht von vollständig verbundenen Schichten (fully connected layers oder auch „dense“ layers). Die Neuronen in den versteckten Schichten haben häufig eine andere Aktivierungsfunktion als die Ausgabeschicht. Allgemein könnte jedes Neuron eine eigene Aktivierungsfunktion besitzen, in der Praxis ist es aber üblich, dass alle Neuronen einer Schicht die selbe Aktivierungsfunktion haben. Während anfangs die Sigmoid Funktion für die versteckten Schichten bevorzugt wurde, da diese Aktivierung der der biologischen Neuronen am nächsten kommt, wird heutzutage häufig ReLU eingesetzt.

Die letzte Schicht wird **Output layer** (Ausgabeschicht) genannt. Die Ausgabeschicht besteht aus einem Neuron für jeden Zielwert. Ein Netzwerk für die Bestimmung eines Hauspreises zum Beispiel hat nur ein Output Neuron, ein Netzwerk für die Klassifizierung von handgeschriebenen Ziffern dagegen hat ein Neuron für jede mögliche Ziffer 0 bis 9. Die Aktivierungsfunktion der Ausgabeschicht wird abhängig von der zu lösenden Aufgabe gewählt.

Die Gewichte der Verbindungen zwischen Neuronen werden in der Regel als eine Matrix

repräsentiert, während die Inputs einer Schicht und die Bias-Werte Vektoren sind. Somit lässt sich durch lineare Algebra die Aktivierung einer Schicht L mit Aktivierungsfunktion σ berechnen:

$$a^L = \sigma(w^L \cdot a^{L-1} + b^L). \tag{11}$$

Ist L die erste versteckte Schicht, so ist a^{L-1} der Vektor der Inputs. Jede Zeile der Matrix entspricht dabei den Gewichten eines Neurons und jede Spalte den Inputs in dieser Schicht.

Neuronale Netze werden für verschiedenste Aufgaben verwendet, die Hauptanwendungsfälle lassen sich in **Klassifizierung** und **Regression** aufteilen.

Das Ziel der Klassifizierung ist es, die Beispiele, die das System verarbeitet, in Klassen einzuteilen - es sollen diskrete Werte vorhergesagt werden. So kann ein Klassifizierer beispielsweise identifizieren, ob auf einem Bild ein Hund oder eine Katze zu sehen ist.

Regressionsaufgaben sind Aufgaben, bei denen ein kontinuierlicher Zielwert vorhergesagt werden soll. Zum Beispiel der Preis eines Hauses basierend auf Informationen wie der Lage, dem Baujahr und der Fläche des Grundstücks. Für die Regression benötigt das neuronale Netz einen Output für jeden Zielwert, der vorhergesagt werden soll.

Für Regression wird häufig keine Aktivierungsfunktion in der Ausgabeschicht benutzt, wenn ein skalarer Wert, wie ein Preis, bestimmt werden soll. Liegt der Zielwert dagegen in einem bestimmten Wertebereich, so kann auch hier zum Beispiel die Sigmoid Funktion verwendet werden. Soll der Wert immer positiv sein, so kann die ReLU Funktion eingesetzt werden.

4.3 Training eines neuronalen Netzes

Bevor ein neuronales Netz eingesetzt werden kann, muss es zunächst angelernt werden. Für dieses Training ist eine große Menge an Trainingsdaten notwendig und das neuronale Netz muss diese Trainingsdaten mehrfach durchlaufen, bis es sichere Vorhersagen treffen kann. Jeder Durchlauf durch die gesamten Trainingsdaten wird als **Epoche** bezeichnet. Jedes Beispiel der Trainingsdaten besteht aus Features, den Eingabewerten in das neuronale Netzwerk, und Zielwerten. Das Netzwerk lernt dabei die Zusammenhänge zwischen den Features und den Zielwerten.

4.3.1 Fehlerfunktion

In jedem Durchlauf durch diese Trainingsdaten wird mit einer **Fehlerfunktion** der Fehler des Netzwerks berechnet. Dieser Fehler gibt an, wie sehr das Netzwerk im Durchschnitt mit seinen Vorhersagen daneben lag. Die Fehlerfunktion (cost function oder auch loss function) muss abhängig von der Aufgabe und der Art der Trainingsdaten gewählt werden.

Für Regression wird häufig der durchschnittliche Fehler im Quadrat (mean squared error, MSE) oder, wenn es viele Ausreißer in den Daten gibt, der mittlere Fehler im Betrag (mean absolute error, MAE) berechnet.

Im folgenden gilt: y ist der Zielwert, $\hat{y}$ ist die Vorhersage des Netzwerks, m ist die Anzahl der Trainingsbeispiele und die Indexierung y_i bezeichnet das i-te Beispiel im Trainingsdatensatz.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{12}$$

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i| \tag{13}$$

Der Fehler ist gering, wenn die Vorhersage sehr nahe am erwarteten Wert liegt, und wird groß wenn die Vorhersage stark abweicht. Die Fehlerfunktion berechnet den Fehler über alle Trainingsbeispiele.

Die Vorhersagen des Netzwerks, und damit auch der Fehler, hängen direkt von den Parametern (Gewichten) des Netzwerks ab. Da die Trainingsbeispiele statisch sind, kann der Fehler nur reduziert werden, indem die Parameter des Netzwerks verändert werden. Das Ziel ist es, diese Parameter so anzupassen, dass der Wert der Fehlerfunktion minimiert wird. Das verwendete Optimierungsverfahren ist auch als Gradientenverfahren (gradient descent) bekannt.

4.3.2 Gradientenverfahren und Backpropagation

Das Ziel des Gradientenverfahrens ist es, den Fehler des Netzwerks zu minimieren. Das Minimum einer einfachen Funktion wie $y = x^2$ lässt sich durch Bilden der Ableitung $y' = \frac{\delta y}{\delta x}$ lösen, die Funktion, die das neuronale Netz repräsentiert, ist aber um ein vielfaches komplexer. Selbst das einfache Netzwerk in Abb. 13 hat bereits 25 Gewichte und 6 Bias-Werte, die entsprechende Funktion hätte also 31 Variablen. Eine solche Funktion lässt sich nicht mehr trivial ableiten und lösen.[52]

52. Siehe François Chollet, *Deep Learning with Python*, 1st (Shelter Island, New York: Manning Publications, 22. Dezember 2017), S. 49.

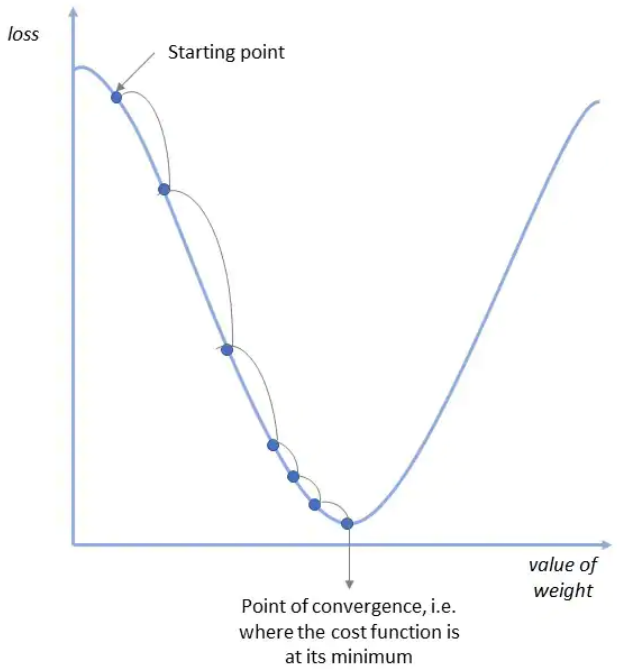


Abbildung 14: Das Gradientenverfahren nähert sich schrittweise dem Minimum der Funktion[53]

Das Gradientenverfahren, und spezieller das Verfahren des steilsten Abstiegs, ist ein numerischer Lösungsansatz für ein solches Optimierungsproblem:

Ausgehend von einem Startpunkt schreitet man so lange in Richtung des steilsten Abstiegs, bis sich der Funktionswert nicht mehr verbessert. Die Weite eines Schrittes wird über die Lernrate λ gesteuert. Ist sie zu groß gewählt, wird das Minimum möglicherweise übersprungen oder der Algorithmus konvergiert gar nicht. Ist sie zu klein gewählt, benötigt der Algorithmus sehr lange bis er konvergiert oder er konvergiert zu einem lokalen Minimum, anstatt das globale Minimum zu finden.

Für den Abstieg muss die Steigung an jedem Punkt der Funktion bestimmt werden können. Die Steigung ∇C, auch Gradient genannt, ist der Vektor der partiellen Ableitungen der Funktion C.

Die Berechnung der partiellen Ableitungen geschieht im Backpropagation-Algorithmus. Nachdem die Trainingsbeispiele durch das Netzwerk geführt wurden und der Fehler bestimmt wurde, wird der Fehler rückwärts durch das Netzwerk zurückgeführt. Zunächst wird für jeden Output bestimmt, welchen Einfluss er auf den Fehler des Netzwerks hatte. Durch Anwendung der Kettenregel wird dann die Beteiligung jeder Verbindung, also die partielle Ableitung der Fehlerfunktion in Abhängigkeit vom Gewicht der Verbindung, der

53. „What Is Gradient Descent?“, 28. Oktober 2020, besucht am 16. November 2020, https://www.ibm.com/cloud/learn/gradient-descent

vorherigen Schicht berechnet. Der Algorithmus arbeitet sich Schicht für Schicht durch das Netzwerk, bis die Eingabeschicht erreicht ist. Nachdem auf diese Weise alle partiellen Ableitungen berechnet wurden, kann nun ein Schritt des Gradientenverfahrens durchgeführt und die Parameter etwas verbessert werden.

