HAW
HAMBURG

Bachelorarbeit

Philipp Goemann

Wenn die Maschine den Menschen schlägt - AlphaZero am Beispiel von 4-Gewinnt

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Philipp Goemann

Wenn die Maschine den Menschen schlägt - AlphaZero am Beispiel von 4-Gewinnt

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr. Stefan Sarstedt

Eingereicht am: 24. September 2019

Philipp Goemann

Thema der Arbeit

Wenn die Maschine den Menschen schlägt - AlphaZero am Beispiel von 4-Gewinnt

Stichworte

Neuronale Netze, Maschinelles Lernen, Bestärkendes Lernen, Künstliche Intelligenz

Kurzzusammenfassung

Inhalt dieser Arbeit ist die Veranschaulichung von *AlphaZero*. *AlphaZero* ist das erfolgreichste Computerprogramm für komplexe Brettspiele, welche es sich mit Hilfe eines neuartigen Algorithmus selbst beibringt. Um dies zu veranschaulichen, wurde basierend auf *AlphaZero* eine eigene Implementation für das Spiel Vier-Gewinnt entwickelt. In *AlphaZero* wird eine *Monte Carlo Tree-Search* von einem neuronalen Netz geleitet. Das neuronale Netz gibt Wahrscheinlichkeiten P an, aus gegebenem Zustand die möglichen Folgezüge auszuwählen. Zusätzlich approximiert es einen Wert v für das Spielergebnis z, anstatt dieses durch ein *rollout* zu schätzen, wodurch die zeitintensive Phase des *rollouts* in der *Monte Carlo Tree-Search* entfällt. Am Ende einer Simulation der *Monte Carlo Tree-Search* in *AlphaZero* liefert diese ebenfalls Wahrscheinlichkeiten, die Folgezüge auszuwählen. Diese Wahrscheinlichkeiten, als π bezeichnet, können als eine Verbesserung der initialen Wahrscheinlichkeiten P angesehen werden. Die verbesserten Wahrscheinlichkeiten π sowie das tatsächliche Spielergebnis z werden genutzt, um v und P des neuronalen Netzes zu verbessern. Für die eigene Implementation werden Optimierungsmöglichkeiten aufgezeigt und unterschiedliche Ansätze für die Gestaltung der Architektur und Wahl der Parameter diskutiert.

Philipp Goemann

Title of Thesis

When the machine beats human - AlphaZero explained by Connect-Four

Keywords

neural networks, machine learning, reinforcement learning, Monte-Carlo Tree Search, MCTS, alphazero, alphago zero, artifical intelligence, deep learning

Abstract

The goal of this paper is to illustrate *AlphaZero*, which is the most succesful computer program for complex board games so far. It learns to play these boardgames solely by self-play, using a novel algorithm. For the purpose of demonstrating *AlphaZero*, an own implementation of Connect-Four, based on *AlphaZero*, has been developed. In *AlphaZero* a *Monte Carlo Tree-Search* is guided by a neural network. The neural network outputs probabilties, called P, for choosing the next move. Additionally, a value v for the game result z is approximated by the neural network, instead of performing a *rollout* to guess z. By doing so, the time-consuming simulation phase of *Monte Carlo Tree-Search* gets omitted. At the end of a simulation of *Monte Carlo Tree-Search* in *AlphaZero*, the performed simulation returns probabilities π, which also represent which of the following moves to choose next. π can be viewed as an improvement over the initial probabilites P. π and z are used to improve P and v of the neural network. Possible ways to improve the own implementation and different approaches to creating the neural networks, as well as choosing parameters, are discussed in this paper.

Inhaltsverzeichnis

Abbildungsverzeichnis vii

1 Einleitung 1

2 Vier-Gewinnt 4

3 Monte Carlo Tree-Search 5

3.1 Minimax 5
3.2 Alpha-Beta-Pruning 6
3.3 Monte Carlo Methoden 8
3.3.1 Monte Carlo Evaluation 8
3.3.2 Monte Carlo Tree Search 9
3.4 Monte Carlo Tree Search in AlphaZero 12

4 Neuronale Netze in AlphaZero 19

4.1 Funktionsweise neuronaler Netze 19
4.2 Gradientenverfahren 22
4.3 Unstable Gradient Problem 24
4.4 Das neuronale Netz in AlphaZero 24
4.4.1 Convolutional Layer 26
4.4.2 Residual Layer 26
4.4.3 Aktivierungsfunktionen 28
4.4.4 Value-Head 30
4.4.5 Policy-Head 30
4.4.6 Zusammenfassung 30

5 Implementation 32

5.1 Spielbrett 32
5.2 Monte Carlo Tree-Search 34
5.3 Neuronales Netz 34

5.4 Parameter 36
5.5 Bewertung der Implementation 37

6 Diskussion **38**
6.1 Fazit 41

Literaturverzeichnis **42**

Glossar **46**

Selbstständigkeitserklärung **47**

Abbildungsverzeichnis

1.1 Ablauf AlphaGo Zero 3

2.1 Mögliche Gewinnreihen bei Vier-Gewinnt 4

3.1 Beispielhafter Suchbaum des Minimax-Algorithmus 6
3.2 Beispielhafter Suchbaum des Alpha-Beta-Prunings 7
3.3 Die 4 Phasen der Monte-Carlo Tree Search 10
3.4 MCTS: Selektion in AlphaZero 15
3.5 MCTS: Selektion des nächsten Knotens in AlphaZero 15
3.6 MCTS: Expansion in AlphaZero 16
3.7 MCTS: Evaluation in AlphaZero 17
3.8 MCTS: Backpropagation in AlphaZero 17

4.1 Die verschiedenen Schichten eines neuronalen Netzes 20
4.2 Visualisierung der angestrebten Veränderung des Gradienten in einer multidimensionalen Funktion 21
4.3 Aufbau des neuronalen Netzes in AlphaZero 25
4.4 Beispielhafte Struktur eines Convolutional Neural Networks 27
4.5 Aufbau residualer Netzwerke 27
4.6 Grafische Visualisierung der Rectified Linear Unit 28
4.7 Batch-Normalization 29
4.8 Grafische Visualisierung der hypberbolischen Tangente 30

5.1 Visualisierung eines Bitboards 33

1 Einleitung

Die nachfolgende Arbeit behandelt den *AlphaZero* Algorithmus, welcher erstmals am 18. Oktober 2017 in der naturwissenschaftlichen Fachzeitschrift Nature unter dem Namen *AlphaGo Zero* veröffentlicht wurde[32]. Dieser veranschaulicht *AlphaZero* an dem Spiel Go, welches auf Grund seiner enormen Komplexität (Zustandsraum-Komplexität: 10^{360}, Spielbaum-Komplexität: 10^{172})[3] als eine der Königsdisziplinen im Bereich der KI-Forschung angesehen wird. Die Autoren veröffentlichten eine Verallgemeinerung des Algorithmus im darauf folgenden Jahr mit dem Namen *AlphaZero*[31]. *AlphaZero* zeichnet sich dadurch aus, dass es der erste Algorithmus ist, der ein neuronales Netz beinhaltet, das einzig durch Selbstspiel trainiert wird. Bei bisherigen Ansätzen fand immer ein *Supervised Learning* (deutsch: überwachendes Lernen) anhand von Zügen professioneller Spieler statt, sowie eine Verbesserung im Selbstspiel durch *Reinforcement Learning* (deutsch: bestärkendes Lernen)[32].

Somit wurde ein Algorithmus geschaffen, der jegliche Art von Spielen erlernen kann, sofern diese gewisse Eigenschaften erfüllen. Dabei konnte ein Niveau über das heutiger professioneller Go-Spieler hinaus erzielt werden. Beispielsweise wurden für Go bisher unbekannte Strategien gefunden[32].

Ziel dieser Arbeit ist es, *AlphaZero* verständlich zu erklären. Dazu wird vor allem auf das Zusammenspiel von *Monte Carlo Tree Search* und dem neuronalen Netz eingegangen. Außerdem wird auch auf die generelle Funktionsweise von *Monte Carlo Tree Search* und einem neuronalen Netz eingegangen. Zur Veranschaulichung wird das Spiel Vier-Gewinnt als Ersatz für Go hinzugezogen. Dies hat mehrere Gründe. Zum einen gibt es weniger Regeln, die beachtet werden müssen, was den Fokus auf die Funktionsweise von Alpha Zero erleichtert. Zum anderen ist es mit Vier-Gewinnt möglich, eine eigene Implementierung erfolgreich zu trainieren, da der Zustandsraum verglichen viel kleiner ist (Zustandsraum-Komplexität: 10^{14}, Spielbaum-Komplexität: 10^{23})[2], aber dennoch groß genug, um repräsentativ zu sein für die Art von Spielen, die von Algorithmen dieser Art gelöst werden.

AlphaZero ist ein mittels *Reinforcement Learning* sich selbst trainierender Algorithmus, welcher ein neuronales Netz und *Monte Carlo Tree-Search* (*MCTS*) miteinander verbindet. Ein Problem bei Spielen mit großen Zustandsräumen ist, dass nicht jeder mögliche Pfad im Suchbaum traversiert werden kann, um abzuschätzen, welcher der erfolgversprechendste ist. Für diese Fälle hat sich *Monte Carlo Tree-Search* bewährt[5]. *MCTS* ist eine Methode, um in Suchbäumen ein gutes Verhältnis zwischen Exploration und Exploitation zu nutzen, um letztendlich unter begrenzten Ressourcen dennoch die vielversprechendsten Züge zu finden. *MCTS* besteht aus mehreren Phasen. Die zeitintensivste Phase, die Simulation, wird in *AlphaZero* übersprungen und stattdessen werden direkt repräsentative Werte einer Simulation von einem neuronalen Netz zurückgegeben. Das Ergebnis der Simulation durch das neuronale Netz sind zwei Werte: Ein Vektor aus Wahrscheinlichkeiten - um den nächsten Zug zu wählen - und ein Skalar, der die aktuelle Gewinnwahrscheinlichkeit aus gegeben Zustand darstellt. Beide Werte werden genutzt, um effizient vielversprechende Pfade zu nutzen und Spielergebnisse zu approximieren, ohne das Spiel bis Spielende spielen zu müssen. Abbildung 1.1 stellt den Ablauf von *AlphaGo Zero* dar. Durch Selbstspiel werden eine Reihe von Trainingsdaten erschaffen. Diese werden im darauf folgenden Schritt genutzt, um die Genauigkeit des neuronalen Netzes zu erhöhen. Nach dessen Training duelliert sich dieses mit seiner Vorgängervariante. Sollte eine festgelegte Gewinnquote (bei *AlphaGo Zero* waren es 55%, um statistischem Rauschen entgegenzuwirken) vom neuen Netz erreicht werden, wird dieses als neues bestes Netz genutzt - anderenfalls wird es verworfen. Danach wiederholt sich dieser Prozess. Der Unterschied zwischen den Spielen im Training und den im kompetitiven Umfeld ist der, dass jegliche Parameter für Exploration im kompetitiven Modus auf 0 gesetzt werden, um nur die zurzeit als beste geltende Züge zu wählen. In den folgenden Kapiteln werden *MCTS* und das neuronale Netz, die in *AlphaGo Zero* genutzt werden, detailliert beschrieben.

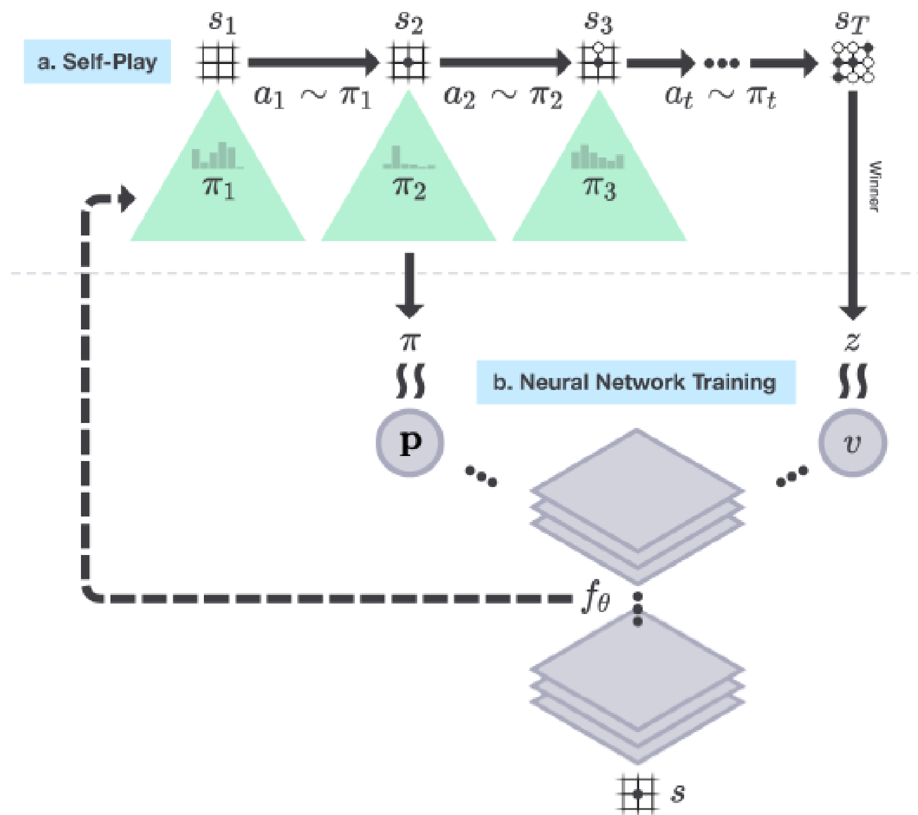


Abbildung 1.1: Ablauf AlphaGo Zero

Quelle: [32]

2 Vier-Gewinnt

Vier-Gewinnt ist ein 2-Spieler Spiel mit einem 6 (Reihen) x 7 (Spalten) Spielbrett. Die Spieler nehmen abwechselnd Züge und entscheiden sich für eine der Spalten, in die sie ihren Spielstein werfen wollen. Dieser fällt so weit es geht herunter, bis er auf den Boden der leeren Spalte gefallen ist oder auf einen anderen Spielstein in selbiger Spalte landet. Es ist nur möglich Spalten auszuwählen, die noch nicht vollkommen mit Spielsteinen belegt sind. Das Spiel ist beendet, sobald ein Spieler vier seiner Spielsteine in einer Reihe hat. Diese Gewinnreihen können, wie in Abbildung 2.1 dargestellt, horizontal, vertikal oder diagonal sein. Das Spiel ist ebenfalls beendet, wenn es keine möglichen Züge mehr gibt. Es wird dann als Unentschieden gewertet. Bei 4-Gewinnt handelt es sich um ein gelöstes Spiel. Für 4-Gewinnt bedeutet dies, dass auch wenn beide Spieler perfekt spielen, der startende Spieler das Spiel immer gewinnt, wenn er das Spiel mit seinem Stein in der mittigen Spalte eröffnet[2]. Mit Hilfe dieser Information lässt sich der Fortschritt eigener Implementationen von *AlphaZero* für Vier-Gewinnt beobachten. In vorangeschrittenen Versionen sollte das neuronale Netz bevorzugt die mittlere Spalte zu Spielbeginn auswählen.

