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

Stefan Göppert

16. August 2020

1 Einleitung

Im Oktober 2015 wurde zum ersten Mal ein Profi-Spieler im Brettspiel Go unter Turnierbedingungen von einem Computerprogramm geschlagen. Fan Hui, 2015 Europameister in Go, unterlag dem von Google Deepmind entwickelten Programm AlphaGo fünf zu null. Ein halbes Jahr später wurde auch der 18-fache Weltmeister Lee Sedol von AlphaGo vier zu eins geschlagen.[1] AlphaGo ist ein Meilenstein in der Entwicklung von Go-Computerprogrammen. Vor 2015 konnten sich Go-Programme nur auf kleineren Spielfeldern und mit zusätzlichen Steinen zu Beginn des Spiels mit guten Spielern messen.

Aufgrund des hohen Verzweigungsgrads und der Schwierigkeit Spielpositionen gut zu bewerten, stoßen traditionelle Suchverfahren wie die Alpha-Beta-Suche mit Go schnell an ihre Grenzen. Seit 2006 wurde daher für viele Go-Programme die Monte-Carlo-Baumsuche eingesetzt. Anstatt alle möglichen Spielpositionen auszuprobieren, wie es in der Alpha-Beta-Suche der Fall ist, werden in der Monte-Carlo-Baumsuche zufällige Spiele simuliert und das vielversprechendste Ergebnis weiterverfolgt.

AlphaGo setzt ebenfalls auf die Monte-Carlo-Baumsuche, ersetzt aber die Simulation von Zufallsspielen durch ein neuronales Netz. Der Einsatz von neuronalen Netzen und Deep (Reinforcement) Learning hat auch in anderen Spielen zu großem Erfolg geführt. In 2017 hat OpenAI mit einem Deep Learning Programm einen Profi-Spieler im Computerspiel Dota 2 besiegt und konnte in 2019 das weltbeste Dota 2-Team in fünf von fünf Spielen schlagen.[2] Und auch das von Google Deepmind entwickelte AlphaStar konnte in 2019 zwei Profi-Spieler im Computerspiel Starcraft 2 zehn zu null besiegen.

Einfachere Brettspiele wie Vier Gewinnt können effektiv mit einer Alpha-Beta-Suche gelöst werden, da ihr Verzweigungsgrad deutlich geringer ist als der von Go. Aber auch wenn der Aufwand verhältnismäßig gering ausfällt, kann ein moderner Computer immer noch einige Sekunden benötigen, um einen optimalen Spielzug zu bestimmen.

Auf der Online-Plattform Kaggle gibt es seit Januar 2020 einen neuen Wettbewerb, in dem Teilnehmer mit ihren Vier Gewinnt-Programmen um einen Platz auf der Rangliste kämpfen. In diesem Wettbewerb und im Rahmen der Limitierungen des Wettbewerbs soll das in dieser Arbeit vorgestellte Vier-Gewinnt-Programm möglichst gut abschneiden.

Wie gut kann die Monte-Carlo-Baumsuche ein Spiel wie Vier Gewinnt spielen und kann sie auch in diesem einfacheren Anwendungsfall von Deep Learning profitieren?

1.1 Aufbau der Arbeit

Nach einer Erklärung der Regeln von Vier Gewinnt und dem allgemeineren Spiel „Connect X“, wird der Begriff des Agenten definiert und die Online-Plattform Kaggle vorgestellt,

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

2. OpenAI u. a., „Dota 2 with Large Scale Deep Reinforcement Learning“, 13. Dezember 2019, besucht am 13. August 2020, arXiv: `1912.06680 [cs, stat]`, `http://arxiv.org/abs/1912.06680`.

sowie die Regeln des Wettbewerbs erklärt. In Kapitel 2 wird dann die Monte-Carlo-Baumsuche erklärt und einige Verbesserungen beschrieben, welche in Kapitel 3 implementiert werden. Kapitel 4 gibt einen Überblick über den Aufbau und die Funktionsweise von neuronalen Netzen und betrachtet eine besondere Form neuronaler Netze, die ConvNets. Im darauffolgenden Kapitel 5 werden dann zwei verschiedene Netzwerke implementiert und mit der Monte-Carlo-Baumsuche kombiniert.

Die Ergebnisse aus den Kapiteln 3 und 5 werden in Kapitel 6 gegenüber gestellt und verglichen, bevor in Kapitel 7 ein Fazit gezogen wird.