Bibliotheken wie Tensorflow und PyTorch implementieren verschiedene Varianten dieses Optimierungsverfahrens und können die Funktionen mit Hilfe von symbolischer Differenzierung automatisch ableiten.[54]

4.4 Convolutional Neural Networks

In einem vollständig verbundenen Netzwerk existiert für jedes lernbare Feature ein Neuron im Input Layer. Diese Architektur kann auch mit wenigen hundert Neuronen noch gut funktionieren. Wenn es sich bei den Daten aber beispielsweise um Bilder handelt, so müsste das Netzwerk ein Neuron für jedes Pixel im Bild (mal drei wenn das Bild aus drei Kanälen besteht) haben. Die Anzahl der nötigen Neuronen und damit der Verbindungen zur nächsten Schicht steigt explosionsartig. Außerdem sind vollständig verbundene neuronale Netze nicht in der Lage, die zum Beispiel in der oberen Hälfte eines Bildes erkannten Muster auch an anderen Stellen wieder zu erkennen. Ist ein Objekt in einem Bild auch nur leicht verschoben, wird es nicht mehr erkannt. Die Reihenfolge und räumliche Abhängigkeit der Features wird ebenfalls ignoriert, jedes Pixel wird einzeln betrachtet, obwohl das direkte Umfeld mehr Informationen über die Natur des Pixels liefern könnte.

Convolutional Neural Networks (ConvNets) sollen dieses Problem lösen. Yann LeCun u.a. kombinieren drei Ideen um das Netzwerk robuster gegen solche Transformationen zu machen: local receptive fields, shared weights und sub-sampling.[55]

Da das Hauptanwendungsgebiet von ConvNets die Bilderkennung ist, hilft es bei der Erklärung von Convolutional Networks und den zugehörigen Operationen von Bildern und Pixeln zu reden. Die tatsächlichen Daten auf denen ein ConvNet angewandt werden kann, müssen aber keine Bilder sein.

Anstatt dass jedes Neuron einer Schicht mit allen Neuronen der vorherigen Schicht verbunden ist, wird nur noch ein kleines Fenster von Neuronen, das local receptive field, betrachtet. Das zum Beispiel 5×5 Neuronen große Fenster wird von oben links Zeile für Zeile über das Bild geschoben, bis das letzte Pixel unten rechts erreicht ist. Dabei wird das Bild für die nächste Schicht des Netzwerks kleiner, da die Fenster nicht über den Rand hinausragen können. Abb. 15 zeigt die Verbindung des ersten Neurons in der versteckten Schicht mit dem 5×5 großen Fenster der Eingabeschicht.

54. *Automatic Differentiation*, in *Wikipedia* (6. Oktober 2020), besucht am 16. November 2020, https://en.wikipedia.org/w/index.php?title=Automatic_differentiation&oldid=982160496.
55. Siehe Yann LeCun u. a., „Gradient-Based Learning Applied to Document Recognition“, 1998, S. 6.

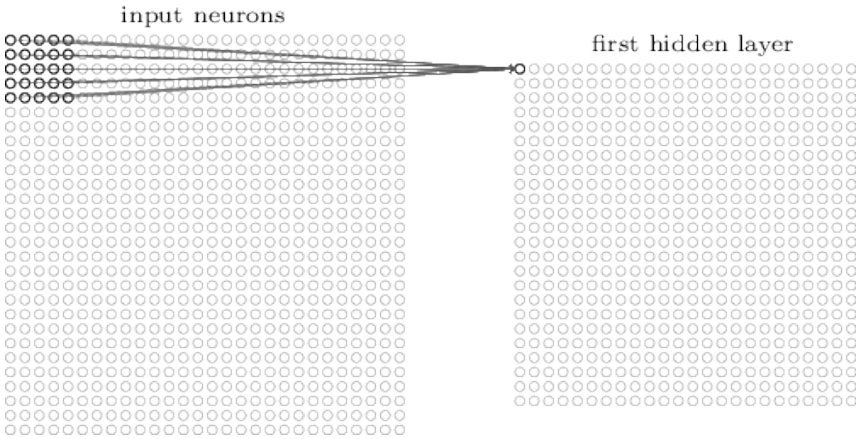


Abbildung 15: Local receptive fields in einem ConvNet[56]

Im Gegensatz zu normalen neuronalen Netzen hat jedes Neuron der versteckten Schicht keine eigenen Gewichte, alle Neuronen benutzen die selben 5×5 Gewichte. Eine solche Matrix aus Gewichten wird Filter oder Kernel genannt. Eine Schicht eines ConvNets, ein convolutional layer, kann aus mehreren solcher Filter bestehen. Abb. 16 zeigt 20 solcher Filter, wie sie von einem ConvNet gelernt werden könnten.

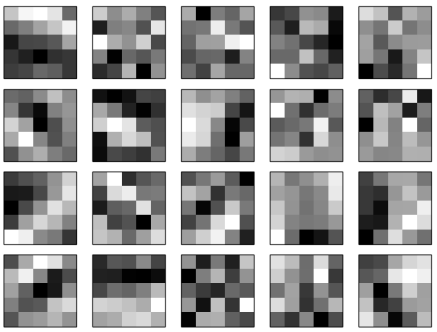


Abbildung 16: Visualisierung der Filter eines ConvNets. Helle Bereiche stehen für ein geringes Gewicht der Verbindung und dunkle Bereiche für ein hohes Gewicht.[57]

Besteht der Input beispielsweise aus einem 28×28 Pixel großen Bild und die erste versteckte Schicht hat 3 Filter, dann wird durch die erste Schicht das Bild in drei kleinere Bilder, die sogenannten Feature Maps, bestehend aus jeweils 24×24 Pixeln, zerlegt. Jede Feature Map hebt dabei ein anderes Merkmal im Bild hervor.

Durch sub-sampling wird das Bild danach weiter in der Größe reduziert. Die dafür verwendeten Pooling Layer sind meistens nur 2×2 Neuronen groß und reduzieren somit die Größe um die Hälfte. Ein Pooling Layer vereinfacht die Informationen in der Ausgabe des vorherigen Convolutional Layers, indem in jedem pooling-Fenster nur die stärkste Aktivierung benutzt wird (max-pooling). Die Pooling Layer werden auf alle Feature Maps separat angewendet.

56. Nielsen, „Neural Networks and Deep Learning“, Kapitel 6
57. Nielsen, Kapitel 6

Ein Convolutional Network reduziert Schritt für Schritt die Dimensionen des Eingabebildes und erzeugt dabei einen Stapel von Feature Maps. Mit jedem Schritt werden die Feature Maps kleiner und der Stapel größer. Im Extremfall sind die letzten Feature Maps nur noch 1×1 Pixel groß und somit äquivalent zu normalen Neuronen.

Nachdem genug Convolutional Layer und Pooling Layer verwendet wurden, werden die verbleibenden Pixel der Feature Maps als Eingabeschicht für ein flaches vollständig verbundenes Netzwerk verwendet.

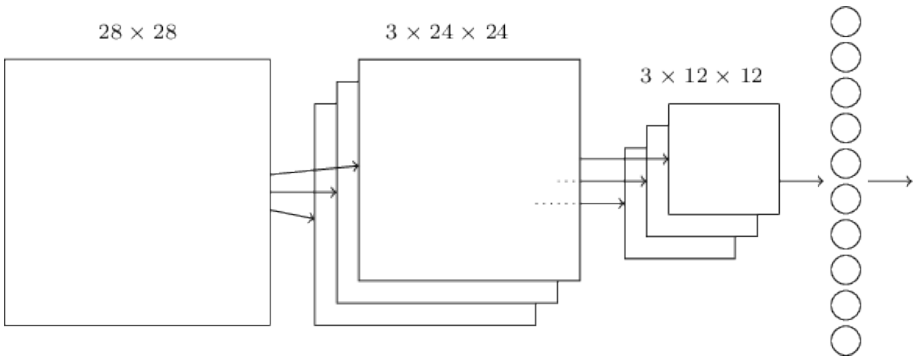


Abbildung 17: Nach Anwendung des Convolutional Layers mit drei 5×5 Filtern wird die Größe der Feature Maps durch Pooling reduziert. Danach werden die verbleibenden Neuronen der Feature Maps vollständig mit 10 Outputs verbunden.[58]

Da VierGewinnt auf einem zweidimensionalen Spielfeld gespielt wird und die Siegesbedingung von lokalen Mustern abhängt, sollten sich Convolutional Networks gut eignen, um das Spiel lernen zu können. Dass neuronale Netze lernen können (Brett-)Spiele zu spielen hat DeepMinds AlphaGo[59] bereits gezeigt. Ob diese Annahme auch auf VierGewinnt zutrifft, wird in den nächsten Kapiteln untersucht.