Abbildung 2.1: Mögliche Gewinnreihen bei Vier-Gewinnt

Quelle: https://www.bernhard-gaul.de/spiele/4gewinnt/4gewinnt-menue.php

3 Monte Carlo Tree-Search

In diesem Kapitel werden Methoden vorgeschlagen, die zur Lösung von adversarialen Suchproblemen, auch als Spiele bezeichnet, genutzt werden[29]. Unter Spielen werden zur Vereinfachung hier nur endliche, deterministische, Zwei-Personen-Nullsummenspiele mit vollständiger Information betrachtet, in denen zwei Agenten/Spieler abwechselnd interagieren. Ziel des Kapitels ist es zu verstehen, wie die modifizierte *Monte Carlo Tree-Search* in *AlphaZero* funktioniert. Dafür werden aufeinander aufbauende Methoden beschrieben: *Minimax*, *Alpha-Beta-Pruning*, *Monte Carlo Evaluationen* und *Monte Carlo Tree-Search*.

3.1 Minimax

Minimax ist ein Backtracking (deutsch: Rückverfolgung) Algorithmus, bei dem zwei Spieler abwechselnd einen Zug auswählen. Ein Spieler ist der Maximierer und versucht das höchstmögliche Ergebnis zu erzielen. Der andere Spieler ist der Minimierer und versucht das niedrigste Ergebnis zu erzielen. Um den zu erwartenden Rückgabewert v bei der Wahl einer Aktion a aus gegeben Zustand s zu ermitteln, muss ein vollständiger Suchbaum, ausgehend vom Startzustand s_0, ausgespannt werden. Ab einer gewissen Tiefe ist dies jedoch nicht mehr praktikabel, sodass v basierend auf Heuristiken ermittelt wird. Daraus kann dann unter der Berücksichtigung, dass beide Spieler optimal spielen, den jeweiligen Knoten im Suchbaum der Wert v zugewiesen werden.

Abbildung 3.1 veranschaulicht den *Minimax*-Algorithmus. Δ stellen Max-Knoten dar, aus denen heraus der Maximierer agiert. Der Minimierer agiert aus den Min-Knoten, welche durch ∇ dargestellt werden. Jeder Knoten ist mit v versehen, wobei v unter der in *Minimax* gegeben Annahme des optimalen Spiels bestimmt wird. So ist aus Sicht des startenden Spielers, in der Rolle des Maximierers, die beste Wahl die Aktion a_1, da sie den höchstmöglichen zu erzielenden Wert darstellt. v von $B = (A, a_1)$ entspricht 3, da im darauffolgenden Zug der Minimierer sich für das Minimum, $(B, b_1) = 3$ entscheidet.

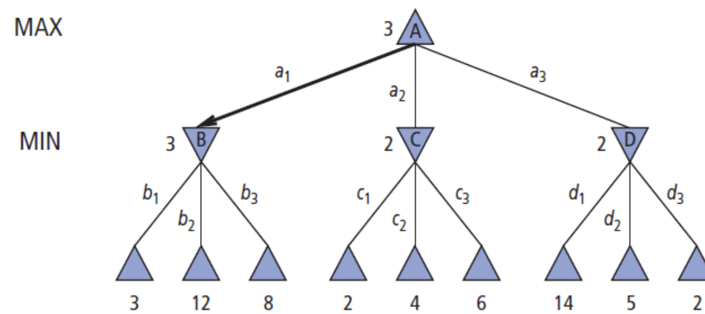


Abbildung 3.1: Beispielhafter Suchbaum des Minimax-Algorithmus

Quelle:[29]

3.2 Alpha-Beta-Pruning

Durch die Struktur des *Minimax*-Algorithmus steigt mit steigender Anzahl an Zügen die Anzahl auszuwertender Spielzustände exponentiell an[29]. Ein Verfahren, um die Anzahl auszuwertender Spielzustände zu minimieren, ist das *Alpha-Beta-Pruning* (dt. Alpha-Beta-Kürzung). Die Vorgehensweise gleicht der des *Minimax*-Verfahrens, jedoch werden Knoten nur so lange expandiert, wie sie einen möglichen Folgezug für den jeweiligen Spieler darstellen. Hierfür werden Knoten zusätzlich mit den Parametern α und β versehen, wobei α für den bislang besten ermittelten Wert entlang des Suchbaums für den Maximierer und β für den bislang besten ermittelten Wert entlang des Suchbaums für den Minimierer steht.

Abbildung 3.2 zeigt den gleichen Suchbaum, der auch für den *Minimax*-Algorithmus verwendet wird. Zu sehen ist hier die schrittweise Expansion der Knoten und das Auslassen von nicht erreichbaren Knoten. In diesem Beispiel wird die Expansion von Knoten C frühzeitig abgebrochen, da - sollte der Maximierer aus Knoten A heraus sich für C entscheiden, würde der Minimierer im darauf folgenden sich für einen Knoten entscheiden, der maximal 2 als Rückgabewert hat. Dieses Ergebnis wäre bereits schlechter als die Wahl von Knoten B aus A heraus, sodass die Suche abgebrochen wird.

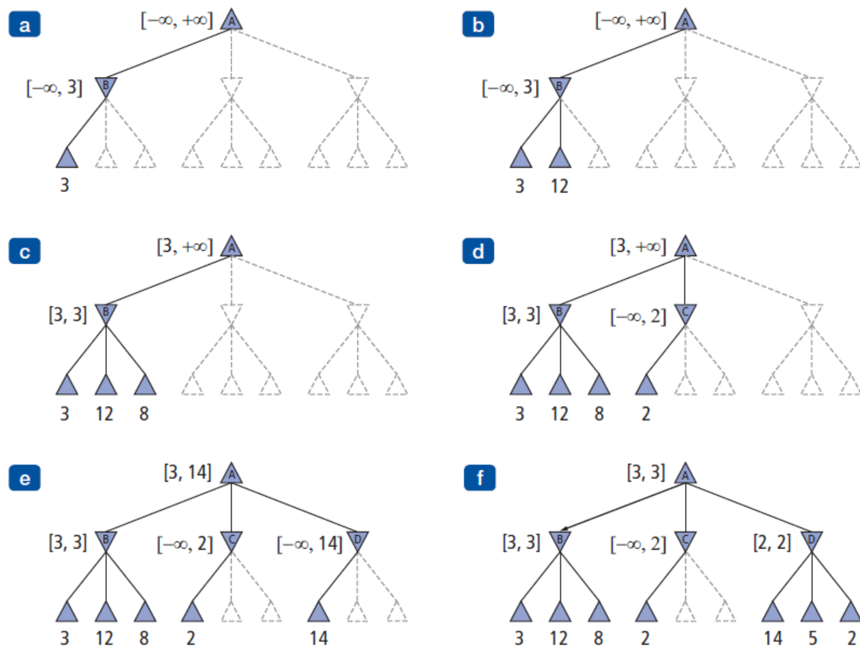


Abbildung 3.2: Beispielhafter Suchbaum des Alpha-Beta-Prunings

Quelle:[29]

3.3 Monte Carlo Methoden

Monte Carlo Methoden kennzeichnen eine Gruppe von Algorithmen, welche „Zufallszahlen zur approximativen Lösung oder zur Simulation verschiedener Prozesse einsetzen“[19]. Monte Carlo ist ein sehr allgemeiner Begriff. In der ursprünglichen Definition bezieht sich Monte Carlo auf einen statistischer Ansatz zur Studie von Integro-Differentialgleichungen, welche in verschiedenen Branchen der Naturwissenschaften auftreten[26]. Aus diesem Grund wird im restlichen Verlauf nur auf Monte Carlo Methoden im Bereich der Spiele in der künstlichen Intelligenz eingegangen: Monte Carlo Evaluationen und *Monte Carlo Tree-Search*.

3.3.1 Monte Carlo Evaluation

Monte Carlo Evaluationen bewerten eine Position P, indem sie von dieser aus Simulationen erzeugen. In diesen Simulationen, auch unter *rollout* oder *playout* bekannt, werden Züge zufällig ausgewählt, bis ein Spielende erreicht wird. Jede Simulation i gibt einen Ergebnisvektor R_i zurück, der die Ergebnisse für beide Spieler beinhaltet. Die Evaluation der Position P nach n Simulationen ist der Mittelwert dieser Ergebnisse:

$$E_n(P) = \frac{1}{n} \sum_{i}^{n} R_i \tag{3.1}$$

Ein Vorteil dieser Art von Simulationen gegenüber damaligen Algorithmen, wie eben vorgestelltes *Minimax* und *Alpha-Beta-Pruning* ist, dass nicht erst beachtlichte Teile des Suchbaums expandiert werden müssen, bevor ein Rückgabewert erhalten wird. Dies ist oft unter zeitlichen Restriktionen nicht möglich. Hinzu kommt, dass eben genannte Algorithmen unter der Annahme arbeiten, dass beide Spieler zu jedem Zeitpunkt optimal spielen. Dadurch ist es bei *Minimax* und *Alpha-Beta-Pruning* schwieriger, gegen nicht-optimale Züge korrekt zu reagieren, Fehler des Kontrahenten können nicht genutzt werden und eine effiziente Berechnung der Evaluierungsfunktion ist nicht möglich[1]. Monte-Carlo Evaluationen werden auch in Kombinationen mit *Minimax/Alpha-Beta-Pruning* verwendet, um ab einer gewissen Tiefe des Suchbaums Rückgabewerte zu schätzen. Anstatt den Baum weiter auszuspannen, wird so die Komplexität geringer gehalten und die Effizienz gesteigert. Gemäß dem Gesetz der großen Zahlen führt eine sehr häufige Durchführung der Monte Carlo Evaluation($n \to \infty$) zu einer präzisen Abschätzung des tatsächlichen

Ergebnisses aus Position P heraus. Bei steigender Komplexität und somit steigender Anzahl an zu beachtender Positionen P ist diese Art von Berechnung jedoch praktisch nicht tragbar, da die dafür notwendigen Berechnungen zu zeitintensiv sind. Diese für praktische Anwendung schwerwiegende Einschränkung führte zur Entstehung der *Monte Carlo Tree-Search*.

3.3.2 Monte Carlo Tree Search

Vor der Einführung der *Monte Carlo Tree-Search* war der gängigste Ansatz eine Verbindung von *Alpha-Beta-Pruning* und Heuristiken zur Bewertung von Positionen[20]. Für das Spiel Go war dieser Ansatz erfolglos - eine Maschine kam nicht über das Niveau eines Amateurs hinaus. *Monte Carlo Tree-Search* ist ein im Bereich der künstlichen Intelligenz für Spiele weit verbreiteter Algorithmus, um effektiv aus einer Vielzahl von Handlungsoptionen die vielversprechenden herauszufiltern. Hierbei wird eine Baumsuche mit einer Monte Carlo Evaluation verbunden, die nicht zwischen *Minimax*-Phase und Monte Carlo-Phase unterscheidet[8]. *MCTS* kann auf jegliche Art von Spielen mit endlicher Länge angewandt werden und basiert auf stochastischen Simulationen[6]. Durch Simulationen können schneller Ergebnisse produziert werden als bei *Minimax*. Mit steigender Anzahl an Simulationen konvergiert die Lösung der *MCTS* mit der von *Minimax*[21].

Grundlegend besteht MCTS auf vier Phasen: *Selektion*, *Expansion*, *Simulation* und *Backpropagation* (deutsch: Rückpropagierung). Diese vier Phasen werden unter dem Begriff „Simulation " zusammengefasst. Da dieser Überbegriff identisch mit dem der Phase der *Simulation* ist, wird die Phase der *Simulation* (wie in der Monte Carlo Evaluation) oft als *rollout* bezeichnet. Nur eine Vielzahl solcher Episoden ermöglichen gute Prognosen. Während die Simulationen durchlaufen werden, baut *MCTS* einen Suchbaum auf, welcher immer weiter ergänzt wird. Dieser Suchbaum beinhaltet zu Beginn nur den Startzustand s_0, welcher übergeben wird. Von dort aus werden die vier Phasen durchlaufen.

1. Selektion

Vom Wurzelknoten ausgehend wird eine legale Aktion (zum Erreichen eines Folgezustands) ausgewählt und zu dem dazugehörigen Kindknoten traversiert. Sofern zu einem solchen Zug noch kein Kindknoten exisiert oder keine Zugoption mehr besteht, ist die Selektionsphase beendet. Bei der Auswahl des Folgezustandes muss ein gutes Verhältnis zwischen Exploration und Exploitation gefunden werden. Da es sich um ein stochastisches Verfahren handelt, wird durch die Exploration sichergestellt, dass möglichst alle

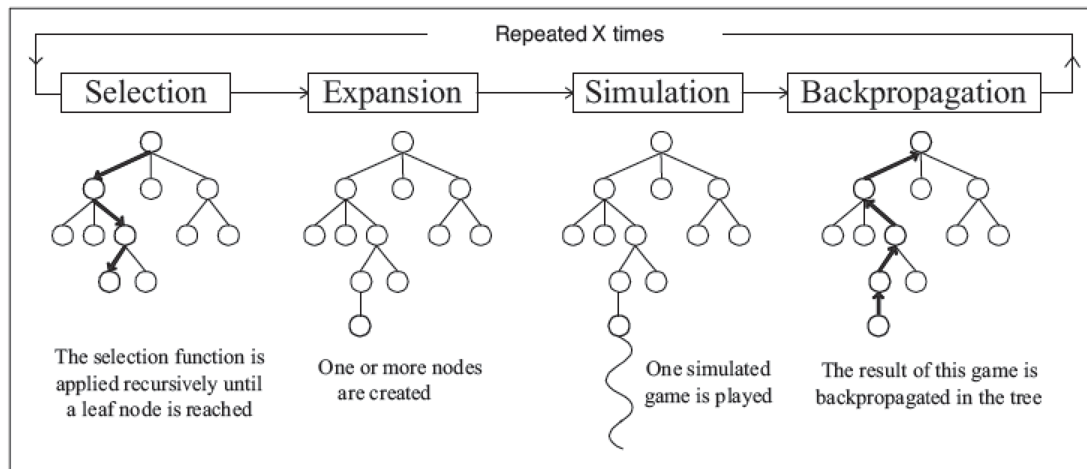


Abbildung 3.3: Die 4 Phasen der Monte-Carlo Tree Search

Quelle: [6]

vielversprechenden Züge gefunden werden. Über die Exploitation wird sichergestellt, dass solche vielversprechenden Pfade auch vermehrt weiter verfolgt werden. Um ein gewünschtes Verhältnis zu erreichen, wird der *Upper Confidence Bound 1(UCB1) applied to trees*, kurz *UCT*, angewandt[21]. *UCB1* stammt aus dem Kontext des Multi-Armed Bandit Problems. Dieses beschreibt ein Problem, in dem begrenzte Ressourcen zwischen miteinander konkurrierenden Handlungen (da nur eine gewählt werden kann) aufgeteilt werden müssen, um die zu erhaltende Belohnung zu maximieren. Zum Zeitpunkt der Aufteilung sind die Eigenschaften der Auswahlmöglichkeiten sowie deren Rückgabewert nur teilweise bekannt und können besser abgeschätzt werden, je mehr Zeit bzw. Ressourcen in diese investiert werden[4][12]. Zwischen Exploration und Exploitation zu wählen ist ein klassisches Dilemma des *reinforcement learnings*. Der Name des Multi-Armed Bandit Problems hat seinen Ursprung in der Vorstellung, dass ein Spieler vor einer Reihe von Slotmaschinen (im englischen als „one-armed bandits" bekannt) steht und sich entscheiden muss, welche der Maschinen am lukrativsten ist. Nach dem Verständnis von *UCB1* soll die Maschine j gewählt werden, die

$$x_j + \sqrt{\frac{2ln(n)}{n_j}} \tag{3.2}$$

maximiert. Hierbei ist x_j die durchschnittliche zu erwartende Belohnung von Maschine j aus, n_j die Anzahl, wie oft j gespielt wurde und n die Gesamtanzahl getätigter Spiele.

Die Wahl einer Maschine ist also von deren zu erwartendem Gewinn und dem Verhältnis, wie oft diese Maschine im Vergleich zu allen anderen gespielt wurde, abhängig. Der zu erwartende Gewinn spiegelt dann den exploitativen Part des Multi-Armed Bandit Problems wider und das Besuchsverhältnis den explorativen Part. Aus *UCB1* wurde speziell für die Baumsuche UCT[21]:

$$\frac{W_i}{n_i} + c\sqrt{\frac{ln(N_i)}{n_i}} \tag{3.3}$$

Hierbei steht W_i für die Anzahl der gewonnen Simulationen aus Zustand i heraus, n_i für die Anzahl aller Simulationen aus diesem Zustand, N_i für die Anzahl sämtlicher Simulationen des Elternknotens und c für einen Parameter, der den Grad der Exploration bestimmt. Die linke Seite der Gleichung betrifft der exploitativen Teil. W_i/n_i ist das Gewinnverhältnis. Die rechte Seite betrifft den explorativen Teil. Diese wird umso größer, je seltener der Knoten bisher besucht wurde. c dient dazu, diese Exploration zu gewichten. Die gängige Zahl hierfür ist $\sqrt{2}$ wie in *UCB1* und wird, falls nötig, empirisch angepasst.

2. Expansion

Der Zustand, dessen dazugehöriger Knoten noch nicht im Suchbaum enthalten ist, wird expandiert, d.h. der Knoten wird dem Suchbaum hinzugefügt.

3. Simulation

Ausgehend vom neu erzeugten Knoten aus Phase 2 wird das Spiel ausgespielt. Dabei handelt es sich um das *rollout*. Bei der Wahl der Folgezüge gibt es verschiedene Strategien. Ursprünglich und weit verbreitet ist das zufällige Auswählen von Zügen, bis ein terminaler Zustand erreicht wird. Ein weitere Möglichkeit ist die *Rapid Action Value Estimation*[10][11], kurz *RAVE*. *RAVE* basiert auf der *All-Moves-As-First*(*AMAF*)-Heuristik. Bei *AMAF* werden Knoten während der Simulationsphase wie Knoten, welche zur Selektionsphase expandiert wurden, behandelt. Dadurch wird ein größerer Teil des Suchbaums aktualisiert, da auch Knoten, die letztendlich eine nicht gespielte Aktion darstellen, aktualisiert werden, wenn die dazugehörige Aktion in einer Simulation auftritt. Die Ergebnisse der *MCTS* sind genauer, wohingegen die *AMAF* schneller arbeitet. Aus diesem Grund wird in der *RAVE* ein Parameter eingeführt, der die Gewichtung der Ergebnisse von *AMAF* und *MCTS* festlegt. Zu Beginn werden die Resultate der *AMAF* stark gewichtet. Mit steigender Anzahl an Simulationen geht deren Gewichtung gegen Null.

Strategien, die Heuristiken zur Evaluation eines Knoten anwenden, sind *progressive bias*[7] und *search seeding*. Beim *progressive bias* wird eine Evaluationsfunktion in der Selektionsphase hinzugezogen. Im Gegensatz dazu werden beim *search seeding* die Werte eines Knoten bei seiner Initialisierung anstatt mit Null mit Werten gemäß einer Evaluationsfunktion belegt[18]. Bei beiden Strategien sinkt der Einfluss der Heuristik mit steigender Besuchszahl des Knotens - beim *progressive bias* wird dies über einen Parameter geregelt, beim *search seeding* erfolgt dies automatisch, da die initiale Belegung bei steigender Anzahl an Aktualisierungen zunehmend an Einfluss verliert. Eine Übersicht über Optimierungsmöglichkeiten bei der *MCTS* liefern *Brown et al.*[5]

4. Backpropagation

Nachdem in Phase 3 ein Spielende und somit das Ende der Simulation erreicht wird, beginnt die Phase der Backpropagation. Dabei werden die Besuchsanzahl und das Gewinnverhältnis jedes Knotens, der während der Simulation durchlaufen wurde, aktualisiert. Nach einer festgelegten Anzahl an Simulationen ist die MCTS beendet. Das Ergebnis der MCTS ist der Kindknoten des Wurzelknotens, der an meisten besucht wurde.

3.4 Monte Carlo Tree Search in AlphaZero

Auch wenn *MCTS* für Situationen mit großen Zustandsräumen genutzt wird, liegt in eben diesen die Herausforderung. Je mehr mögliche Zustände es gibt, desto mehr Simulationen müssen durchlaufen werden, um aussagekräftige Ergebnisse zu erzielen. Bei komplexen Spielen mit relativ kurzen zeitlichen Rahmen kann die erzielte Genauigkeit ungenügend sein. *AlphaZero* steigert die Effizienz der Simulationen, indem es *MCTS* mit einem neuronalen Netz verbindet. Anstatt das Spielergebnis von einem gegebenen Zustand aus per *rollout* zu approximieren, wird dieses von einem neuronalen Netz berechnet. Somit besteht *MCTS* bei *AlphaZero* nur aus drei Phasen: Selektion, Expansion und Backpropagation. Die zeitintensive Simulationsphase fällt weg. Die Simulation wird vom neuronalen Netz f_θ übernommen, wobei θ für die Parameter des neuronalen Netzes steht. Das Ergebnis der Simulation durch das neuronale Netz sind zwei Werte. Zum einen der Vektor P aus Wahrscheinlichkeiten, welchen der möglichen Folgezüge zu wählen, und zum anderen ein Skalar v, der die aktuelle Gewinnchance für den jeweiligen Spieler aus gegebenem Zustand darstellt. In jeder Position s wird eine *MCTS* ausgeführt, die vom neuronalen Netz gestützt wird. Das Resultat der *MCTS* ist der Vektor π, der, wie P, die

Wahrscheinlichkeiten für die Auswahl des Folgezugs angibt. Die errechneten Wahrscheinlichkeiten in π sind meist besser als die in P. Somit kann MCTS hier als ein Mittel zur Verbesserung der Strategie (*policy improvement*) angesehen werden[33]. v wird genutzt, um das Simulationsergebnis einer ursprünglichen *MCTS* ohne *rollout* zu erhalten. P wird genutzt, um während der *MCTS* vielversprechende Züge auszuwählen, die in π resultieren. Das Selbstspiel, das während der Suche π nutzt, um den nächsten Zug zu wählen, und das tatsächliche Spielergebnis z als Probe für den zu erzielenden Wert v, kann als *policy evaluation* (deutsch: Strategiebewertung) angesehen werden. Durch diesen Aufbau stehen *MCTS* und das neuronale Netz in einer Wechselwirkung, um sich gegenseitig zu verbessern, was als *policy iteration* (deutsch: Strategieiteration) bezeichnet wird. P und v werden aktualisiert, um mehr π und z zu entsprechen. Die aktualisierten Parameter wiederum verbessern die Suche im Selbstspiel der darauf folgenden Iteration[32]. Ermöglicht wird dies durch eine Modifikation des *UCT*, der sogenannte *Polynomial Upper Confidence Tree*(*PUCT*). Die Suche startet beim Wurzelknoten s_0 und endet wenn die Simulation zu Zeitschritt L einen Blattknoten s_L erreicht. Zu jedem dieser Zeitschritte $t < L$ wird eine Aktion a nach

$$a_t = argmax(Q(s_t, a) + U(s_t, a)) \tag{3.4}$$

ausgewählt, mit

$$U(s, a) = c_{puct} P(s, a) \frac{\sqrt{\sum_b N(s, b)}}{1 + N(s, a)} \tag{3.5}$$

$Q(s, a)$ steht für den durchschnittlichen Wert der Aktion a. $N(s, a)$ gibt an, wie oft Aktion a aus Zustand s gewählt wurde und $\sum_b N(s, b)$ gibt die Gesamtanzahl gewählter Aktionen aus Zustand s an. Damit die obige Gleichung berechnet werden kann, müssen die Knoten im sich aufspannenden Suchbaum der *Monte Carlo Tree-Search* zusätzliche Informationen speichern. So wird jeder Knoten mit Statistiken für alle legalen Aktion $a \in A(s)$ versehen mit

$$\{N(s, a), W(s, a), Q(s, a), P(s, a)\} \tag{3.6}$$

Hier steht $N(s, a)$ für die Besuchszahl, $W(s, a)$ für den totalen Aktionswert, $Q(s, a)$ für den durchschnittlichen Aktionswert und $P(s, a)$ steht für die bisherige Wahrscheinlichkeit, den mit Aktion a assoziierten Knoten auszuwählen. Wie bei *UCT* wird zwischen Knoten, die konsistent gute Werte erzielen und Knoten, welche unerforscht sind, abgewogen. Die Exploration wird nun vom neuronalen Netz geleitet. Je besser das neuronale Netz, desto besser die ausgewählten Züge der *MCTS*. Bei wachsender Anzahl an Spielen wird der errechnete Wert mehr und mehr von der linken Seite der Gleichung, dem Gewinnverhältnis, bestimmt. Während der Simulationen von *AlphaZero* wird v genutzt,

um das Ergebnis der fehlenden *rollout*-Phase vorherzusagen und p wird in Verbindung mit $PUCT$ genutzt, um den nächsten Zug zu bestimmen. Wird ein Knoten expandiert, so muss es sich um einen Blattknoten L handeln und dessen Statistiken werden folgendermaßen initialisiert:

$$\{N(s_L, a) = 0,\ W(s_L, a) = 0,\ Q(s_L, a) = 0,\ P(s_L, a) = p_a\} \tag{3.7}$$

p_a wird vom neuronalen Netz vorhergesagt. In der Phase der Rückpropagierung werden die Statistiken aktualisiert mit

$$\begin{aligned} N(s_t, a_t) &= N(s_t, a_t) + 1 \\ W(s_t, a_t) &= W(s_t, a_t) + v \\ Q(s_t, a_t) &= \frac{W(s_t, a_t)}{N(s_t, a_t)} \end{aligned} \tag{3.8}$$

Nach Beendigung der Monte Carlo *Tree-Search* wählt *AlphaZero* eine Aktion a aus s_0, die proportional zu der potenzierten Besuchshäufigkeit ist, mit τ als Temperaturparameter, um den Grad der Exploration zu bestimmen:

$$\pi(a|s_0) = \frac{N(s_a, a)^{\frac{1}{\tau}}}{\sum bN(s_0, b)^{\frac{1}{\tau}}} \tag{3.9}$$

Der während der *MCTS* erzeugte Baum wird für folgende Simulationen teilweise weiterverwendet. Der Knoten, der der ausgewählten Aktion nach Beendigung der *Monte Carlo Tree-Search* entspricht, wird zum neuen Wurzelknoten der darauf folgenden *MCTS*. Alle dazugehörigen Statistiken werden behalten. Der Rest des ursprünglichen Baums aus vorheriger MCTS wird verworfen.

Nachfolgend wird beispielhaft ein Teil einer Partie von Vier-Gewinnt mittels der verbesserten *Monte Carlo Tree-Search* in *AlphaZero* dargestellt. In Abbildung 3.4 wird von der Wurzel aus bis zu einem Blattknoten traversiert. Dabei werden die Knoten nach Gleichung 3.4 ausgewählt. Abbildung 3.5 veranschaulicht eine mögliche Selektion des Knotens in Abbildung 3.4. Aus Gründen der Übersicht wurde für alle dargestellten Knoten ein $N = 10$ angenommen.