58. Nielsen, „Neural Networks and Deep Learning“, Kapitel 6
59. „AlphaGo“.

5 Implementierung der neuronalen Netze

Das Finden des richtigen neuronalen Netzes für eine gegebene Aufgabe ist schwierig und mit sehr viel Experimentieren verbunden. Bereits ein einfaches, vollständig verbundenes neuronales Netz kann mit unzähligen Parametern konfiguriert werden. Es gilt, die richtige Anzahl an Schichten mit der richtigen Menge an Einheiten pro Schicht zu finden, gute Aktivierungsfunktionen im Netzwerk auszuwählen, die Lernrate zu optimieren, den Lernalgorithmus zu wählen und mehr. Um die Suche nach guten Netzwerk-Architekturen und guten Werten für diese Hyperparameter zu erleichtern, wurden Werkzeuge entwickelt, die einen großen Teil dieser Variablen automatisch erforschen. Man spricht von automatisiertem maschinellem Lernen.

Für die Suche nach einem guten Netzwerk wird in dieser Arbeit AutoKeras verwendet.

5.1 AutoKeras

AutoKeras wurde vom Data LAB an der Texas A&M University mit dem Ziel entwickelt, maschinelles Lernen zugänglicher zu machen.[60]

Im einfachsten Anwendungsfall ist es nur noch notwendig zu definieren, ob es sich um eine Klassifizierungs- oder Regressions-Aufgabe handelt und ob die Daten als Bild, Text oder strukturierte Daten vorliegen. AutoKeras erforscht dann einen vordefinierten Hyperparameter-Raum, um die Netzwerk-Konfiguration zu finden, die die höchste Genauigkeit auf den Trainingsdaten erzielt.

Ein AutoKeras-Modell kann aus mehreren Blöcken bestehen. Jeder Block kapselt einen parametrisierbaren Teil eines neuronalen Netzes. Ein `denseBlock` beispielsweise besteht aus bis zu drei vollständig verbundenen Netzwerk-Schichten mit einer variablen Anzahl Einheiten in jeder Schicht. Für komplexere Aufgaben, wie die Klassifikation von Bildern, stellt AutoKeras vortrainierte Blöcke, wie ein auf dem ImageNet-Datensatz[61] trainiertes ResNet[62] und einen Xception-Block,[63] zur Verfügung.

Für VierGewinnt sollte ein einfaches Netzwerk aus vollständig verbundenen Schichten und Convolutional Layern ausreichend sein.

60. Haifeng Jin, Qingquan Song und Xia Hu, „Auto-Keras: An Efficient Neural Architecture Search System“, in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, KDD '19 (New York, NY, USA: Association for Computing Machinery, 25. Juli 2019), 1946–1956.
61. Siehe „ImageNet“, besucht am 17. Oktober 2020, http://www.image-net.org/.
62. Siehe Kaiming He u. a., „Deep Residual Learning for Image Recognition“, 10. Dezember 2015, arXiv: 1512.03385 [cs].
63. Siehe François Chollet, „Xception: Deep Learning with Depthwise Separable Convolutions“, *arXiv:1610.02357 [cs]*, 4. April 2017,

5.2 Untersuchte Konfigurationen

Da AutoKeras die Suche nach einer guten Netzwerk-Architektur übernimmt, ist es leicht möglich, verschiedene Repräsentationen des Spielfelds auf ihre Lernbarkeit zu untersuchen. Außerdem kann so verglichen werden, ob es einen Vorteil bringt, die Zielwerte, die das Netzwerk lernen soll, zu normalisieren. Es werden drei Repräsentationen des Spielfelds verglichen:

1. Die **naive Repräsentation**: Das Spielfeld wird, so wie es in der `Observation` dargestellt ist (siehe Kapitel 3.1), vom Netzwerk gelernt. Die Steine der Spieler sind mit 1 respektive 2 kodiert, ein freies Feld ist eine 0.

2. Die **reguläre Repräsentation**: Sie unterscheidet sich von der naiven Repräsentation dadurch, dass die Steine der Spieler mit 1 und -1 anstatt 1 und 2 kodiert sind.

3. Die **mehrdimensionale Repräsentation**: Um die zweidimensionalität der Filter in einem ConvNet auszunutzen, muss das Spielfeld mehrdimensional repräsentiert werden. Die Repräsentation besteht aus drei Kanälen. Jeder Kanal hat die selbe Größe wie das Spielfeld, also sechs Zeilen und sieben Spalten. Es gibt einen Kanal pro Spieler in dem die Spielsteine dieses Spielers durch eine 1 kodiert sind. Der dritte Kanal ist mit 0 gefüllt, wenn Spieler 1 zuletzt gezogen hat und sonst mit 1.

Zu jedem Spielzustand muss das Netzwerk lernen, ob er für den Besitzer des Zustands gut oder schlecht ist, also zu einem Sieg oder einer Niederlage geführt hat. Diese Bewertung lässt sich, wie im Minimax-Baum, durch eine 1 bzw. -1 darstellen. Alternativ kann die Bewertung aber auch auf den Bereich $[0, 1]$ normalisiert werden. Für jede Spielfeldrepräsentation wurde ein Netzwerk mit Minimax-Spielwerten und ein Netzwerk mit normalisierten Spielwerten trainiert.

5.3 Trainingsdaten

Die Trainingsdaten bestehen aus 400 000 Spielzuständen mit der dazugehörigen Bewertung, ob dieser Zustand für den Besitzer zu einem Sieg oder einer Niederlage geführt hat. Die Zustände, die das leere Spielfeld repräsentieren, haben eine neutrale Bewertung. Die Spielzustände wurden generiert, indem zwei MCTS-Agenten 16 000 Spiele gegeneinander gespielt haben, dabei wurden 267 022 einzigartige Zustände erreicht.

Der Datensatz wurde danach augmentiert, indem jeder Spielzustand horizontal gespiegelt wurde. Da ein Zustand und seine Spiegelung in VierGewinnt äquivalent sind, kann so der Datensatz deutlich vergrößert werden. Nach der Augmentierung besteht der Datensatz aus 509 066 einzigartigen Zuständen. Doppelt vorhandene Zustände wurden auf einen einzigen reduziert, indem der Durchschnitt ihrer Bewertung gebildet wurde. Fünf Pro-

zent der gesammelten Trainingsdaten werden während des Trainings für die Validierung benutzt. Damit wird die Genauigkeit des Netzwerks auf unbekannten Daten gemessen. Die Entfernung von Duplikaten war notwendig, um zu verhindern, dass Zustände aus den Trainingsdaten in den Validierungsdaten auftauchen.

Aus den 509 066 einzigartigen Zuständen wurde dann ein zufälliger Ausschnitt mit 400 000 Zuständen für das Training verwendet.

5.4 Trainingsergebnisse

AutoKeras erstellt und trainiert viele verschiedene Netzwerke auf der Suche nach einer optimalen Konfiguration. Um die Trainingszeit im Rahmen zu halten, wurde AutoKeras auf zehn Netzwerke pro Repräsentation begrenzt. Die besten Netzwerke für jede Repräsentation werden in diesem Kapitel verglichen.

Beim Training werden zwei Metriken beobachtet. Die mittlere quadratische Abweichung (MSE) ist die Fehlerfunktion des Netzwerks und wird während des Trainings optimiert. Die mittlere Abweichung im Betrag (MAE) ist ein verständliches Maß dafür, wie stark einzelne Vorhersagen daneben liegen.

Es hat sich in einem verkürzten Trainingslauf mit nur 20 000 Beispielen gezeigt, dass die naive Repräsentation nicht für das Lernen geeignet ist. Sie erreicht nur einen MSE von 0.887 und einen MAE von 0.886 und zeigt auch keine Anzeichen einer Verbesserung über 100 Epochen (siehe Abb. 18).

Zum Vergleich: durch einfaches Raten kann bereits ein MSE von ca. 0.896 und MAE von 0.9 erreicht werden. Die anderen beiden Repräsentationen zeigen, dass ein Potential zum Lernen vorhanden ist.

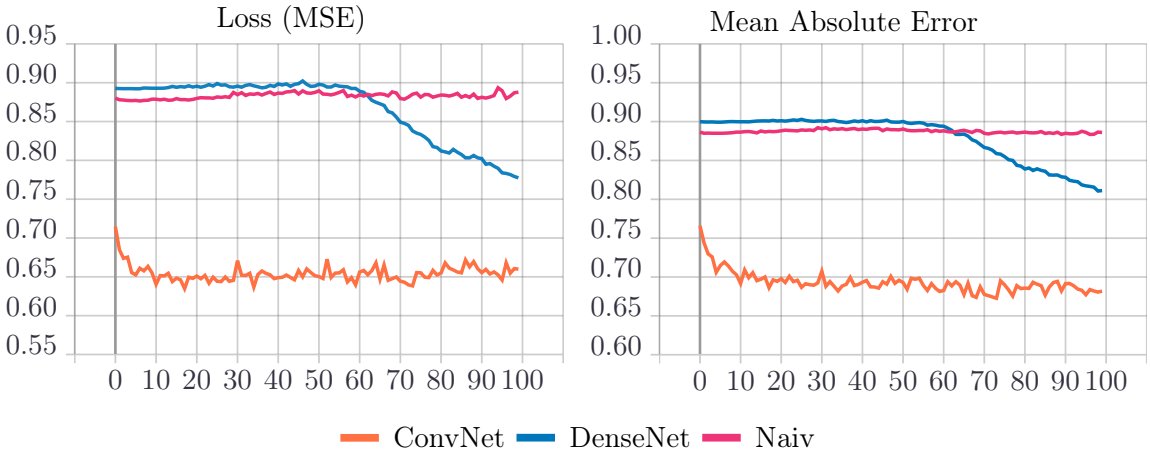


Abbildung 18: Trainingsergebnisse eines Testlaufs mit 20 000 Beispielen ohne Normalisierung

Im Folgenden wird das Netzwerk, das die reguläre Repräsentation lernt, als DenseNet und

das Netzwerk, das die mehrdimensionalen Repräsentation lernt, als ConvNet bezeichnet.