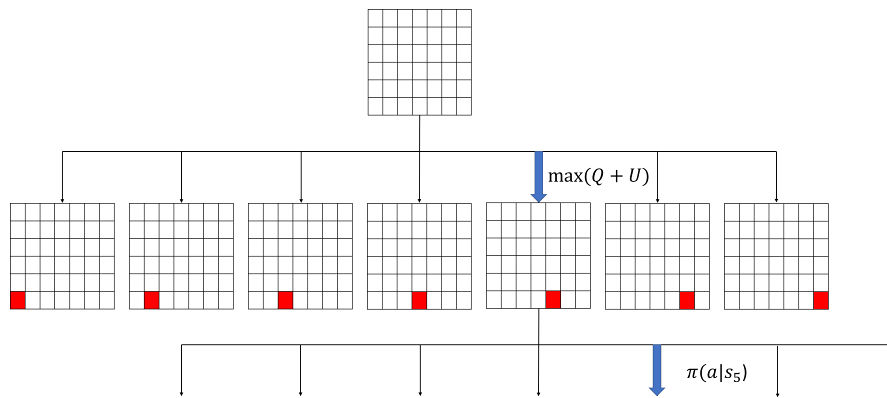
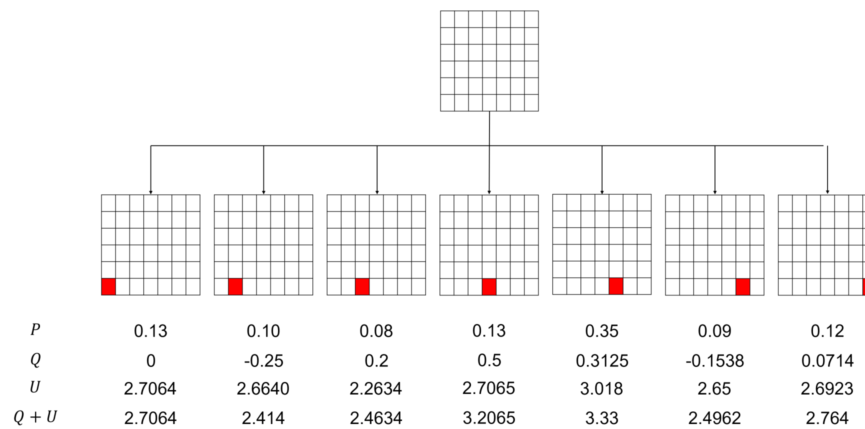


Abbildung 3.4: MCTS: Selektion in AlphaZero



P	0.13	0.10	0.08	0.13	0.35	0.09	0.12
Q	0	-0.25	0.2	0.5	0.3125	-0.1538	0.0714
U	2.7064	2.6640	2.2634	2.7065	3.018	2.65	2.6923
$Q+U$	2.7064	2.414	2.4634	3.2065	3.33	2.4962	2.764

Abbildung 3.5: MCTS: Selektion des nächsten Knotens in AlphaZero

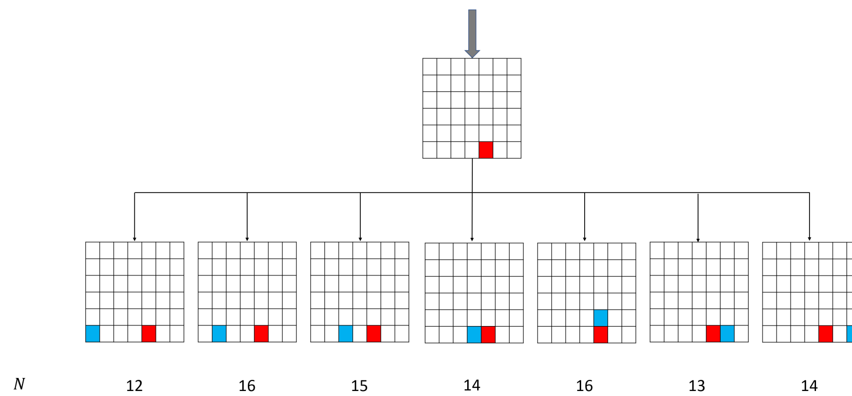


Abbildung 3.6: MCTS: Expansion in AlphaZero

Eine Berechnung für den 3. Knoten von rechts mittels Gleichung 3.4 und beispielhaften Werten ist:

$$
\begin{aligned}
Q &= 0.3125 \\
U &= \sqrt{2} \times 0.35 + \sqrt{\frac{70}{11}} \\
Q + U &= 3.33
\end{aligned}
\tag{3.10}
$$

Abbildung 3.6 zeigt ein mögliches Ende nach 100 *MCTS*-Simulationen, bevor es zu einer endgültigen Expansion nach Gleichung 3.9 kommt. In Abbildung 3.7 wird der zu dem selektierten Zug gehörende Kindknoten dem Suchbaum hinzugefügt. Die dazugehörigen Statistiken werden initialisiert und p und v werden vom neuronalen Netz vorhergesagt. Abbildung 3.8 zeigt beispielhaft die Rückpropagierung des Aktionswertes Q. Dieser wird entlang aller an der Suche beteiligten Knoten aktualisiert, genau wie alle weiteren Statistiken nach dem in Gleichung 3.8 beschriebenen Vorgehen.

Zusammenfassend wurde in diesem Kapitel aufeinander aufbauend beschrieben, wie unterschiedliche Algorithmen angewandt werden, um nachfolgende Aktionen aus einem Zustand bedingter Informationen auszuwählen. *MCTS* ist eine effektive Methode, um ein angemessenes Verhältnis zwischen Exploration und Expolitation zu erhalten und zusätzlich frühzeitig Handlungsempfehlungen aufzeigt. Die größte Einschränkung bei der *MCTS* ist der zeitliche Aufwand, Zuständen Aktionswerte durch das *ausrollen* zuzuschreiben. *AlphaZero* umgeht diesen zeitlichen Aufwand, indem es ein neuronales Netz für die Approximation dieser Aktionswerte hinzuzieht. Zusätzlich lenkt das *neuronale*

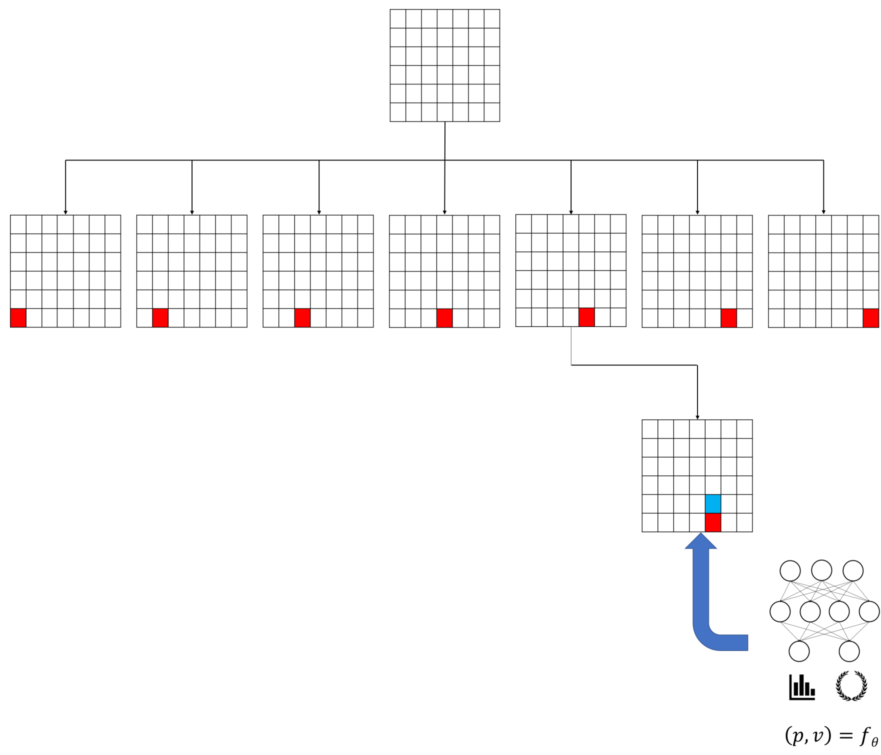


Abbildung 3.7: MCTS: Evaluation in AlphaZero

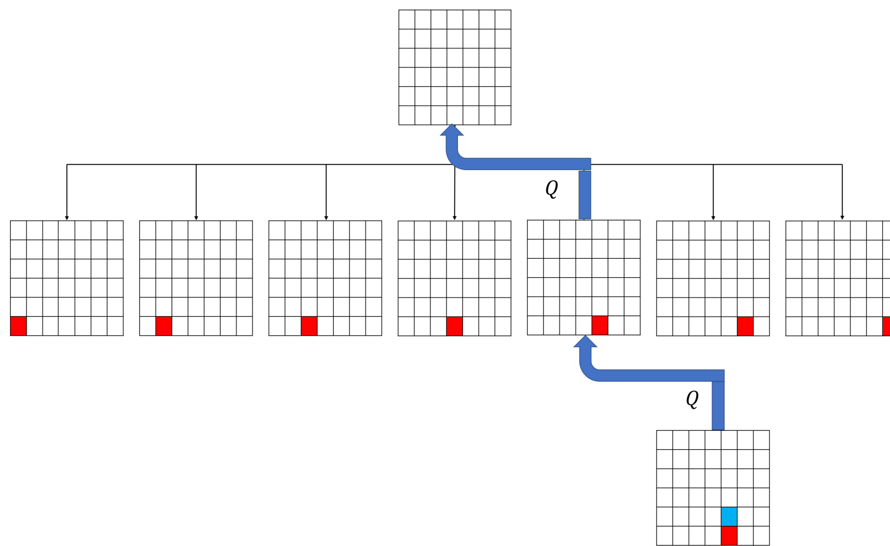


Abbildung 3.8: MCTS: Backpropagation in AlphaZero

Netz die MCTS, um effizienter vielversprechende Pfade auszuwählen. Die Ergebnisse der *MCTS*, sowie das tatsächliche Spielergebnis, werden genutzt, um das neuronale Netz zu optimieren. Im nachfolgenden Kapitel wird beschrieben, wie genau das neuronale Netz in *AlphaZero* arbeitet.

4 Neuronale Netze in AlphaZero

Das folgende Kapitel gibt zunächst eine allgemeine Einführung in neuronale Netze (gleichzusetzen mit neuronalen Netzwerken) und geht im Anschluss genauer auf die Struktur des neuronalen Netzes in *AlphaZero* ein. Die nachfolgenden Beschreibungen der Funktionsweise neuronaler Netze stützen sich auf Andrew Ngs Materialien zum Thema „deep learning“ und Michael Nielsens Buch „Neural Networks and Deep Learning“.

4.1 Funktionsweise neuronaler Netze

Grundsätzlich erhält ein neuronales Netz eine oder mehrere Eingaben x_i, auch als *Feature* bezeichnet, und produziert eine oder mehrere Ausgaben $\hat{y}$. Die erste Schicht ist die Eingabeschicht und wird als *Input Layer* bezeichnet. Die letzte Schicht ist die Ausgabeschicht und wird als *Output Layer* bezeichnet. Alle Schichten dazwischen werden als *Hidden Layers*, zu deutsch „verborgene Schichten“ bezeichnet. Veranschaulicht ist dies in Abbildung 4.1. Netzwerke mit vielen *Hidden Layers* werden häufig als „deep networks“ (deutsch: tiefe Netzwerke) bezeichnet. Dabei gibt es keine klare Abgrenzung, ab wann ein neuronales Netz als tief angesehen wird. Jede Schicht eines neuronalen Netzes besteht aus Neuronen. Auf dem Weg zwischen Ein- und Ausgabe werden diese Neuronen, welche mit Gewichten w (für den englischen Begriff „weight“) und Bias b versehen sind, durchlaufen. Dabei besitzt jedes Neuron genau so viele Gewichte, wie es Eingabedaten gibt. Zusätzlich enthält es noch einen Bias, der für jede Eingabe gleich ist. Während die Eingabedaten von Schicht zu Schicht weitergereicht werden, werden verschiedene mathematische Operationen, abhängig von der gewählten Architektur des Netzes, auf die Eingabedaten angewandt. Dabei erzeugt jede Schicht eine Ausgabe, welche als neue Eingabe für die darauf folgende Schicht genutzt wird. Am Ende wird überprüft, ob die tatsächliche Ausgabe des neuronalen Netzes $\hat{y}$ der gewünschten Ausgabe y entspricht. Im anschließenden Trainingsprozess werden die Gewichte und Bias möglichst so angepasst, dass die Ausgabe des neuronalen Netzes mehr der gewünschten Ausgabe entspricht.

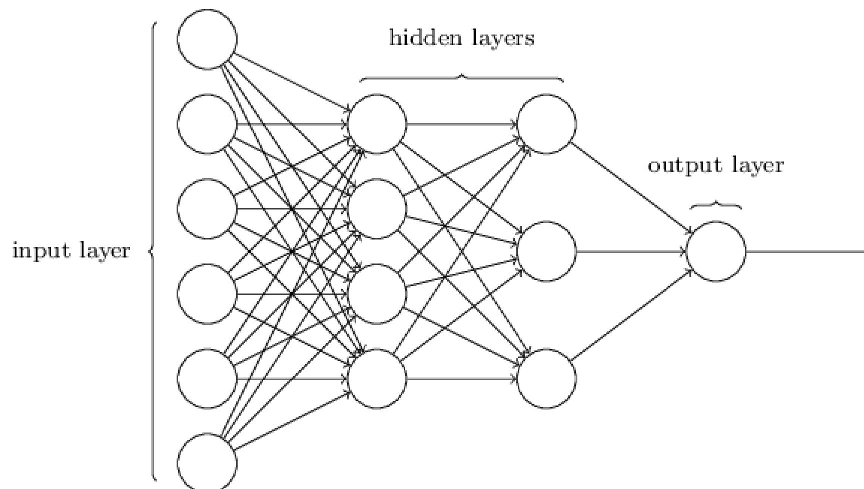


Abbildung 4.1: Die verschiedenen Schichten eines neuronalen Netzes

In der Literatur ist die Notation für neuronale Netze nicht einheitlich, darum sollte beim Lesen unterschiedlicher Werke zu Beginn die Bedeutung der verwendeten Notation geprüft werden. Ein *Feature*-Vektor X der Form (n, m), mit n als Anzahl an *Features* und m als Anzahl an Trainingsbeispielen, dient als Eingabe des neuronalen Netzes. Somit ist X die Eingabematrix mit $X \in R^{n_x \times m}$ mit n_x als Eingabegröße. Eine Eingabe wird, unter Berücksichtigung der mit ihr assoziierten Gewichten $w_1, w_2, ...w_n$ und Bias b, welche in den Neuronen gespeichert sind, mittels einer Aktivierungsfunktion g verrechnet. Das Ergebnis wird als Aktivierung a bezeichnet. Verschiedene Schichten können unterschiedliche Aktivierungsfunktionen aufweisen. Die Ausgabe eines neuronalen Netzes ist von dessen beabsichtigter Funktion abhängig. Unabhängig davon, zu welchem Zwecke neuronale Netze eingesetzt werden, wird deren Effektivität über *Kostenfunktionen* (*Cost Functions*) gemessen. Dabei stehen die Kosten für die Diskrepanz zwischen tatsächlicher Ausgabe und gewollter Ausgabe. Die *Kostenfunktion* bezieht sich auf den gesamten Trainingssatz. Zur Bestimmung der Kosten einzelner *Features* wird eine *Verlustfunktion* (*Loss Function*) genutzt. Abhängig von der Art des zu lösenden Problems werden unterschiedliche *Kosten*- und *Verlustfunktionen* genutzt. Wie genau neuronale Netze iterativ ihre Kosten senken und somit ihre Effektivität steigern, wird im folgenden anhand des *Mean Squared Errors* (*MSE*) beschrieben, welcher auch von *AlphaZero* zur Verbesserung von v genutzt wird. Der *MSE* berechnet die mittlere quadratische Abweichung zwischen

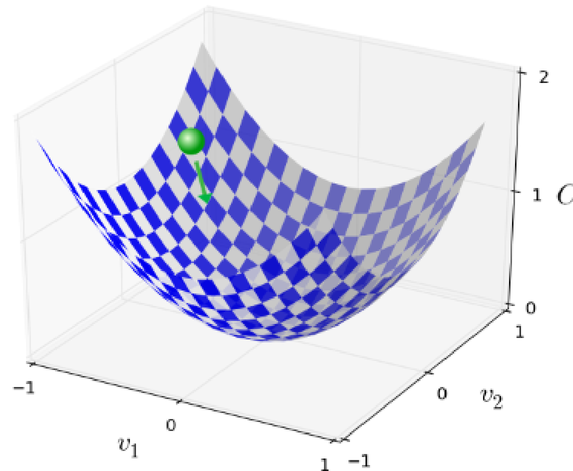


Abbildung 4.2: Visualisierung der angestrebten Veränderung des Gradienten in einer multidimensionalen Funktion

Quelle: http://neuralnetworksanddeeplearning.com/chap1.html

gewünschtem Output $y = y(x)$ und tatsächlichem Output $\hat{y}$.

$$C(w, b) = \frac{1}{2n} \sum_x \|y(x) - \hat{y}\|^2 \tag{4.1}$$

w bezieht sich auf alle Gewichte des neuronalen Netzes, b auf alle Bias, n ist die Gesamtanzahl der Trainingseingabedaten, $y(x)$ die gewollte Ausgabe zu x und $\hat{y}$ die tatsächliche Ausgabe. Diese hängt von x, w und b ab. Diese Abhängigkeit wird in der Gleichung nicht explizit dargestellt. Der Grund für die Bestimmung der Effektivität des neuronalen Netzes mittels einer *Kostenfunktion* liegt darin, dass man auch geringe Veränderungen, die nicht zwingend Einfluss auf die letztliche Ausgabe des Netzes haben, messen kann. Aufgrund der Form des *MSE* können die Kosten C nie negativ sein. Je näher $\hat{y}$ an y ist, desto geringer fallen die Kosten aus. Daher wird versucht, w und b so zu wählen, dass C möglichst klein ausfällt. Dies wird mit Hilfe des *Gradientenverfahrens* erreicht.

4.2 Gradientenverfahren

Um das Gradientenverfahren zu erklären, wird in Abbildung 4.2 beispielhaft eine Funktion $C(v)$ dargestellt, deren Minimum per Gradientenverfahren zu bestimmen ist. Zur Vereinfachung wird C nur von zwei Variablen, v_1 und v_2 beeinflusst. In Anbetracht der Tatsache, dass in neuronalen Netzen die Anzahl der beeinflussenden Variablen bis in die Milliardenbereiche gehen kann, wird deutlich, dass eine Bestimmung der Minima durch Berechnung der Extrempunkte nicht möglich ist. Anstatt die Extrempunkte zu berechnen, können v_1 und v_2 schrittweise so verändert werden, dass die Kosten von C kontinuierlich sinken. Wie dies erreicht wird, veranschaulicht Abbildung 4.2. Dabei stellt der grüne Ball das Ergebnis von C mit der aktuellen Belegung der Variablen v_1 und v_2 dar. Der grüne Pfeil visualisiert die Senkung der Kosten von C bei Anpassung von v_1 und v_2. Die Veränderung von C, ΔC, in Abhängigkeit von v, wird dargestellt als:

$$\Delta C \approx \frac{\partial C}{\partial v_1} \Delta v_1 + \frac{\partial C}{\partial v_2} \Delta v_2 \tag{4.2}$$

Der Vektor der partiellen Ableitungen, ∇C, wird als Gradient bezeichnet.

$$\nabla C \equiv (\frac{\partial C}{\partial v_1}, \frac{\partial C}{\partial v_2})^T \tag{4.3}$$

T steht für das Transponieren von Vektoren. In diesem Zusammenhang werden Zeilenvektoren in Spaltenvektoren transponiert, um eine effektive Matrixmultiplikation zu ermöglichen. Somit kann ΔC zu

$$\Delta C \approx \nabla C \cdot \Delta v \tag{4.4}$$

umgeschrieben werden. Die Besonderheit dieser Gleichung ist, dass wenn die Veränderung in v durch

$$\Delta v = -\eta \nabla C \tag{4.5}$$

beschrieben wird, wobei η, die *Lernrate*, eine kleine positive rationale Zahl ist, fällt ΔC immer negativ aus. Zu beachten ist hierbei, dass es sich bei Gleichung 4.4 um eine Approximation handelt. Nach Gleichung 4.5 wird v nach folgendem Schema aktualisiert:

$$v \rightarrow v' = v - \eta \nabla C \tag{4.6}$$

Durch die Veränderungen des Gradienten, welcher die Steigung der einzelnen Variablen in C darstellt, kann so iterativ ein Minimum erreicht werden. Das gleiche Prinzip lässt sich auf neuronale Netzwerke mit vielen Variablen anwenden. Statt v werden w und b per *Gradientenverfahren* verändert, um C zu senken. Aus Gleichung 4.6 wird

$$\begin{aligned} w \rightarrow w' &= w - \eta \frac{\partial C}{\partial w} \\ b \rightarrow b' &= b - \eta \frac{\partial C}{\partial b} \end{aligned} \tag{4.7}$$

wobei w sich nun auf einzelne Gewichte bezieht. In Gleichung 4.1 bezog sich w auf alle Gewichte, um die Notation dieser simpel zu halten. Abhängig von der Literatur werden alle Gewichte auch als W zusammengefasst. Wird das *Gradientenverfahren* auf ein neuronales Netzwerk mit einer Vielzahl an Variablen angewandt, dauert die Berechnung sehr lange, da der Gradient für jede Eingabe x einzeln berechnet werden muss, bevor dieser gemittelt werden kann:

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x \tag{4.8}$$

Um diese Berechnungen zu beschleunigen, wird das *stochastische Gradientenverfahren*, kurz *SGD*, angewandt. Bei dem *stochastische Gradientenverfahren* wird der wahre Gradient ∇C approximiert, indem dieser aus einer zufälligen Auswahl einiger Trainingsbeispiele (Proben), welche im Kontext des *stochastische Gradientenverfahrens* als *mini-batch* bezeichnet werden, berechnet wird. Nachdem jede Probe für die Approximation verwendet wurde, ist eine *Episode* beendet. Nach jeder *Episode* werden die Parameter des neuronalen Netzes aktualisiert. Die Parameter des neuronalen Netzes werden mittels *Backpropagation* aktualisiert. Im Gegensatz zu Kapitel 3 kann in diesem Kontext *Backpropagation* passender als *Fehlerrückführung* übersetzt werden. Die Anwendung der *Backpropagation* ist eine praktische Anwendung der Kettenregel, um den Gradienten einer Funktion in Bezug zu den Gewichten mehrschichtiger Module zu berechnen[23]. Dabei kann der Gradient berechnet werden, indem dieser rückwärts, startend bei der Ausgabe des Moduls, berechnet wird. Das hier beschriebene *stochastische Gradientenverfahren* fällt in die Kategorie der Optimierungsfunktionen, die oft als *Optimizer* (deutsch: Optimierer) bezeichnet werden. *SGD* wird in *AlphaZero* verwendet.

4.3 Unstable Gradient Problem

Mit steigender Anzahl an Schichten in einem neuronalen Netz steigt die Wahrscheinlichkeit, dass das *Unstable Gradient Problem* (deutsch: Problem instabiler Gradienten) auftritt. Dies ist durch die schichtweise Berechnung der Gradienten zu begründen. Bei dem Durchqueren der Schichten werden diverse Matrixmultiplikationen durchgeführt. Haben die Gradienten einen geringen Wert (< 1), so führt eine Vielzahl an Multiplikationen dazu, dass der Wert immer kleiner wird, bis er nahe 0 ist und die Parameter des zu trainierenden Modells eines neuronalen Netzes nicht mehr aktualisiert werden. Dieser Umstand wird als *Vanishing Gradient* (deutsch: verschwindener Gradient) bezeichnet. Umgekehrt kommt es zum *Exploding Gradient* (deutsch: explodierenden Gradienten), wenn die Werte der Gradienten beim Durchqueren der Schichten einen hohen Wert (> 1) aufweisen und somit durch starke Wertveränderungen der Reliabilität des Modells schaden. *Explodierende* und *verschwindene Gradienten* können auch bei Werten zwischen 0 und 1 auftreten, und zwar dann, wenn die Gewichte mit sehr hohen oder sehr niedrigen Werten belegt sind. Das potenzielle Auftreten des *Vanishing Gradients* und des *Exploding Gradients* wird als *Unstable Gradient Problem* beschrieben. Verschiedene Methoden haben sich als nützlich erwiesen, um dem *Unstable Gradient Problem* entgegenzuwirken. Welche Methoden genutzt werden, hängt von Zweck und Architektur des neuronalen Netzes ab. Nachfolgend wird die Architektur des verwendeten neuronalen Netzes in *AlphaZero* beschrieben und die Wahl dieser Architektur begründet.

4.4 Das neuronale Netz in AlphaZero

Abbildung 4.3 stellt die Architektur des neuronalen Netzes von *AlphaGo Zero* dar. Zu sehen ist ein neuronales Netz, das als Input den Zustand des Spiels erhält und diesen in einem *Convolutional Layer* verarbeitet. Darauf folgen 20 oder 40 *Residual Layers* und anschließend wird das neuronale Netz in einen *value-* und einen *policy-head* aufgespalten. Die in *AlphaGo Zero* beschriebene Architektur besteht aus 20 *Residual Layers*, jedoch wiesen einige der dort zur Validation verwendeten Netze 40 *Residual Layers* auf.

Abbildung 4.3: Aufbau des neuronalen Netzes in AlphaZero

Quelle: https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0

4.4.1 Convolutional Layer

Der Name *Convolutional Layer* stammt von der gleichnamigen mathematische Operation, der „Convolution“ (deutsch: Faltung / Konvolution). Diese *Konvolution* ist das Erschaffen einer Funktion aus zwei Funktionen, die als Produkt dieser verstanden werden kann. Somit werden mehrere Funktionen in eine „gefaltet“. In der Literatur wird das erste Argument der *Konvolution* häufig als „Input“ (deutsch: Eingabe) bezeichnet, das zweite Argument als „Kernel“ (deutsch: Kern / Kernel). Die Ausgabe wird als „Feature Map“ (deutsch: Merkmalskarte) bezeichnet. Netzwerke, die unter anderem aus einer Vielzahl von *Convolutional Layers* bestehen, werden als *Convolutional Neural Networks*, kurz *CNN*, bezeichnet. Deren häufigster Anwendungsfall ist die Bilderkennung, in dem *CNNs* durch Architekturen wie *AlexNet*[22] hohe Popularität errangen. Generell sind *CNNs* spezialisiert auf die Bearbeitung von Daten, die eine gitterähnliche Topologie aufweisen[13]. *CNNs* halten die räumlichen und temporalen Abhängigkeiten gitterähnlicher Daten fest. Aus diesem Grund bieten sie sich auf für Spiele wie *Go* oder *Vier-Gewinnt* an, um die Position der Spielsteine und deren Abhängigkeiten zueinander festzuhalten. Abbildung 4.4 veranschaulicht den Prozess der *Konvolution*. Dabei werden die räumlichen Abhängigkeiten entsprechend der Größe des *Kernels* festgehalten. In diesem Fall sind beispielsweise a, b, e und f in dem ersten Teil der Ausgabe zueinander in Bezug gesetzt. Die in *AlphaGo Zero* verwendeten *Convolutional Layers* verwenden einen 3x3 Filter, wobei hier der Begriff „Filter“ mit dem des *Kernels* gleichzusetzen ist.

4.4.2 Residual Layer

Residual Layers (deutsch: residuale Schichten) werden im Kontext sehr komplexer Netzwerke genutzt. In diesen tritt häufig, wie in Abschnitt 4.3 beschrieben, das Problem instabiler Gradienten auf. Bei solch komplexen Netzwerken existiert auf Grund deren Länge ein *Unstable Gradient Problem*. *Residual Layers* umgehen das Problem der steigenden Tiefe, indem sie *Skip-Connections* einbauen. Werte aus früheren Schichten werden direkt als Eingabe für die nächste Schicht und spätere Schichten genutzt und überspringen dabei 0 bis n Schichten, wobei n für die absolute Anzahl an *Hidden Layers* steht. Das Überspringen einer Schicht wird als *Skip Connection* bezeichnet. Abbildung 4.5 zeigt beispielhaft die Struktur eines *Residual Networks*, welches aus drei Schichten besteht, die alle über eine *Skip Connection* verfügen. Die linke Seite der Abbildung stellt die typische Darstellung solcher *residualen Blöcke* dar und die rechte zeigt die aufgelöste Struktur an,

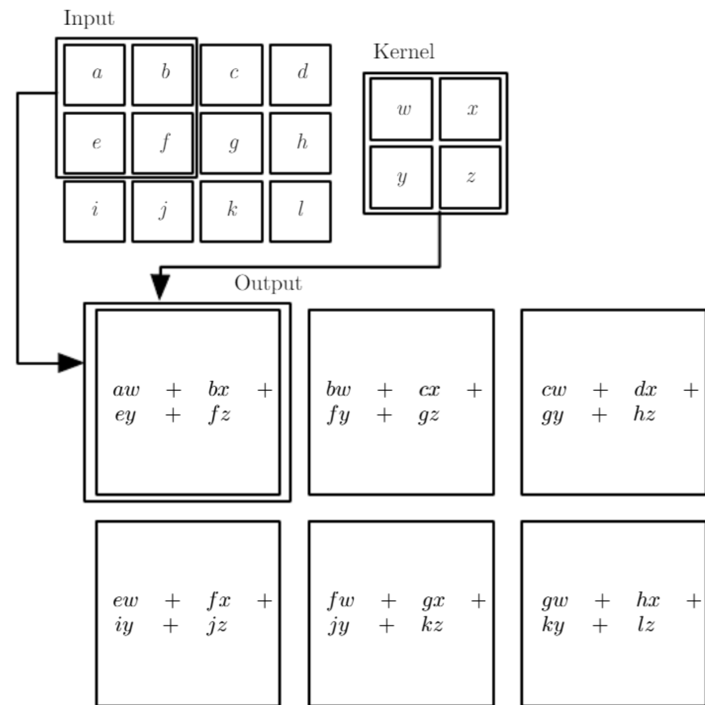


Abbildung 4.4: Beispielhafte Struktur eines Convolutional Neural Networks

Quelle: [35]