Beide Netzwerk-Arten werden auf den gesamten 400 000 Trainingsbeispielen trainiert. Die Entwicklung des Fehlers und des MAE ist in Abb. 19 mit Normalisierung und in Abb. 20 ohne Normalisierung der Zielwerte zu sehen.

Mit normalisierten Zielwerten erreichen beide Netzwerke einen sehr geringen Fehler. Obwohl das DenseNet einen geringeren Fehler erreicht, hat das ConvNet einen geringeren MAE. Der Fehler eines einzelnen Trainingsbeispiels liegt dabei im Bereich $0 \leq (y - \hat{y})^2 \leq 0.25$ während der MAE im Bereich $0 \leq |y - \hat{y}| \leq 0.5$ liegt.

Eine genauere Untersuchung der Vorhersagen des DenseNet hat offenbart, dass einige der frühen Spielzustände, mit weniger als zehn Steinen auf dem Spielfeld, eine identische Bewertung durch das Netzwerk erhalten, obwohl sich ihr tatsächlicher Wert unterscheidet. Da es frühe Zustände sind, haben sie eine durchschnittliche Bewertung nahe 0, was dafür sorgt, dass der Fehler im Quadrat sehr viel geringer ausfällt als im Betrag.

Dieses Verhalten kann auf das sogenannte dying ReLU-Problem[64] hindeuten, bei dem Einheiten mit ReLU-Aktivierungsfunktion in tiefen neuronalen Netzen mit der Zeit absterben und keine Ausgabe mehr erzeugen. Durch ein erneutes Trainieren des DenseNet mit einer neuen, zufälligen Initialisierung der Gewichte lässt sich das Problem möglicherweise verhindern.

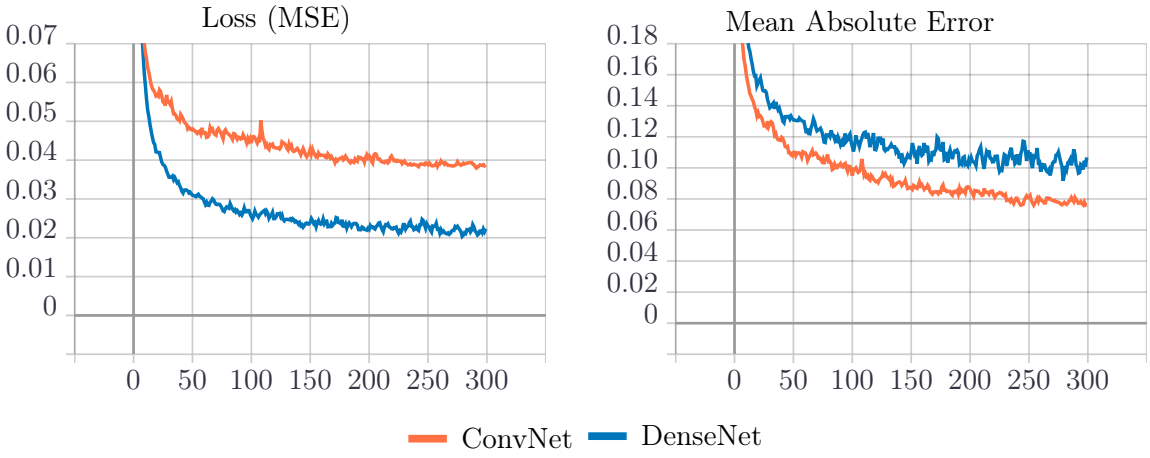


Abbildung 19: Trainingsverlauf mit Normalisierung

Ohne eine Normalisierung der Zielwerte fällt der Fehler und MAE deutlich höher aus. Dies ist zu erwarten, da der Wertebereich der Zielwerte mit $[-1, 1]$ doppelt so groß ist. Eine falsche Vorhersage kann somit einen Fehler $0 \leq (y - \hat{y})^2 \leq 4$ und einen MAE im Bereich $0 \leq |y - \hat{y}| \leq 2$ verursachen. Generell scheint es schwerer zu sein, die Repräsentation ohne Normalisierung zu lernen. Beide Netzwerke haben nach 50 Epochen ein Plateau erreicht und verbessern sich nur noch geringfügig, das ConvNet ist dabei dem DenseNet klar

64. Lu Lu u. a., „Dying ReLU and Initialization: Theory and Numerical Examples“, *arXiv:1903.06733 [cs, math, stat]*, 21. Oktober 2020,

überlegen.

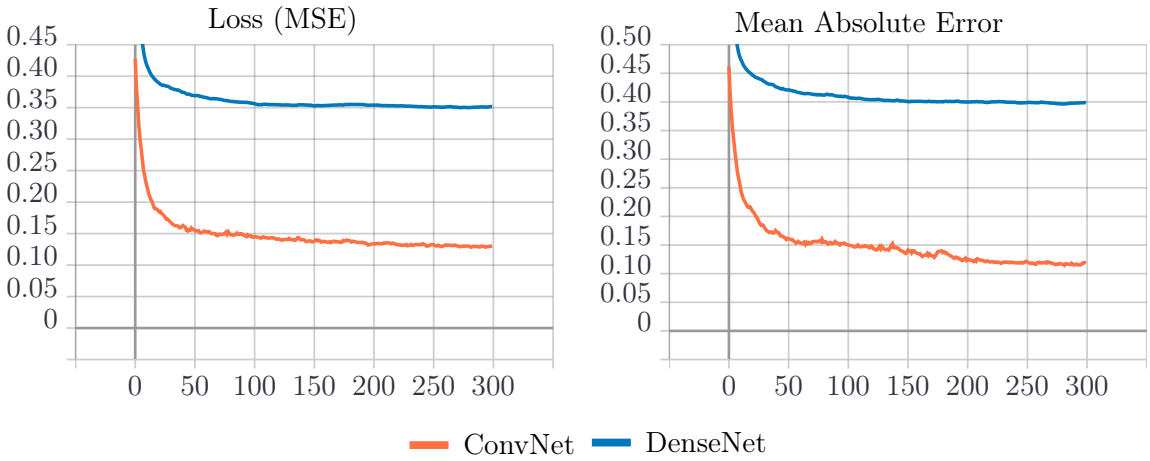


Abbildung 20: Trainingsverlauf ohne Normalisierung

In Tabelle 1 sind die trainierten Netzwerke einander direkt gegenüber gestellt. Das ConvNet erreicht sowohl mit Normalisierung als auch ohne einen sehr guten MAE im Vergleich zum DenseNet, es scheint also genauere Vorhersagen machen zu können. Im folgenden Kapitel werden die Netzwerke mit der Baumsuche kombiniert und verglichen, wie stark sich der Agent mit den einzelnen Netzwerken gegenüber der normalen Baumsuche verbessert.

	Normalisiert		Minimax	
	DenseNet	ConvNet	DenseNet	ConvNet
MSE	0.022	0.039	0.351	0.130
MAE	0.107	0.076	0.398	0.116

Tabelle 1: MSE und MAE der besten Netzwerke

5.5 Integration des neuronalen Netzes in die Baumsuche

Die Kombination des neuronalen Netzwerks mit der Monte-Carlo-Baumsuche ist trivial. Das Netzwerk augmentiert die Evaluation der Spielposition, ersetzt sie aber nicht komplett. Es wird weiterhin eine vollständige Simulation aus dem zu bewertenden Knoten durchgeführt, das Ergebnis dieser Simulation r wird mit der Vorhersage des Netzwerks r_{pred} mit einem Gewichtungsfaktor α_{net} kombiniert.

Der `Evaluator` erzeugt also eine Bewertung

$$r^* = \alpha_{net} r_{pred} + (1 - \alpha_{net}) r \tag{14}$$

Um die Vorhersage des Netzwerks stabiler zu machen, wird die Symmetrieeigenschaft des Spielbretts ausgenutzt. Anstatt nur eine einzelne Spielposition durch das Netzwerk zu

bewerten, wird zusätzlich die Spiegelung bewertet und der Durchschnitt dieser beiden Bewertungen als r_{pred} verwendet.

Die Vorhersage durch das Netzwerk ist im Vergleich zur Simulation zufälliger Spiele sehr langsam. In einem ersten Vergleich erhält der Agent mit Netzwerk deshalb nur 100 Iterationen Bedenkzeit während der Vergleichs-Agent ohne Netzwerk 400 Iterationen zur Verfügung hat. In Tabelle 2 werden alle vier Netzwerke mit den Gewichtungsfaktoren $\alpha_{net} = 0.5$ und $\alpha_{net} = 0.85$ verglichen und ihre Laufzeit gemessen.