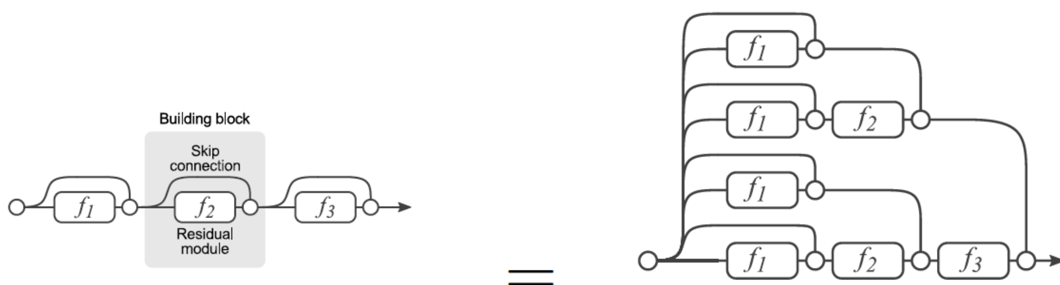


Abbildung 4.5: Aufbau residualer Netzwerke

Quelle: [35]

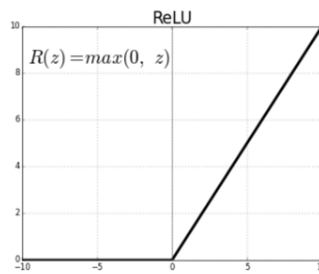


Abbildung 4.6: Grafische Visualisierung der Rectified Linear Unit

Quelle: https://medium.com/@kanchansarkar/relu-not-a-differentiable-function-why-used-in-gradient-based-optimization-7fef3a4cecec

die hinter der üblichen Notation steht. Die Eingaben werden in allen möglichen Kombinationen miteinander verbunden. So wird beispielsweise die Eingabe der ersten Schicht f_1 alle Schichten (f_1, f_2, f_3) durchlaufen, was im untersten Weg dargestellt wird, aber sie übergeht auch alle Schichten, wie es der obere Weg darstellt. Durch diese Kombination von Eingaben aus unterschiedlichen Schichten wird das Netz resistenter gegenüber dem *Unstable Gradient Problem*. *Residual Layers* beinhalten zusätzlich zu den *Skip Connections* meist noch ein *Convolutional Layer*, wie es auch in *AlphaGo Zero* der Fall ist. Zu begründen ist dies dadurch, dass *Residuale Netze* im Kontext der Bildverarbeitung entstanden sind[15], in der mit *Convolutional Layern* gearbeitet wird. Da *Residual Layers* intern aus mehreren Komponenten zusammengesetzt sind, werden diese Interna in der Literatur häufig als *Residual Block* (deutsch: residualer Block) zusammengefasst.

4.4.3 Aktivierungsfunktionen

Die in *AlphaGo Zero* meist verwendete Aktivierungsfuntkion ist die *Rectified Linear Unit*, kurz *ReLU*. Abbildung 4.6 zeigt deren charakteristischen Verlauf. Die *ReLU* ist eine der meist verwendeten Aktivierungsfunktion [27]. Ihre Beliebtheit ist damit zu begründen, dass sie dem *Vanishing Gradient Problem* entgegenwirkt und leicht zu berechnen ist. Im positiven Wertebereich verläuft diese linear und vermeidet dadurch die vermehrte Berechnung geringer Gradienten, da deren Ableitung immer 1 (bei $x > 0$) ergibt. Da es zu keiner Eingrenzung des Wertebereichs durch die *ReLU* kommt, besteht weiterhin die Gefahr *explodierender Gradienten*. *Normalisierung* ist ein Verfahren, um unter anderem *explodierenden Gradienten* entgegenzuwirken. Zusammenfassend werden die Aktivierungen der Schichten eines neuronalen Netzes über deren Durchschnitt und Standardabwei-

Input: Values of x over a mini-batch: $\mathcal{B} = \{x_{1\ldots m}\}$;
Parameters to be learned: γ, β
Output: $\{y_i = \mathrm{BN}_{\gamma,\beta}(x_i)\}$

$$\mu_\mathcal{B} \leftarrow \frac{1}{m}\sum_{i=1}^{m} x_i \qquad \text{// mini-batch mean}$$

$$\sigma_\mathcal{B}^2 \leftarrow \frac{1}{m}\sum_{i=1}^{m} (x_i - \mu_\mathcal{B})^2 \qquad \text{// mini-batch variance}$$

$$\widehat{x}_i \leftarrow \frac{x_i - \mu_\mathcal{B}}{\sqrt{\sigma_\mathcal{B}^2 + \epsilon}} \qquad \text{// normalize}$$

$$y_i \leftarrow \gamma \widehat{x}_i + \beta \equiv \mathrm{BN}_{\gamma,\beta}(x_i) \qquad \text{// scale and shift}$$

Abbildung 4.7: Batch-Normalization

Quelle: [17]

chung angepasst. In der Umsetzung wird diese *Normalisierung* meist auf *mini-batches* angewandt, um effizienter zu sein. Das genaue Verfahren trägt daher den Namen *Batch Normalization* und wird durch die Gleichungen in Abbildung 4.7 beschrieben. Zu sehen ist, dass nach der Subtraktion des Durchschnitts des *batches* und der Division durch die Standardabweichung das Ergebnis nicht direkt verwendet wird, sondern noch mit den Parametern γ und β verrechnet wird. Dies sind zu trainierende Parameter, die dem *SGD* ermöglichen, die durch die Normalisierung veränderten Parameter zu optimieren, um die Identität (deren beabsichtigte Funktion/Charakteristik, welche durch Normalisierung zu stark eingeschränkt werden kann) der Aktivierungsfunktionen zu erhalten [17]. Vereinfacht ausgedrückt minimiert *Batch Normalization* das Problem der wechselnden Eingabewerte, indem es den Effekt, den die Aktualisierung der Parameter in früheren Schichten auf die Verteilung der Werte, die spätere Schichten erhalten (was als *Covariance Shift* bezeichnet wird), reduziert. Zuletzt ist zu *Batch Normalization* noch anzumerken, dass diese auch zu schnelleren Lerneffekten führt, indem die Wertebereiche der Eingabedaten einander angepasst werden. Für *AlphaGo Zero* hat dieser Faktor jedoch keine Relevanz, da der Wertebereich der Eingabedaten für alle *Features* identisch ist.

Die letzte verwendete Aktivierungsfunktion in *AlphaGo Zero* ist *tanh*, der *Hyperbeltangens*, definiert als:

$$f(x) = \frac{e^x - e^- x}{e^x + e^- x} \tag{4.9}$$

Wie in Abbildung 4.8 zu sehen, dient der *Hyperbeltangens* der Eingrenzung des Wertebereichs auf -1 bis 1.

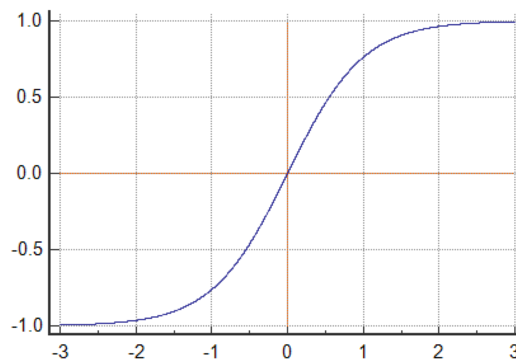


Abbildung 4.8: Grafische Visualisierung der hypberbolischen Tangente

4.4.4 Value-Head

Der *Value-Head* in *AlphaGo Zero* erhält als Eingabe das Ergebnis der letzten residualen Schicht. Daraufhin folgt ein *Convolutional Filter* der Größe 1, um die Dimension der erhaltenen Eingabe zu reduzieren. Notwendig ist dies, da *Convolutional Layers* mit mehrdimensionalen Eingabedaten arbeiten. Anschließend wird nach Durchqueren zweier *Fully-Connected Layers* ein Skalar v, der die aktuelle Erfolgswahrscheinlichkeit aus gegebener Position des Spiels (der Eingabe des neuronalen Netzes) wiedergibt, ausgegeben. Ein *Fully-Connected Layer* ist die traditionelle Struktur eines *neuronalen Netzes*, in dem die Neuronen mit allen Eingaben und Ausgaben verbunden sind.

4.4.5 Policy-Head

Der *Policy-Head* in *AlphaGo Zero* folgt dem Aufbau des *Value-Heads*. Dessen Ausgabe ist statt eines Skalars ein Vektor, dessen Größe von den möglichen Aktionen im gegebenem Spiel abhängt. Bei *Go* sind dies 19 x 19 + 1 Aktionen, bei *Vier-Gewinnt* hingegen 7 Aktionen.

4.4.6 Zusammenfassung

Zusammenfassend besteht das *neuronale Netz* in *AlphaGo Zero* aus einer Vielzahl *residualer Blöcke* und spaltet sich am Ende in einen *Value-Head* und einen *Policy-Head* auf. Durch die hohe Anzahl an Schichten können komplexe Spielzusammenhänge von dem

neuronalen Netz erfasst werden. Dabei dienen die *Convolutional Layers* dazu, Abhängigkeiten auf dem Spielbrett besser zu erfassen. Die aufgrund der verwendeten *residualen Blöcke* existierenden *Skip-Connections* führen dazu, dass ein stabiles Lernen in diesem Netz, trotz dessen Länge, möglich ist.

5 Implementation

Dieses Kapitel beschreibt die Implementation von *AlphaZero* für Vier-Gewinnt und vergleicht diese mit der in *AlphaGo Zero* beschriebenen Implementation. Dabei wird auf vier wesentliche Komponenten eingegangen: das Spielbrett und dessen Repräsentation, die Implementation der *Monte Carlo Tree-Search*, der Aufbau des neuronalen Netzes und die Konfiguration der Parameter. Das neuronale Netz der eigenen Implementation wurde auf der frei zugänglichen Plattform von Google, Google-Colab, mit der dort verfügbaren GPU trainiert[14].

5.1 Spielbrett

Da auf das Spielbrett häufig zugegriffen wird, hat dessen Repräsentation Einfluss auf die Performanz der gesamten Implementation. In jedem Schritt einer *MCTS* wird das Spielbrett immer wieder abgerufen, modifiziert und neu erzeugt. Zusätzlich dient es als Eingabe für das neuronale Netz.

Die häufigste Implementation von Vier-Gewinnt besteht aus einem zweidimensionalen Vektor, welcher die Zeilen und Spalten repräsentiert und somit jedes Feld in diesem Vektor ein Stein darstellt. So kann beispielsweise eine 0 für keinen Stein, eine 1 für einen Stein des ersten Spielers und eine 2 für einen Stein des zweiten Spielers stehen. Nach jedem getätigten Zug muss geprüft werden, ob ein Spielende erreicht wurde. Bei dieser Überprüfung werden in mehreren Schleifen sämtliche mögliche Kombinationen, welche zu einem Sieg führen, durchlaufen. Um zeitaufwendige Schleifendurchläufe zu umgehen, wird ein Bitboard als Repräsentation der Spielzustände genutzt. Abbildung 5.1 veranschaulicht eine mögliche Implementation eines Bitboards für Vier-Gewinnt. Die Implementation des Bitboards basiert auf den Erklärungen von Dominikus Herzberg[16], die sich auf *Fhourstones*[34], eine Implementation eines Bitboards von John Tromp, stützen. Ein Bitboard speichert die Repräsentation des Spielbretts für jeden Spieler mit dessen

```
  6 13 20 27 34 41 48   55 62     Zusätzliche Reihe
+---------------------+
| 5 12 19 26 33 40 47 | 54 61     Oberste Reihe
| 4 11 18 25 32 39 46 | 53 60
| 3 10 17 24 31 38 45 | 52 59
| 2  9 16 23 30 37 44 | 51 58
| 1  8 15 22 29 36 43 | 50 57
| 0  7 14 21 28 35 42 | 49 56 63  Unterste Reihe
+---------------------+
```

Abbildung 5.1: Visualisierung eines Bitboards

Quelle: https://github.com/denkspuren/BitboardC4/blob/master/BitboardDesign.md

Spielsteinen in einem *Long*-Datentyp. Ein *Long* besteht aus 64 *Bit* und wie Abbildung 5.1 zeigt, kann so jede mögliche Konstellation des Spielbretts durch eine binäre Repräsentation des *Longs* dargestellt werden. Anstatt in mehreren Schleifendurchläufen alle möglichen Kombinationen zu durchlaufen, um ein Spielende zu überprüfen, kann dies mit wenigen Bitoperationen überprüft werden[16]. Da die *Convolutional Layers* des neuronalen Netzes mit mehrdimensionalen Daten arbeiten, wird ein zusätzliches Spielbrett für das neuronale Netz parallel aktualisiert. Dieser zusätzliche Aufwand ist im Vergleich zu der eingesparten Zeit durch die Darstellung als Bitboard jedoch vernachlässigbar.

Der zu übergebene Spielzustand an das neuronale Netz in *AlphaGo Zero* besteht aus 17 Vektoren der Größe 19 x 19. Die Dimensionen 19 x 19 ist durch die Größe des Spielbretts in *Go* bedingt. Die 17 Vektoren setzen sich folgendermaßen zusammen: Die aktuelle Position der Spielsteine wird für jeden Spieler separat kodiert, sodass pro Zustand zwei Vektoren übergeben werden; einer für die Position der schwarzen Spielsteine und einer für die Positionen der weißen Spielsteine. Zusätzlich werden die sieben vorherigen Spielzustände (bezeichnet als *History Features*) mit übergeben. *History Featurs* sind notwendig, da der aktuelle Zustand alleine nicht ausreicht, um die Situation in Go wiederzugeben, da beispielsweise Wiederholungen verboten sind[32]. Der letzte der 17 Vektoren besteht gänzlich aus Einsen, wenn Schwarz am Zuge ist und gänzlich aus Nullen, wenn Weiß am Zuge ist. Dies wird als *Color Feature* bezeichnet. Das *Color Feature* ist ebenfalls notwendig, da eine weitere Regel in Go, das *komi*, nicht allein aus dem aktuellen Zustand bemessen werden kann[32]. Hingegen kann Vier-Gewinnt vollkommen aus dem aktuellen Zustand heraus beobachtet werden. Aus diesem Grund werden keine *History Features* an das neuronale Netz der eigenen Implementation übergeben. Ein *Color Feature* ist aus gleichem Grund überflüssig. Die Positionen der Spielsteine auf dem Brett werden in ei-

nem Vektor zusammengefasst. Anstatt in einem zusätzlichen Vektor die Information zu übergeben, welcher Spieler an der Reihe ist, werden die Eingaben an das neuronale Netz so übergeben, dass dies immer aus Sicht von Spieler 1 heraus agiert. Dies wird erreicht, indem nach jedem Zug die Kennzeichnung der Spielsteine umgedreht wird. Zusammenfassend führen diese Veränderungen dazu, dass an Stelle von 17 (19 x 19) Vektoren dem neuronalen Netz ein Vektor der Größe 6 x 7 übergeben wird.

5.2 Monte Carlo Tree-Search

Die in Kapitel 3 beschriebenen Statistiken werden in *AlphaGo Zero* in Kanten gespeichert. Im Gegensatz dazu wird in der selbst erstellen Implementation von Vier-Gewinnt auf das Erschaffen eigenständiger Objekte zur Speicherung der Statistiken verzichtet. Stattdessen werden die Statistiken direkt in den jeweiligen Knoten gespeichert. Dadurch wird die Anzahl zu erschaffender Objekte, welche im Speicher gehalten werden müssen, reduziert.

In *AlphaGo Zero* wird nach dem 30. Zug in einer Simulation der Temperaturparameter τ von $\tau = 1$ auf $\tau \rightarrow 0$ gesetzt. Ein τ nahe Null führt dazu, dass künftige Züge vollständig durch Exploitation ausgewählt werden. Eine Veränderung von τ nach einer bestimmten Anzahl an Zügen wurde vermutlich eingeführt, da ab einer gewissen Spiellänge weitere Exploration nicht zielführend wäre, sondern eher den Lerneffekt durch Selbstspiel verlangsamen würde. In Vier-Gewinnt hingegen ist die Auswahl an möglichen Zügen ab einer Spiellänge von 30 Zügen bereits sehr begrenzt. Um unerwünschte Effekte, wie das nicht Auffinden eines möglichen Gewinnzugs, zu vermeiden, wurde der Wert für τ dauerhaft bei Eins gelassen. Das frühzeitigere Herabsetzen von τ ist ohne weitere empirische Daten nicht empfehlenswert, um ungewollte Nebeneffekte zu verhindern. Weder zu *AlphaZero* noch zu *AlphaGo Zero* steht der Source-Code zur Verfügung. Aus den in *AlphaGo Zero* beschriebenen Mehtoden kann jedoch darauf geschlossen werden, dass die restliche Implementation von *MCTS* in *AlphaZero* der eigenen ähnelt.

5.3 Neuronales Netz

In *AlphaGo Zero* ist die meist verwendete Aktivierungsfunktion die *Rectified Linear Unit*. Ein mögliche Einschränkung der *ReLU* ist das *Dying ReLU Problem*, welches darauf hin-

weist, dass die *ReLU* bei sämtlichen Eingaben <= 0 Null ergibt. Dieses Verhalten kann dazu führen, dass gradientenbasierte Optimierung die Gewichte eines Neurons nicht anpasst, sofern dieses initial nicht aktiviert ist[25]. Während die *ReLU* definiert ist durch:

$$h^{(i)} = max(w^{(i)^T} x, 0 = \begin{cases} w^{(i)^T} x \; w^{(i)^T} x > 0 \\ 0 \quad else \end{cases} \tag{5.1}$$

ist die *Leaky ReLU* definiert durch:

$$h^{(i)} = max(w^{(i)^T} x, 0 = \begin{cases} w^{(i)^T} x \; w^{(i)^T} x > 0 \\ \alpha \, w^{(i)^T} x \quad else \end{cases} \tag{5.2}$$

α ist ein frei wählbarer Parameter, der in der Einführung der *LeakyReLU* mit 0.01 belegt ist. Das *Dying ReLU Problem* wird durch das Vorhandensein eines geringen Gradienten bei negativen Eingabewerten behoben.

Ein weiterer Unterschied zu *AlphaGo Zero* ist die Anzahl *residualer Schichten*. Es gibt keine festen Regeln, nach denen die optimale Anzahl an Schichten eines neuronalen Netzes für ein gegebenes Problem berechnet werden kann. Die richtige Auswahl der Struktur des neuronalen Netzes wird empirisch ermittelt und basiert häufig auf Erfahrungen und Heuristiken.

Für die Tiefe der Implementation des neuronalen Netzes für Vier-Gewinnt wurden mehrere Faktoren in Betracht gezogen. Betrachtet man die Struktur existierender neuronaler Netze, liegt die Schlussfolgerung nahe, dass mit steigender Komplexität der zu lösenden Aufgabe auch die Tiefe des Netzes steigt. Ein weiterer Faktor für die Tiefe eines Netzes ist die zur Verfügung stehende Rechenleistung. Das Prinzip neuronaler Netze ist nicht neu [30], aber deren Popularität hängt mit der erhöhten Rechenleistung und Entwicklung effizienterer Verfahren zusammen. Es wurden Versuche mit unterschiedlich tiefen Netzwerken durchgeführt, um einen Schätzwert für die Anzahl der benötigten Schichten zu erhalten. Die Anzahl der Schichten wurde auf Grund der zur Verfügung stehenden Hardware so gering wie möglich gehalten. Nach einigen Durchläufen der Versuche hat sich eine Anzahl von 4 *residualen Blöcken* als ausreichend erwiesen.

In *AlphaGo Zero* werden mehrere Netzwerke gleichzeitig trainiert und optimiert. Hierdurch ist eine Evaluation des besten Spielers durch das Duellieren der unterschiedlichen neuronalen Netze eine effektive Methode, um nur mit dem erfolgversprechendsten neuronalen Netz in die nächste Trainingsiteration zu gehen. In *Alpha Zero* wurde diese

Duell-Komponente entfernt und stattdessen ein Netz kontinuierlich verbessert. Da in der eigenen Implementation kein Mutli-Threading betrieben wird, wird ebenfalls auf die Duell-Komponente verzichtet.

5.4 Parameter

In *AlphaZero* wird zu den Wahrscheinlichkeiten P des Wurzelknotens die *Dirichlet Noise* (deutsch: Dirichlet-Verteilung als Störgröße) hinzugefügt. Die *Dirichlet-Verteilung* gibt die Wahrscheinlichkeit über das Auftreten von Wahrscheinlichkeitsverteilungen an. Die genaue Anpassung der Parameter in *AlphaGo Zero* wird definiert durch:

$$P(s,a) = (1-\epsilon)p_a + \epsilon\eta_a \tag{5.3}$$

ϵ wurde mit 0.25 und η mit $\eta \sim Dir(0.03)$ belegt. Das Hinzufügen der *Dirichlet Noise* soll eine Exploration aller Züge gewährleisten[32]. Ein $\eta < 1$ führt dazu, dass die *Dirichlet Noise* einen zufälligen Wert stark bevorzugt. Wiederum führt ein $\eta > 1$ zum gegenteiligen Effekt: kein Wert wird besonders bevorzugt. In *AlphaZero* wird erwähnt, dass die Größe der *Dirichlet Noise* antiproportional zu der geschätzten Anzahl an möglichen Zügen aus einer typischen Position heraus ist. Aus den in *AlphaZero* erwähnten Werten der *Dirichlet Noise* lässt sich eine Annäherung zur Bestimmung von $Dir(\alpha)$ berechnen, mit $\alpha = \frac{n}{10{,}6}$, wobei n für die Anzahl möglicher Züge aus gegebener Position steht. Daraus ergibt sich für Vier-Gewinnt mit einem $n = 7$ ein Wert von ~ 1.5. Da die *Dirichlet Noise* jedoch eingeführt wurde, um eine Auswahl aller möglichen Züge zu garantieren und ein $\alpha > 1$ einen gegenteiligen Effekt hat, wurde für Vier-Gewinnt ein Wert von 0.8 festgelegt. Somit wird garantiert, dass im Wurzelknoten ausreichend exploriert und zusätzlich die begrenzte Anzahl an Zügen berücksichtigt wird. Dies führt dazu, dass die *Dirichlet Noise* geringer ausfällt als in Spielen mit größeren Zustandsräumen. Um den Unterschied zwischen dem theoretisch errechnetem und tatsächlich verwendeten α zu reduzieren, wurde c_{puct}, ein weiterer Parameter zur Bestimmung des Grades der Exploration, von dem Standardwert $\sqrt{2}$ auf 2 erhöht.

Die Gewichte und der Bias des neuronale Netzes werden nach jeder Episode anhand bestehender Trainingsdaten angepasst. Eine Episode besteht dabei aus 100 Selbstspielen und jeder Zug innerhalb eines solchen Spiels generiert ein Datum. Neu erzeugte Trainingsdaten werden zu den bisherigen hinzugefügt. Sobald der Speicher der Google Colab Cloud voll ist, werden die ältesten Trainingsdaten verworfen.

5.5 Bewertung der Implementation

Zur Validierung der in diesem Kapitel beschriebenen Implementation wurden mehrere Spiele gegen einen Spieler, welcher zufällig Züge auswählt, gespielt. Die zum Zeitpunkt der Validierung neueste Version des neuronalen Netzes gewann 99% der Spiele. Eine Sättigung des Lerneffekts des neuronalen Netzes ist noch nicht beobachtbar. Niederlagen gegen einen zufälligen Spieler lassen sich in einem nicht zu Ende trainierten Netzwerk begründen. Durch die gegebene Zufallskomponente ist es auch möglich, dass mehrere aufeinanderfolgende optimale Züge von dem Zufallsspieler gewählt werden. Weiterführende Untersuchungen mit einem fortgeschritteneren Netzwerk sind daher notwendig. Im nachfolgenden Kapitel 6 wird hierauf näher eingegangen.

6 Diskussion

In der vorliegenden Arbeit wurde die Funktionsweise von *AlphaZero* anhand von Vier-Gewinnt beschrieben. Dabei wurde auf die wesentlichen Komponenten, die *AlphaZero* ausmachen, eingegangen. Besonderer Fokus lag hierbei auf dem Zusammenspiel von *Monte Carlo Tree-Search* und dem neuronalen Netz. Gezeigt wurde, wie das neuronale Netz die Suche der *MCTS* leitet und Rückgabewerte, an Stelle eines *rollouts*, approximiert. Die *MCTS* generiert im Gegenzug fundiertere Entscheidungen zur Auswahl des nächsten Zuges. Somit konnte veranschaulicht werden, wie sich die *MCTS* und das neuronale Netz gegenseitig verbessern.

Bei der Auswahl der Aktivierungs- und Optimierungsfunktionen wurde in *AlphaZero* ein konservativer Ansatz gewählt. In den letzten Jahren wurden diesbezüglich diverse neue Konzepte vorgeschlagen[27]. Nachfolgend wird auf zwei dieser Ansätze eingegangen, die für *Alpha Zero* effektiver sein könnten als die genutzten Funktionen. Eine ernstzunehmende alternative Aktivierungsfunktion an Stelle der verwendeten *ReLU* ist *Swish*[28]. *Swish* eignet sich besonders für tiefe neuronaler Netze. Mit der Verwendung von *Swish* konnten bekannte *deep neural networks*, die für *ImageNet*[9] konzipiert wurden, eine höhere Genauigkeit erzielen als mit den dort verwendeten Aktivierungsfunktionen. *ImageNet* repräsentiert eine große Datenbank aus Bildern, die zum Training neuronaler Netze verwendet wird. Die Struktur der neuronalen Netze, die für *ImageNet* entworfen sind und in [28] erwähnt wurden, ähnelt der von *AlphaZero*. Aus diesem Grund ist *Swish* eine erfolgversprechende Substitution für die *ReLU* und könnte in weiterführenden Arbeiten angewandt werden.

Einer der meist verwendeten *Optimizer* in bekannten neuronalen Netzen ist das *stochastische Gradientenverfahren*. Durch das *SGD* kann in neuronalen Netzen eine hohe Genauigkeit und *Generalisierung* erreicht werden. Im Kontext neuronaler Netze beschreibt *Generalisierung* bzw. *Verallgemeinerung*, wie gut ein trainiertes Netz mit ähnlichen, aber noch nicht bekannten Daten, umgeht. Der Nachteil des *SGD* ist, dass zum Erreichen hoher Genauigkeiten das Lernen sehr langsam voranschreitet. Aus diesem Grund wird als

Alternative häufig *Adam* verwendet, welcher ein schnelleres Lernen (vor allem zu Beginn) ermöglicht. Der Nachteil von *Adam* wiederum ist dessen schlechtere *Generalisierung*, weshalb häufiger in den bekannten neuronalen Netzen auf *SGD* zurückgegriffen wird, auch wenn das Training länger dauert. Neue, vielversprechende *Optimizer* sind *AdaBound* und *AmsBound* [24]. Diese haben - laut Autoren - einen schnelleren Lerneffekt als *SGD*, ohne dabei das Problem mangelnder *Generalisierung*, wie es bei *Adam* der Fall ist, aufzuweisen. Bei ausgewählten Benchmarks für neuronale Netze wurde sogar eine höhere Genauigkeit als mit *SGD* erreicht. Da *AdaBound* und *AmsBound* neuen Konzepten entsprechen, sind weitere empirische Untersuchungen notwendig, um deren Qualität zu bewerten. Die dazu bestehenden empirischen Daten legen nahe, dass *AdaBound* und *AmsBound* eine mögliche Alternative zum verwendeten *SGD* sind. Besonders für die eigene Implementation ist deren Verwendung aus mehreren Gründen interessant.
Ein Hauptgrund ist die Zeiteinsparung, da schneller Lerneffekte beobachtbar sind. Zum einen ist eine Sättigung des Lerneffekts bei tiefen Netzen, die unter begrenzten Ressourcen trainiert werden, kaum zu erwarten. Erst wenn eine hohe Genauigkeit erreicht werden muss, ist ein Rückgriff auf das *SGD*, welches sich bewährt hat, ein sicherer Ansatz, um die gewünschte Genauigkeit zu erreichen. Zum anderen lassen sich durch die Zeiteinsparung schneller unterschiedliche Architekturen vergleichen und somit wird ein effizienteres Erreichen verbesserter Parameter ermöglicht.

Ein weiterer Diskussionsgegenstand ist der aktuelle Trend neuronaler Netze, immer tiefer zu werden, welcher auf leistungsstärkere Hardware, optimierte Algorithmen und die Verwendung residualer Blöcke zurückzuführen ist. Es hat sich gezeigt, dass hauptsächlich die kurzen Pfade in residualen Netzen zu deren Erfolg führen[35]. Der Effekt kurzer Pfade wirft die Frage auf, inwiefern residuale Netze zu einer Lösung der auftretenden Problemen bei steigender Tiefe eines neuronalen Netzes beitragen, anstatt diese zu umgehen. Vielmehr wirft der Erfolg kurzer Pfade in *residualen Netzen* die Frage auf, ob die mit *residualen Blöcken* erreichte Tiefe überhaupt notwendig ist.

Schmidhuber kritisiert, dass residuale Netzwerke nicht mit zukünftigen Konzepten effizienter neuronaler Netzwerke, welche sich an biologischen neuronalen Netzen orientieren, kompatibel sind[30]. In biologischen neuronalen Netzen sind zu einem gegebenen Zeitpunkt nur wenige Neuronen aktiv. Die Neuronen sind in engem Raum stark miteinander vernetzt, in weiteren Distanzen hingegen spärlich vernetzt[30]. Diese beiden Eigenschaften werden von residualen Netzwerken nicht eingehalten, dort sind die meisten Neuronen aktiv und durch die *Skip-Connections* bestehen auch viele weitreichende Vernetzungen.