	Gewicht α_{net}		
	0.5	0.85	Zeit (s)
DenseNet	0.300	0.169	317
DenseNet$_{Norm}$	0.468	0.222	433
ConvNet	0.467	0.201	662
ConvNet$_{Norm}$	0.541	0.342	919

Tabelle 2: Gewinnchance des MCTS-Agenten mit Neuronalem Netz bei 100 Iterationen pro Zug gegen einen normalen MCTS-Agenten mit 400 Iterationen pro Zug über 400 Spiele. Gemessen wurde die Laufzeit über 800 Spiele pro Netzwerk.

Eine hohe Gewinnchance bei hohem Gewichtungsfaktor deutet darauf hin, dass das Netzwerk allein bereits eine akkurate Bewertung treffen kann. Die ConvNets erzielen in diesem Vergleich die besten Ergebnisse. Das ConvNet mit Normalisierung kann sogar, mit einem Gewichtungsfaktor $\alpha_{net} = 0.5$, den normalen MCTS-Agenten in mehr als der Hälfte der Spiele schlagen, obwohl der normale Agent vier Mal so viele Iterationen durchlaufen kann; es benötigt allerdings mehr als doppelt so viel Zeit wie das DenseNet mit Normalisierung. Die Netzwerke ohne Normalisierung schneiden beide schlechter ab. In Kapitel 6.7 werden nur noch die Netzwerke mit Normalisierung verglichen.

Während der MCTS-Agent ohne Netzwerk 7 000 Iterationen pro Sekunde ausführen kann, verlangsamt er sich mit neuronalem Netz auf 909 Iterationen pro Sekunde mit dem DenseNet und nur 386 Iterationen pro Sekunde mit dem ConvNet. Die Netzwerke sind also in etwa 8 bis 18 Mal langsamer als die normale Baumsuche.[65]

Die reduzierte Geschwindigkeit durch das neuronale Netz wird beim Vergleich der Agenten im folgenden Kapitel berücksichtigt.

65. Durch eine Parallelisierung der Baumsuche kann die Geschwindigkeit des Agenten möglicherweise verbessert werden, siehe Browne u. a., „A Survey of Monte Carlo Tree Search Methods“, S. 24 ff.

6 Vergleiche und Ergebnisse

Die zuvor in den Kapiteln 3 und 5.5 entwickelten Agenten werden in diesem Kapitel ausführlich miteinander verglichen.

6.1 Methodik

Die Auswirkung der Verbesserungen auf die Spielstärke wird ermittelt, indem Spiele gegen einen MCTS-Agenten ohne Verbesserungen gespielt werden. Sofern nicht anders angegeben, hat jeder Agent 1 000 Iterationen der Baumsuche zur Verfügung, bevor ein Zug gewählt werden muss. Der Parameter C_p des Vergleichsagenten wird in Kapitel 6.2 bestimmt.

Die unterschiedlichen Verbesserungen wirken sich unterschiedlich stark auf die Geschwindigkeit des Algorithmus aus, bei gleicher Anzahl der Iterationen haben dadurch die Verbesserungen einen Vorteil. Am Ende dieses Kapitels werden die Verbesserungen noch einmal unter Berücksichtigung ihrer Ausführungszeit verglichen, um eine realistischere Aussage über ihre Spielstärke machen zu können.

Jede Verbesserung hat eine Reihe von Parametern, die optimiert werden müssen. Dafür wurden mindestens 500 Spiele mit jeder Konfiguration gegen den MCTS-Agenten, abwechselnd als erster und als zweiter Spieler, durchgeführt. Die durchschnittliche Gewinnchance aus diesen Spielen ist in den jeweiligen Tabellen festgehalten. Eine Niederlage zählt dabei zu 50% (haben zum Beispiel von 10 Spielen alle 10 zu einem Unentschieden geführt, wird die Gewinnchance mit 50% angegeben). Auch wenn 500 Spiele in vielen Fällen ausreichend sind, um sich ein grobes Bild von der Stärke einer Verbesserung zu machen, so unterliegen die in diesem Kapitel vorgestellten Ergebnisse dennoch großen Schwankungen.

Der von jedem Agent erstellte Baum wird nicht zwischen den Zügen beibehalten.

Der Vergleich wird auf einem AMD Ryzen 5 3600 Prozessor mit 6 Kernen/12 Threads @3600MHz mit 16GB RAM und einer NVIDIA GeForce GTX 960 mit 4GB VRAM durchgeführt. Da die Agenten selbst nicht parallelisiert sind, kann die Simulation der Spiele parallel durchgeführt werden.

6.2 MCTS

Der optimale Wert des Parameters C_p der Baumsuche muss für jeden Algorithmus individuell bestimmt werden. Um den idealen Wert für den Agenten ohne Verbesserungen zu finden, spielt der Agent mit verschiedenen Werten $C_p \in \{0.6, 0.7, .., 1.4\}$ gegen einen Agent mit $C_p = 1.0$. Siehe dazu Tabelle 3.

Der optimale Wert liegt in etwa bei 0.8 bis 0.9. In den weiteren Vergleichen wird $C_p = 0.8$

C_p	0.6	0.7	0.8	0.9	1.0	1.1	1.2	1.3	1.4
Sieg %	42.5	43.5	50.6	50.0	49.4	49.5	48.7	48.0	46.6

Tabelle 3: Gewinnchance des MCTS-Agenten für verschiedene C_p gegen einen MCTS-Agenten mit $C_p = 1.0$, $900 - 1\,700$ Spiele pro Parameter.

für den Vergleichsagenten verwendet.

Zusätzlich zur Bestimmung des Parameters C_p wurde die Ausführungszeit des Agenten gemessen. Der MCTS-Agent benötigt für 20 000 Iterationen 2.857 Sekunden was in etwa 7 000 Iterationen pro Sekunde entspricht.

6.3 Transpositionen

Die Erweiterung um Transpositionen fügt zwei weitere Parameter hinzu. Zum einen müssen die verschiedenen Auswahlregeln UCT1, UCT2 und UCT3 miteinander verglichen werden und zum anderen muss untersucht werden, ob die Ausnutzung der Symmetrieeigenschaft des Spiels einen Vorteil für die Transpositions-Erkennung bringt. In Tabelle 4 werden die Auswahlregeln, oben mit Berücksichtigung der Symmetrie und unten ohne, verglichen.

Mit Symmetrie						
C_p	0.7	0.8	0.9	1.0	1.1	1.2
UCT1	45.3	**53.0**	46.2	49.6	50.0	50.0
UCT2	46.2	51.4	51.5	**52.1**	**52.1**	49.3
UCT3	47.0	49.5	**50.8**	47.0	49.0	48.2

Ohne Symmetrie						
C_p	0.7	0.8	0.9	1.0	1.1	1.2
UCT1	47.4	**51.0**	49.4	49.6	48.4	47.2
UCT2	44.6	49.5	46.8	47.4	**49.8**	45.3
UCT3	49.3	48.2	45.5	50.0	48.4	**50.9**

Tabelle 4: Gewinnchance gegen den MCTS-Agent bei 1 000 Iterationen pro Zug, $500 - 800$ Spiele pro Parameter.

Auch mit Transpositionen befindet sich der optimale Wert für C_p nahe 1.0, die einzelnen Auswahlmethoden haben aber jeweils leicht unterschiedliche C_p-Werte mit denen sie besser funktionieren. Generell bringt es einen deutlichen Vorteil, die Symmetrie des Spielfeldes auszunutzen. Lediglich mit der UCT3-Auswahlregel verbessert sich die Gewinnchance dadurch nicht signifikant.

Der zusätzliche Aufwand, der durch das Verarbeiten der Transpositionstabelle entsteht, verlangsamt den Algorithmus deutlich. Die Auswahlmethoden UCT1 und UCT2 schaffen

so nur noch 5 952 Iterationen pro Sekunde und der Agent mit UCT3 verlangsamt sich sogar auf 4 866 Iterationen pro Sekunde. Damit ist die Verbesserung mit Transpositionen 15-30% langsamer.

Bei gleicher Anzahl Iterationen ist der Agent mit Transpositionen nur geringfügig besser als der normale MCTS-Agent.

6.4 AMAF

Die beiden AMAF-Varianten α-AMAF und RAVE werden separat untersucht. Da sich die AMAF-Statistik eines Knotens sehr schnell ändert, sorgt sie indirekt dafür, dass der Baum gleichmäßiger durchsucht wird. Dadurch fällt der optimale Wert für C_p deutlich geringer aus. In Tabelle 5 werden Werte für $\alpha \in \{0.3, 0.5, 0.8\}$ betrachtet.

	C_p	0.2	0.3	0.4	0.5	0.6
α	0.3	28.2	35.5	52.5	53.3	60.0
	0.5	43.9	58.5	**62.4**	62.1	57.6
	0.8	61.5	55.8	51.8	53.0	51.6

Tabelle 5: α-AMAF

Die α-AMAF-Verbesserung bewirkt eine große Steigerung der Spielstärke. Mit $\alpha = 0.5$ und $C_p = 0.5$ gewinnt der Agent 62.4% der Spiele gegen den normalen MCTS-Agenten. Der Parameter α kann möglicherweise noch weiter optimiert werden, gleichzeitig muss dabei allerdings auch C_p angepasst werden. Wie in Tabelle 5 zu sehen ist, wird der optimale Wert von C_p kleiner, wenn α steigt und er wird größer, wenn α sinkt.

Die RAVE-Variante hat an Stelle des Parameters α den Parameter k. k bestimmt, ab wie vielen Besuchen eines Knotens die Gewichtung $\beta = 0.5$ ist und somit die AMAF-Statistik das gleiche Gewicht in der Auswahl des Kindknotens hat, wie die Statistik $Q(v)$ (siehe Gleichung (7) auf Seite 20). Der ideale Wert dieses Parameters hängt stark von der Anzahl der Iterationen ab. Wenn nur wenige Iterationen durchgeführt werden, sorgt ein hoher Wert k dafür, dass fast ausschließlich die AMAF-Statistik in der TreePolicy verwendet wird. Wird k dagegen, im Verhältnis zur Anzahl der Iterationen, sehr gering gewählt, wird die AMAF-Statistik nur für wenige Besuche des Knotens benutzt, bevor $Q(v)$ verwendet wird.

Nach 1 000 Iterationen der Monte-Carlo-Baumsuche wurde der beste Kindknoten in der Wurzel in etwa 500 Mal besucht. Deshalb werden in Tabelle 6 Werte von $k \in \{10, 20, 50, 100, 200\}$ untersucht.

Ein Wert von $k = 100$ bei $C_p = 0.2$ hat die größte Gewinnchance gegen den Vergleichsagent. Der RAVE-Spieler kann damit 61.6% der Spiele gewinnen. Die RAVE-Verbesserung ist

	C_p	0.2	0.3	0.4	0.5	0.6
k	10	52.1	57.3	55.7	57.6	57.0
	20	56.8	60.9	57.7	57.0	56.0
	50	61.0	60.6	59.4	56.2	55.5
	100	**61.6**	59.1	57.6	51.7	52.9
	200	59.6	59.1	53.7	50.3	45.6

Tabelle 6: RAVE

damit dem α-AMAF-Spieler leicht unterlegen.

Beide AMAF-Varianten benötigen in etwa 3.53 Sekunden für 20 000 Iterationen und schaffen somit ca. 5 665 Iterationen pro Sekunde. Sie sind damit in rund 20% langsamer als der normale MCTS-Agent.

6.5 Last-Good-Reply

Da der Last-Good-Reply-Agent lediglich die Simulationsphase verändert, liegt der optimale Wert für C_p wie beim MCTS-Agenten nahe 1.0. In Tabelle 7 wird der LGR-Agent mit allen Kombinationen aus *Vergessen von falschen Antworten* (F mit Vergessen bzw. $\overline{F}$ ohne Vergessen) und *Behalten von guten Antworten zwischen Zügen* (K mit Behalten bzw. $\overline{K}$ ohne Behalten) getestet.

	C_p	0.8	0.9	1.0	1.1	1.2
Variante	$\overline{K}/\overline{F}$	72.9	74.0	**75.1**	70.3	74.3
	$K/\overline{F}$	73.2	75.6	**77.8**	73.9	74.6
	$\overline{K}/F$	52.6	52.7	**55.0**	53.6	52.5
	K/F	51.1	52.4	55.4	**56.9**	51.4

Tabelle 7: Gewinnchance des LGR-Agenten über 500 Spiele.
K=behalte Antworten zwischen Zügen, F=vergesse falsche Antworten

Ohne Vergessen und ohne Behalten von Antworten kann der LGR-Agent 75.1% der Spiele gegen den Vergleichsspieler gewinnen. Wenn die gelernten Antworten zwischen den Zügen erhalten bleiben, steigt die Gewinnchance sogar auf 77.8%. Im Gegensatz zum Behalten wirkt sich das Vergessen von falschen Antworten sehr negativ auf die Spielstärke aus. Dadurch sinkt die durchschnittliche Gewinnchance von ca. 73% auf nur noch 53%.

Diese Steigerung der Spielstärke kommt aber mit starken Verlangsamung des Algorithmus. Der LGR-Agent benötigt in etwa 4.37 Sekunden für 20 000 Iterationen der Baumsuche und schafft damit 4 579 Iterationen pro Sekunde. Er ist somit ca. 35% langsamer als der normale MCTS-Agent.

6.6 AMAF-LGR

Sowohl AMAF als auch LGR haben eine große Verbesserung der Spielstärke bewirkt. Der AMAF-LGR-Agent kombiniert beide Verbesserungen. Die LGR-Komponente wird nur ohne Vergessen und mit Behalten der Antworten betrachtet, da diese Konfiguration die besten Ergebnisse in Kapitel 6.5 geliefert hat. Wie zuvor für die AMAF-Verbesserung werden verschiedene Werte für den Parameter α und den RAVE-Parameter k betrachtet und nach einem optimalen Wert für C_p gesucht.

Die Ergebnisse in Tabelle 8 zeigen, dass sich die LGR-Verbesserung neutral auf den idealen C_p-Wert auswirkt. Die Kombination beider Verbesserungen hat also optimale Werte für C_p im selben Bereich wie zuvor der AMAF-Agent in Kapitel 6.4.

	C_p	0.2	0.3	0.4	0.5	0.6
α	0.3	56.8	70.6	77.9	78.9	80.3
	0.5	77.3	80.2	**83.4**	81.9	78.6
	0.8	81.3	82.4	78.6	76.6	73.4

Tabelle 8: LGR mit α-AMAF

Durch die Kombination von LGR mit α-AMAF kann die Gewinnchance des LGR-Agenten von 77.8% auf 83.4% gesteigert werden. In Verbindung mit der RAVE-Verbesserung kann der LGR-Agent seine Gewinnchance, wie in Tabelle 9 zu sehen ist, sogar auf 87.4% steigern. Wenn RAVE mit LGR kombiniert wird, ist der Agent also stärker als mit AMAF und LGR, einzeln betrachtet ist RAVE aber AMAF unterlegen.

	C_p	0.2	0.3	0.4	0.5	0.6
k	10	80.7	81.8	85.1	81.7	80.3
	20	81.5	82.1	80.4	81.8	81.2
	50	**87.4**	81.8	83.2	81.0	77.3
	100	83.1	81.3	78.3	80.1	79.4
	200	81.3	79.5	77.9	71.7	78.0

Tabelle 9: LGR mit RAVE

Zusätzlich wurden beide Kombinationen mit optimierten Parametern mit dem besten LGR-Agenten verglichen (siehe Tabelle 10). In 1 000 Spielen konnte der RAVE-LGR-Agent 60.3% gegen LGR gewinnen während der AMAF-LGR-Agent 55.6% der Spiele gewonnen hat.

AMAF-LGR	RAVE-LGR
55.6	60.3

Tabelle 10: Gewinnchance von AMAF-LGR und RAVE-LGR gegen LGR über 1 000 Spiele

Dadurch, dass sowohl AMAF als auch LGR deutliche Geschwindigkeitseinbußen haben, ist

der kombinierte Agent langsam. Er schafft nur noch 3 947 Iterationen pro Sekunde und ist somit 44% langsamer als der normale MCTS-Agent.

6.7 Netzwerke

Nachdem die einzelnen Verbesserungen des MCTS-Algorithmus untersucht wurden, wird in diesem Kapitel die Kombination der MCTS-Agenten mit neuronalen Netzen genauer betrachtet.

Die Vorhersage der Netzwerke ist sehr langsam, dadurch schaffen die Agenten mit Netzwerk nur noch einige Hundert Iterationen pro Sekunde im Vergleich zu den bis zu 7 000 Iterationen pro Sekunde der anderen Agenten. Tabelle 11 zeigt die Geschwindigkeit der Agenten mit Netzwerk in Iterationen pro Sekunde.

Agent	It./Sekunde
DenseNet	909
ConvNet	386
LGR-DenseNet	833
LGR-ConvNet	377
RAVE-LGR-DenseNet	772
RAVE-LGR-ConvNet	364

Tabelle 11: Iterationen pro Sekunde der Agenten mit Netzwerk

Unter der Annahme, dass sich die Kombination mit einem neuronalen Netz genauso wenig auf die ideale Parametrisierung der Agenten auswirkt, wie die LGR-Verbesserung, werden die optimalen Parameter der Agenten mit Netzwerk nicht erneut bestimmt. In allen nachfolgenden Vergleichen werden die Parameter, die in den Kapiteln 6.2 bis 6.6 bestimmt wurden, benutzt. Die AMAF-LGR-Agenten benutzen die RAVE-Variante mit einem RAVE-Parameter $k = 50$.

Die bisherigen Vergleiche wurden mit 1 000 Iterationen pro Zug für beide Agenten durchgeführt. Da die Agenten mit Netzwerk aber im schlimmsten Fall nur 364 Iterationen pro Sekunde ausführen können, hätten sie somit bis zu 3 Sekunden Rechenzeit pro Zug zur Verfügung. Damit hätten diese Agenten einen gewaltigen Vorteil gegen den Vergleichsagent, welcher nur ca. 0.143 Sekunden für seine 1 000 Iterationen benötigt.