Die Modellierung effizienter dreidimensionaler Netzwerke mittels residualer Netze scheint daher nicht möglich.

Ein in dieser Arbeit nicht behandelter Aspekt ist der des Multi-Threadings. Wie genau ein Ablauf zu gestalten ist, um parallel eine Optimierung der Parameter des neuronalen Netzes, eine Evaluation der besten Netze und das Generieren neuer Trainingsdaten durch den besten Spieler zu gewährleisten, ist ein möglicher Untersuchungsgegenstand weiterführender Arbeiten.

Der neuartige Ansatz an *Alpha Zero* ist die Verwendung der *Monte Carlo Tree-Search* in Kombination mit einem neuronalen Netz. Die erreichten Leistungen fechten die bisherige Ansicht an, dass *Alpha-Beta-Suche* der beste Ansatz für die untersuchten Brettspiele ist[31]. In *AlphaZero* wird auf die Anzahl bewerteter Positionen pro Sekunde im Vergleich zu *Stockfish* und *Elmo* eingegangen, die bis zur Einführung von *AlphaZero* erfolgreichsten Programme. Letztere nutzen beide die *Alpha-Beta-Suche*. So erreicht *AlphaZero* eine Evaluation von 80.000 Positionen pro Sekunde in Schach, *Stockfish* hingegen 70 Millionen. Zusätzlich zeigt sich, dass mit mehr zur Verfügung stehender Zeit *AlphaZero* besser skaliert[31]. Ein Vorteil von *MCTS* gegenüber der *Alpha-Beta-Suche* ist, dass die während der Suche auftretenden Schätzungsungenauigkeiten sich gegenseitig aufheben, während ein Schätzungsfehler in der *Alpha-Beta-Suche* letztendlich als Resultat des *Minimax* bis in den Wurzelknoten zurückpropagiert wird[31].

Der Erfolg von *AlphaZero* ist in dem eben genannten Zusammenspiel von *MCTS* und neuronalen Netz zu begründen. Dennoch ist zu beachten, dass das erreichte Fertigkeitslevel in *AlphaZero* unter der Verwendung hoher Ressourcen erreicht wurde, so wurden z.B. für die Erstellung von Trainingsdaten 5.000 TPUs, *Tensorflow Processing Units*, Hardware welche speziell für den Bereich des *deep-learnings* konzipiert wurde, verwendet.

Wie in Abschnitt 5.5 beschrieben, konnte die eigene Implementation einen Spieler, welcher Züge zufällig auswählt, schlagen. Es ist jedoch fraglich, ob der erzielte Erfolg auf das in *AlphaZero* vorgestellte Konzept zurückzuführen ist. Jede Implementation eines Suchalgorithmus sollte einen Zufallsspieler schlagen können. Um den Effekt des neuronalen Netzes auf die *MCTS* zu messen, sind weitere Untersuchungen notwendig. Eine Möglichkeit besteht darin, die eigene Implementation gegen eine Implementation, die ausschließlich auf *MCTS* basiert, antreten zu lassen. Dabei sollte auch die Anzahl der durchzuführenden Simulationen der *MCTS* variieren, um den Effekt genauer messen zu können. Ein weiterer Ansatz wäre, die eigene Implementation ohne *MCTS* gegen einen Zufallsspieler antreten zu lassen. Dabei wird bei der eigenen Implementation der auszuwählende Zug durch den

Maximalwert der vom neuronalen Netz gegebenen Wahrscheinlichkeiten bestimmt. So wird eine genauere Unterscheidung zwischen dem Effekt des neuronalen Netzes und dem Effekt der *MCTS* in *AlphaZero* ermöglicht. Daraus schlussfolgernd sollten nachfolgende Arbeiten weitere empirische Daten erheben, um den Effekt von *AlphaZero* in einer Implementation präziser bewerten zu können. Gleiches trifft für die Wahl der Architektur und Parameter zu.

6.1 Fazit

AlphaZero hat mit einem neuartigen Konzept die bisher besten Suchalgorithmen für Brettspiele geschlagen. Dabei sind zwei Aspekte von *AlphaZero* revolutionär: Das Hinzuziehen eines neuronalen Netzes zur Positionsevaluation innerhalb der *Monte Carlo Tree-Search* und das Erlernen eines optimalen Spiels nur durch Selbstspiel, ohne das Hinzuziehen durch Menschen entwickelter Strategien. Demnach ist es denkbar, dass durch Modifikationen das Prinzip von *AlphaZero* auch auf andere Domänen angewandt werden kann. Zieht man die neusten Entwicklungen im Bereich des *deep-learnings* hinzu, kann mit weiteren interessanten neuen Konzepten gerechnet werden.

Literaturverzeichnis

[1] Abramson, Bruce: *The Expected-Outcome Model of Two-Player Games*, Columbia University, Dissertation, 1987

[2] Allis, Louis V.: *A Knowledge-based Approach of Connect-Four*, Vrije Universiteit, Dissertation, 1988

[3] Allis, Louis V.: *Searching for Solutions in Games and Artificial Intelligence*, University of Limburg, Maastricht, Dissertation, 1994

[4] Berry, D.A. ; Fristedt, B.: *Bandit Problems: Sequential Allocation of Experiments*. Chapman & Hall, 1985 (Monographs on Statistics and Applied Probability Series). – ISBN 9780412248108

[5] Browne, Cameron ; Powley, Edward ; Whitehouse, Daniel ; Lucas, Simon ; Cowling, Peter ; Rohlfshagen, Philipp ; Tavener, Stephen ; Perez Liebana, Diego ; Samothrakis, Spyridon ; Colton, Simon: A Survey of Monte Carlo Tree Search Methods. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4:1 (2012), 03, S. 1–43

[6] Chaslot, Guillaume ; Bakkers, Sander ; Szita, Istvan ; Spronck, Pieter: Montecarlo Tree Search: A New Framework for Game AI. In: *Proceedings of the Fourth AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, AAAI Press, 2008 (AIIDE'08), S. 216–217

[7] Chaslot, Guillaume M. J-B. ; Winands, Mark H. M. ; Herik, H. Jaap Van D. ; Uiterwijk, Jos W. H. M. ; Bouzy, Bruno: Progressive Strategies For Monte-Carlo Tree Search. In: *New Mathematics and Natural Computation (NMNC)* 4 (2008), Nr. 03, S. 343–357

[8] Coulom, Rémi: Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. In: Ciancarini, Paolo (Hrsg.) ; Herik, H. J. van den (Hrsg.): *5th International Conference on Computer and Games*. Turin, Italy, Mai 2006. – URL https://hal.inria.fr/inria-00116992

[9] Deng, J. ; Dong, W. ; Socher, R. ; Li, L. ; Kai Li ; Li Fei-Fei: ImageNet: A large-scale hierarchical image database. In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009, S. 248–255

[10] Gelly, Sylvain ; Silver, David: Combining Online and Offline Knowledge in UCT. In: *Proceedings of the 24th International Conference on Machine Learning*. New York, NY, USA : ACM, 2007 (ICML '07), S. 273–280. – ISBN 978-1-59593-793-3

[11] Gelly, Sylvain ; Silver, David: Monte-Carlo Tree Search and Rapid Action Value Estimation in Computer Go. In: *Artif. Intell.* 175 (2011), Juli, Nr. 11, S. 1856–1875. – ISSN 0004-3702

[12] Gittins, John ; Glazebrook, Kevin ; Weber, Richard.: *Multi-Armed Bandit Allocation Indices, 2nd Edition*. 2. Wiley, 2011

[13] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – URL http://www.deeplearningbook.org

[14] Google: *Colab System Specs*. – URL https://colab.research.google.com/drive/151805XTDg--dgHb3-AXJCpnWaqRhop_2#scrollTo=vEWe-FHNDY3E. – Zugriffsdatum: 2019-09-18

[15] He, K. ; Zhang, X. ; Ren, S. ; Sun, J.: Deep Residual Learning for Image Recognition. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, S. 770–778

[16] Herzberg, Dominikus: *Bitboards and Connect Four*. – URL https://github.com/denkspuren/BitboardC4/blob/master/BitboardDesign.md. – Zugriffsdatum: 2019-08-25

[17] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. In: *Proceedings of the 32Nd International Conference on International Conference on Machine Learning - Volume 37*, JMLR.org, 2015 (ICML'15), S. 448–456. – URL http://dl.acm.org/citation.cfm?id=3045118.3045167

[18] James, Steven: *The Effect of Simulation Bias on Action Selection In Monte Carlo Tree Search*, University of Witwatersrand, Diplomarbeit, 2016

[19] Kernbichler, Winfried ; Theis, Christian: Grundlagen der Monte Carlo Methoden. (2002). – URL https://itp.tugraz.at/MML/MonteCarlo/MCIntro.pdf. – Zugriffsdatum: 2019-09-3

[20] Knuth, Donald E. ; Moore, Ronald W.: An analysis of alpha-beta pruning. In: *Artificial Intelligence* 6 (1975), Nr. 4, S. 293 – 326. – ISSN 0004-3702

[21] Kocsis, Levente ; Szepesvári, Csaba: Bandit Based Monte-carlo Planning. In: *Proceedings of the 17th European Conference on Machine Learning*. Berlin, Heidelberg : Springer-Verlag, 2006 (ECML'06), S. 282–293. – ISBN 3-540-45375-X, 978-3-540-45375-8

[22] Krizhevsky, Alex ; Sutskever, Ilya ; Hinton, Geoffrey E.: ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F. (Hrsg.) ; Burges, C. J. C. (Hrsg.) ; Bottou, L. (Hrsg.) ; Weinberger, K. Q. (Hrsg.): *Advances in Neural Information Processing Systems 25*. Curran Associates, Inc., 2012, S. 1097–1105. – URL http://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks.pdf

[23] LeCun, Yann ; Bengio, Yoshua ; Hinton, Geoffrey: Deep Learning. In: *Nature* 521 (2015), 05, S. 436–44

[24] Luo, Liangchen ; Xiong, Yuanhao ; Liu, Yan: Adaptive Gradient Methods with Dynamic Bound of Learning Rate. In: *International Conference on Learning Representations*, URL https://openreview.net/forum?id=Bkg3g2R9FX, 2019

[25] Maas, Andrew L. ; Hannun, Awni Y. ; Ng, Andrew Y.: Rectifier nonlinearities improve neural network acoustic models. In: *in ICML Workshop on Deep Learning for Audio, Speech and Language Processing*, 2013

[26] Metropolis, N. ; Ulam, S.: The Monte Carlo Method. In: *Journal of the American Statistical Association* 44 (1949), Nr. 247, S. 335–341. – ISSN 01621459

[27] Nwankpa, Chigozie ; Ijomah, Winifred ; Gachagan, Anthony ; Marshall, Stephen: Activation Functions: Comparison of trends in Practice and Research for Deep Learning. In: *ArXiv* abs/1811.03378 (2018)

[28] Ramachandran, Prajit ; Zoph, Barret ; Le, Quoc V.: Searching for Activation Functions. In: *ArXiv* abs/1710.05941 (2017)

[29] RUSSELL, Stuart J. ; NORVIG, Peter: *Künstliche intelligenz*. 3. Pearson, 2012. – ISBN 978-3-86894-098-5

[30] SCHMIDHUBER, Jürgen: Deep learning in neural networks: An overview. In: *Neural Networks* 61 (2015), S. 85 – 117. – ISSN 0893-6080

[31] SILVER, David ; HUBERT, Thomas ; SCHRITTWIESER, Julian ; ANTONOGLOU, Ioannis ; LAI, Matthew ; GUEZ, Arthur ; LANCTOT, Marc ; SIFRE, Laurent ; KUMARAN, Dharshan ; GRAEPEL, Thore ; LILLICRAP, Timothy ; SIMONYAN, Karen ; HASSABIS, Demis: A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play. In: *Science* 362 (2018), 12, S. 1140–1144

[32] SILVER, David ; SCHRITTWIESER, Julian ; SIMONYAN, Karen ; ANTONOGLOU, Ioannis ; HUANG, Aja ; GUEZ, Arthur ; HUBERT, Thomas ; BAKER, Lucas ; LAI, Matthew ; BOLTON, Adrian ; CHEN, Yutian ; LILLICRAP, Timothy ; HUI, Fan ; SIFRE, Laurent ; DRIESSCHE, George van den ; GRAEPEL, Thore ; HASSABIS, Demis: Mastering the game of Go without human knowledge. In: *Nature* 550 (2017), 10, S. 354–359

[33] SUTTON, Richard S. ; BARTO, Andrew G.: *Reinforcement Learning: An Introduction*. The MIT Press, 2018. – URL http://incompleteideas.net/book/the-book-2nd.html

[34] TROMP, John: *The Fhourstones Benchmark*. – URL http://tromp.github.io/c4/fhour.html. – Zugriffsdatum: 2019-08-25

[35] VEIT, Andreas ; WILBER, Michael J. ; BELONGIE, Serge: Residual Networks Behave Like Ensembles of Relatively Shallow Networks. In: LEE, D. D. (Hrsg.) ; SUGIYAMA, M. (Hrsg.) ; LUXBURG, U. V. (Hrsg.) ; GUYON, I. (Hrsg.) ; GARNETT, R. (Hrsg.): *Advances in Neural Information Processing Systems 29*. Curran Associates, Inc., 2016, S. 550–558. – URL http://papers.nips.cc/paper/6556-residual-networks-behave-like-ensembles-of-relatively-shallow-networks.pdf

Glossar

Reinforcement Learning Reinforcement Learning ist das Erlernen der Verknüpfung von Situationen zu Handlungen, um eine numerischen Belohnung zu maximieren. Der Lernende muss selbst herausfinden, welche Handlungen zu den höchsten Belohnungen führen [33].

Spielbaum-Komplexität Die Spielbaum-Komplexität ist die Anzahl aller Blattknoten eines vollkommen ausgespannten Suchbaums [3].

Supervised Learning Supervised Learning ist das Lernen anhand von Trainingsdaten, die von einer wissenden Instanz erzeugt wurden. Die Wahl der richtigen Handlungen wird hier vom Trainingssatz vorgegeben [33].

Zustandsraum-Komplexität Die Zustandsraum-Komplexität ist eine Abschätzung aller legal zu erreichenden Zustände von der Startposition heraus. In Fällen, in denen diese nicht genau bestimmt werden kann, wird eine Obergrenze angegeben [3].

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „— bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] — ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen.“

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: ______________________________

Vorname: ______________________________

dass ich die vorliegende Bachelorarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Wenn die Maschine den Menschen schlägt - AlphaZero am Beispiel von 4-Gewinnt

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original