Um die Vergleiche fairer zu gestalten, erhalten die Agenten mit Netzerk in etwa die selbe Rechenzeit wie der Vergleichsagent. Dies bedeutet aber auch, dass sie beim Vergleich mit 1 000 Iterationen gerade einmal 50 bis 130 Iterationen durchführen können. Die Netzwerk-Agenten werden in Tabelle 12 mit vier verschiedenen Laufzeiten, welche 1 000, 2 000, 3 000 bzw. 4 000 Iterationen des Vergleichsagenten entsprechen, verglichen. Der Gewichtungsfaktor des Netzwerks α_{net} ist auf 0.5 festgelegt und wird nicht weiter optimiert.

Relative Iterationen	1 000	2 000	3 000	4 000
DenseNet	40.2	49.0	60.5	62.7
ConvNet	17.0	39.0	44.3	53.8
LGR-DenseNet	45.2	63.5	70.8	77.5
LGR-ConvNet	23.5	53.5	62.2	63.7
RAVE-LGR-DenseNet	65.5	76.4	77.9	84.9
RAVE-LGR-ConvNet	38.8	63.6	76.5	78.2

Tabelle 12: Gewinnchance der Agenten mit neuronalem Netz über jeweils 300 Spiele. Die Netzwerk-Agenten haben die selbe Rechenzeit zur Verfügung, wie der Vergleichsagent für N Iterationen benötigt.

Bereits beim Vergleich mit 1 000 Iterationen ist zu erkennen, dass die Agenten mit DenseNet einen Vorteil gegenüber dem ConvNet haben. Die höhere Geschwindigkeit des einfachen Netzwerks wirkt sich positiv auf die Spielstärke aus, da der Agent mehr Iterationen durchführen kann. Die Kombination mit LGR sowie AMAF und LGR sorgt für eine große Verbesserung der Spielstärke. Ab 2 000 Iterationen reduziert sich der Vorteil des einfachen DenseNet und die bessere Genauigkeit des ConvNet zeigt sich deutlich im Anstieg der Spielstärke, dennoch bleibt die Spielstärke des Agenten mit DenseNet im Durchschnitt höher.

6.8 Gegenüberstellung

Für einen realistischeren Vergleich der Verbesserungen wird in diesem Kapitel ein Test unter echten Bedingungen durchgeführt. Jeder Agent verwendet optimierte Parameter und die Agenten behalten ihren erstellten Spielbaum zwischen den Zügen bei. Außerdem hat nun nicht mehr jeder Agent volle 1 000 Iterationen Bedenkzeit, sondern so viel Rechenzeit wie der Vergleichsagent für 1 000 bis 4 000 Iterationen benötigt. Dadurch haben Verbesserungen, die die Geschwindigkeit stark reduzieren, keinen Vorteil mehr. Tabelle 13 stellt alle einzeln bewerteten Agenten in einer Tabelle dar und gibt die durchschnittliche Geschwindigkeit des Algorithmus in Iterationen pro Sekunde an.

		Relative Iterationen			
Agent	It./s	1 000	2 000	3 000	4 000
UCT1	5 952	47.9	47.7	50.5	46.8
UCT2	5 952	47.5	44.9	44.9	45.0
UCT3	4 866	43.7	39.8	42.4	40.9
α-AMAF	5 665	59.2	60.3	60.8	55.1
RAVE	5 665	61.7	57.6	59.9	60.2
LGR	4 578	68.9	73.9	76.9	79.6
AMAF-LGR	3 947	76.2	77.7	78.2	82.8
RAVE-LGR	3 947	76.0	82.7	83.4	81.4
DenseNet	909	40.2	49.0	60.5	62.7
ConvNet	386	17.0	39.0	44.3	53.8
LGR-DenseNet	833	45.2	63.5	70.8	77.5
LGR-ConvNet	377	23.5	53.5	62.2	63.7
RAVE-LGR-DenseNet	772	65.5	76.4	77.9	84.9
RAVE-LGR-ConvNet	365	38.8	63.6	76.5	78.2

Tabelle 13: Gegenüberstellung aller Agenten, Gewinnchance über jeweils 500 Spiele.

Durch die reduzierte Ausführungszeit schneiden alle Verbesserungen etwas schlechter ab, als im Vergleich mit 1 000 Iterationen, die Verbesserungen scheinen aber von der zusätzlichen Rechenzeit stärker zu profitieren als der Vergleichsagent. Insbesondere die Agenten mit LGR-Verbesserung erhöhen ihre Spielstärke mit steigender Rechenzeit deutlich. Transpositionen wirken sich dagegen insgesamt eher nachteilig auf die Spielstärke des Agenten aus, die gewonnene Spielstärke reicht nicht aus, um die verlorene Rechenzeit auszugleichen.

Die Netzwerk-Agenten erreichen trotz ihrer wenigen Iterationen eine sehr gute Gewinnchance. Die Spielstärke der Agenten mit Netzwerk ist vergleichbar mit denen ohne Netzwerk, bei einem Bruchteil der Durchläufe der Baumsuche.

7 Fazit und Ausblick

Die Monte-Carlo-Baumsuche ist ein sehr flexibler Algorithmus, der gut für das Spielen von VierGewinnt geeignet ist. Die in dieser Arbeit entwickelten Agenten konnten im Kaggle-Wettbewerb konsistent einen Platz unter den 20 besten Agenten von mehr als 400 Teilnehmern erreichen. Die reine Monte-Carlo-Baumsuche hat bereits gute Ergebnisse im Wettbewerb erzielt, durch die vorgestellten Verbesserungen konnte die Spielstärke auch im Wettbewerb gesteigert werden und somit mit der Konkurrenz mithalten.

Die Erkennung von Transpositionen hat leider nicht den erhofften Erfolg erzielt. In ihrem Paper „UCD: Upper confidence bounds for rooted directed acyclic graphs“[66] haben Cazenave u.a. einen parametrisierbaren Ansatz für die Behandlung von Transpositionen vorgestellt, mit dem sie mit vielen Parametrisierungen eine Gewinnchance über 60% und mit optimalen Parametern sogar eine Gewinnchance bis zu 70.9% erzielen. Der in dieser Arbeit verwendete Ansatz, wie er von Childs u.a. vorgestellt wurde,[67] konnte leider keine signifikante Verbesserung der Spielstärke durch Transpositionen feststellen.

Es kann nicht ausgeschlossen werden, dass die Implementierung der Transpositionsbehandlung in dieser Arbeit fehlerhaft ist; es ist auch möglich, dass die Berücksichtigung von Transpositionen erst einen Vorteil bringt, wenn sehr viele Iterationen pro Spielzug ausgeführt werden. Da in dieser Arbeit u.a. ein Agent für den Kaggle-Wettbewerb entwickelt wurde, haben gewisse Einschränkungen des Wettbewerbs die Entwicklung des Agenten beeinflusst. Die genaue Geschwindigkeit der im Wettbewerb verwendeten Maschinen war nicht bekannt und die Ausführung war auf wenige Sekunden pro Zug beschränkt. Dadurch hat sich diese Arbeit besonders auf eine geringe Anzahl an Iterationen pro Zug fokussiert.

All Moves as First und Rapid Action Value Estimation haben beide zu einer Verbesserung der Spielstärke geführt. RAVE wird zwar in vielen Veröffentlichungen als signifikant besser als α-AMAF dargestellt, in dieser Arbeit waren α-AMAF und RAVE allerdings gleichauf. Statt der Formel für den RAVE-Parameter β von Gelly und Silver[68] kann auch die von Helmbold und Parker-Wood[69] verwendet werden. Eine andere Formel könnte zu besseren Ergebnissen führen.

Die Stärke der Last-Good-Reply-Verbesserung war überraschend. LGR ist eine sehr effiziente Methode, um die Simulationsphase zu verbessern. Andere Ansätze versuchen externes Wissen über das Spiel in der DefaultPolicy zu verwenden. Sie vermeiden Züge, die zu einer sofortigen Niederlage führen würden oder spielen Züge sofort, wenn dadurch das Spiel sofort gewonnen wird. Dadurch steigt aber der Rechenaufwand für jede Simulation signifikant. Durch LGR kann ein ähnlicher Effekt erzielt werden, ohne dass eine Strategie für

66. Cazenave, Méhat und Saffidine, „UCD“.
67. Childs, Brodeur und Kocsis, „Transpositions and Move Groups in Monte Carlo Tree Search“.
68. Siehe Gelly und Silver, „Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go“, S. 1866.
69. Siehe Helmbold und Parker-Wood, „All-Moves-As-First Heuristics in Monte-Carlo Go“, S. 3.

das Spiel bekannt sein muss. Die von Baier vorgeschlagene Erweiterung durch Vergessen von falschen Antworten[70] hatte leider einen negativen Effekt auf die Spielstärke. Dies ist vermutlich dem sehr geringen Aktionsraum von VierGewinnt geschuldet. In jedem Zug kann nur eine von bis zu sieben Aktionen gewählt werden, falsche Antworten würden also nicht sehr lange beibehalten werden. Für Probleme mit mehr möglichen Aktionen in jedem Schritt könnte LGR mit Vergessen aber besser sein als die reine LGR-Verbesserung.

Die Agenten mit neuronalem Netz haben ebenfalls ein sehr großes Potential gezeigt, die Geschwindigkeit des Netzwerks ist hier allerdings ein limitierender Faktor. Durch Parallelisierung der Baumsuche könnte die Rechenleistung moderner Grafikkarten besser ausgenutzt werden, wodurch mehr Iterationen im selben Zeitraum möglich würden. Auch die anderen Verbesserungen würden von einer Parallelisierung profitieren, die Agenten mit neuronalem Netz aber vermutlich am meisten. Die geringe Geschwindigkeit der Agenten mit Netzwerk hat auch maßgeblich beeinflusst, wie die Netzwerke trainiert und die Daten gesammelt werden. So war es leider zeitlich nicht möglich, die Netzwerke durch reines Reinforcement Learning zu trainieren und es mussten stattdessen Trainingsdaten durch den normalen Agenten erzeugt werden.

In dieser Arbeit wurde nur eine kleine Auswahl von Verbesserungen der Monte-Carlo-Baumsuche untersucht. Es gibt noch unzählige weitere Verbesserungen, die auf ihre Effizienz im Hinblick auf VierGewinnt untersucht werden können. In zukünftigen Arbeiten könnte außerdem das Verhalten der Verbesserungen bei vielen Iterationen pro Spielzug (mehr als 10 000 Iterationen) untersucht werden, dies setzt aber die Verwendung besserer Hardware oder eine Optimierung der Implementation, Parallelisierung oder Neuimplementierung in einer schnelleren Programmiersprache als Python voraus.

70. Baier und Drake, „The Power of Forgetting“.

Literaturverzeichnis

Allen, James D. „Expert Play in Connect-Four“. Besucht am 13. August 2020. http://tromp.github.io/c4.html.

Allis, Victor. „A Knowledge-Based Approach of Connect-Four: The Game Is Solved: White Wins“, Vrije Universiteit, 1. Dezember 1988.

Auer, Peter, Nicolò Cesa-Bianchi und Paul Fischer. „Finite-Time Analysis of the Multiarmed Bandit Problem“. *Machine Learning* 47, Nr. 2 (1. Mai 2002): 235–256.

Automatic Differentiation. In *Wikipedia*. 6. Oktober 2020. Besucht am 16. November 2020. https://en.wikipedia.org/w/index.php?title=Automatic_differentiation&oldid=982160496.

Baier, Hendrik, und Peter D. Drake. „The Power of Forgetting: Improving the Last-Good-Reply Policy in Monte Carlo Go“. *IEEE Transactions on Computational Intelligence and AI in Games* 2, Nr. 4 (Dezember 2010): 303–309.

Browne, Cameron B., Edward Powley, Daniel Whitehouse, Simon M. Lucas, Peter I. Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis und Simon Colton. „A Survey of Monte Carlo Tree Search Methods“. *IEEE Transactions on Computational Intelligence and AI in Games* 4, Nr. 1 (März 2012): 1–43.

Cazenave, Tristan, Jean Méhat und Abdallah Saffidine. „UCD : Upper Confidence Bound for Rooted Directed Acyclic Graphs“. *Knowledge-Based Systems* 34 (2012): 26–33.

Childs, Benjamin E., James H. Brodeur und Levente Kocsis. „Transpositions and Move Groups in Monte Carlo Tree Search“. In *2008 IEEE Symposium On Computational Intelligence and Games*, 389–395. Dezember 2008.

Chollet, François. *Deep Learning with Python*. 1st. Shelter Island, New York: Manning Publications, 22. Dezember 2017.

———. „Xception: Deep Learning with Depthwise Separable Convolutions“. *arXiv:1610.02357 [cs]*, 4. April 2017.

„Connect X“. Besucht am 17. November 2020. https://kaggle.com/c/connectx/overview/evaluation.

Coulom, Rémi. „Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search“. In *Computers and Games*, herausgegeben von H. Jaap van den Herik, Paolo Ciancarini und H. H. L. M. Donkers, bearbeitet von David Hutchison, Takeo Kanade, Josef Kittler, Jon M. Kleinberg, Friedemann Mattern, John C. Mitchell, Moni Naor u. a., 4630:72–83. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007. https://doi.org/10.1007/978-3-540-75538-8_7.

„AlphaGo: The story so far“. Besucht am 12. August 2020. https://deepmind.com/research/case-studies/alphago-the-story-so-far.

Drake, Peter. „The Last-Good-Reply Policy for Monte-Carlo Go“. *ICGA Journal* 32, Nr. 4 (1. Dezember 2009): 221–227.

Elo-Zahl. In *Wikipedia*. 7. November 2020. Besucht am 17. November 2020. https://de.wikipedia.org/w/index.php?title=Elo-Zahl&oldid=205293641.

en:User:Gdr, Traced by User:Stannered, original by. *First Two Ply of a Game Tree for Tic-Tac-Toe*. 1. April 2007. Besucht am 14. August 2020. https://commons.wikimedia.org/w/index.php?curid=1877696.

Gelly, Sylvain, und David Silver. „Combining Online and Offline Knowledge in UCT“. In *Proceedings of the 24th International Conference on Machine Learning*, 273–280. ICML '07. Corvalis, Oregon, USA: Association for Computing Machinery, 20. Juni 2007.

———. „Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go“. *Artificial Intelligence* 175, Nr. 11 (1. Juli 2011): 1856–1875.

Gelöste Spiele. In *Wikipedia*. 19. März 2019. Besucht am 14. August 2020. https://de.wikipedia.org/w/index.php?title=Gel%C3%B6ste_Spiele&oldid=186759431.

He, Kaiming, Xiangyu Zhang, Shaoqing Ren und Jian Sun. „Deep Residual Learning for Image Recognition“, 10. Dezember 2015. arXiv: 1512.03385 [cs].

Helmbold, David P, und Aleatha Parker-Wood. „All-Moves-As-First Heuristics in Monte-Carlo Go“:6.

„ImageNet“. Besucht am 17. Oktober 2020. http://www.image-net.org/.

Jin, Haifeng, Qingquan Song und Xia Hu. „Auto-Keras: An Efficient Neural Architecture Search System“. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 1946–1956. KDD '19. New York, NY, USA: Association for Computing Machinery, 25. Juli 2019.

„Kaggle/Kaggle-Environments“. Besucht am 13. Oktober 2020. https://github.com/Kaggle/kaggle-environments.

Kocsis, Levente, und Csaba Szepesvári. „Bandit Based Monte-Carlo Planning“. In *Machine Learning: ECML 2006*, herausgegeben von Johannes Fürnkranz, Tobias Scheffer und Myra Spiliopoulou, 282–293. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer, 2006.

Kombinatorische Spieltheorie. In *Wikipedia*. 20. Dezember 2019. Besucht am 14. August 2020. https://de.wikipedia.org/w/index.php?title=Kombinatorische_Spieltheorie&oldid=195080333.

LeCun, Yann, Leon Bottou, Yoshua Bengio und Patrick Ha. „Gradient-Based Learning Applied to Document Recognition“, 1998, 46.

Lu, Lu, Yeonjong Shin, Yanhui Su und George Em Karniadakis. „Dying ReLU and Initialization: Theory and Numerical Examples“. *arXiv:1903.06733 [cs, math, stat]*, 21. Oktober 2020.

Luber, Stefan. „Was ist Kaggle?“, 6. August 2020. Besucht am 7. August 2020. https://www.bigdata-insider.de/was-ist-kaggle-a-951812/.

McCulloch, Warren S., und Walter Pitts. „A Logical Calculus of the Ideas Immanent in Nervous Activity“. *The bulletin of mathematical biophysics* 5, Nr. 4 (1. Dezember 1943): 115–133.

Monte-Carlo-Simulation. In *Wikipedia*. 23. September 2020. Besucht am 13. Oktober 2020. https://de.wikipedia.org/w/index.php?title=Monte-Carlo-Simulation&oldid=203899528.

Nielsen, Michael A. „Neural Networks and Deep Learning“, 2015. Besucht am 30. Juli 2020. http://neuralnetworksanddeeplearning.com.

Nogueira, Nuno. *Minimax Algorithm*. 4. Dezember 2006. Besucht am 14. August 2020. https://commons.wikimedia.org/w/index.php?curid=2276653.

Perceptrons *(Book)*. In *Wikipedia*. 12. Mai 2020. Besucht am 31. Juli 2020. https://en.wikipedia.org/w/index.php?title=Perceptrons_(book)&oldid=956342184.

Rosenblatt, Frank. „The Perceptron: A Probabilistic Model for Information Storage and Organization“. *Psychologial Review* 65, Nr. 6 (November 1958): 386–408.

Russell, Stuart, und Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3 edition. Upper Saddle River: Pearson, 11. Dezember 2009.

Software-Agent. In *Wikipedia*. 6. Juli 2019. Besucht am 14. August 2020. https://de.wikipedia.org/w/index.php?title=Software-Agent&oldid=190180915.

Sutton, Richard S., und Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second edition. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018.

„What Is Gradient Descent?“, 28. Oktober 2020. Besucht am 16. November 2020. https://www.ibm.com/cloud/learn/gradient-descent.

Wikipedia, Springob at German. *Deutsch: Statistische Berechnung von Pi*. 24. November 2004. Besucht am 13. Oktober 2020. https://commons.wikimedia.org/wiki/File:Pi_statistisch.png.