1.2 Das Spiel Vier Gewinnt

„Vier Gewinnt“ ist ein zwei Spieler Brettspiel das auf einem vertikal stehenden, hohlen rechteckigen Spielbrett gespielt wird. Das klassische Spielbrett hat sieben Spalten und sechs Zeilen. Beide Spieler haben zu Beginn des Spieles 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.

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.

Gelingt es keinem Spieler eine Viererreihe zu bilden bevor alle Spalten 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. In der englischen Literatur gibt es hierfür unterschiedliche Namen wie „ply“ oder „half-ply“ was wörtlich übersetzt Schicht bzw. Halbschicht bedeutet. Dabei ist in der Regel ein „ply“ die Kombination der Züge beider Spieler, und ein „half-ply“ ist der Zug eines einzelnen Spielers.

Vier Gewinnt 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.[3] Als Zufalls-freies Spiel mit perfekter Information ist „Vier Gewinnt“ ein lösbares Spiel und wurde bereits 1988 von Victor Allis[4] und unabhängig davon im selben Jahr von James D. Allen schwach gelöst.[5] Ein Spiel gilt als schwach oder stark gelöst, wenn ein realisierbarer Algorithmus existiert, mit dem für jede Startposition, bei 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).[6] Victor Allis hat mithilfe eines Computer-

3. *Kombinatorische Spieltheorie*, in *Wikipedia* (20. Dezember 2019), besucht am 14. August 2020, `https://de.wikipedia.org/w/index.php?title=Kombinatorische_Spieltheorie&oldid=195080333`.

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

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

6. *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`.

programms „VICTOR“ gezeigt, dass der erste Spieler bei perfektem Spiel immer gewinnt, wenn er den ersten Stein in die mittlere Spalte setzt, das Spiel mindestens unentschieden endet, wenn er in die Spalten direkt daneben setzt, und verliert, wenn er in einer der anderen Spalten beginnt.

1.3 Definition eines Agenten

Das Ziel dieser Bachelorarbeit ist es, einen Agenten für das Spiel Vier Gewinnt zu entwickeln. Wikipedia definiert einen Agenten wie folgt (Hervorhebungen meine):

> 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.[7]

Es ist also ein Computerprogramm, das abhängig von verschiedenen Zuständen ohne äußeres Einwirken (durch einen Benutzer) Entscheidungen trifft. Im bestärkenden Lernen, einem Teilgebiet des maschinellen Lernens, lernt ein Agent durch Interaktion mit einer Umgebung. Die Umgebung liefert den Zustand an den Agenten, welcher ausgehend von diesem Zustand eine Aktion wählt. Diese Aktion wiederum wird in die Umgebung gefüttert um einen neuen Folgezustand anzunehmen.

1.4 Kaggle

Kaggle ist eine Online-Plattform für Data-Science Experimente und Wettbewerbe. Inhalt dieser Arbeit ist der Kaggle Wettbewerb “Connect X”, welcher am 03. Januar 2020 gestartet ist. Die Herausforderung im Wettbewerb ist es, einen Spieler (Agent) hochzuladen, der “Connect X” spielen kann.

7. *Software-Agent*, in *Wikipedia* (6. Juli 2019), besucht am 14. August 2020, `https://de.wikipedia.org/w/index.php?title=Software-Agent&oldid=190180915`.

2 Die Monte-Carlo-Baumsuche

Die Monte-Carlo-Baumsuche (engl. Monte Carlo Tree Search, MCTS) wurde 2006 von Rémi Coulom erstmals vorgestellt und im Go-Programm Crazy Stone mit großem Erfolg eingesetzt.[8] Sie basiert auf der Kombination von Minimax-Spielbäumen mit Monte-Carlo-Simulationen.

2.1 Spielbäume

Ein Spiel kann formell mit den folgenden Elementen definiert werden:[9]

- 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 (engl. utility function) die einen numerischen Wert für einen terminalen Zustand s und einen Spieler p liefert. Der Wert ist typischerweise -1, 0 oder +1 für eine Niederlage, Unentschieden oder einen Sieg, kann aber auch 0, $+\frac{1}{2}$ und +1 sein.

Mit dem Ausgangszustand, der ACTIONS Funktion und der TRANSITION Funktion kann ein Spielbaum definiert werden. Ein Spielbaum (siehe Abb. 1) ist ein Baum in dem jeder Knoten einen **Zustand** des Spiels repräsentiert. Der Übergang von einem Knoten zu einem der **Kinder** ist ein **Zug**. Die Anzahl der Kinder eines Knotens ist auch als **Verzweigungsfaktor** bekannt. Der **Wurzelknoten** des Baumes ist der anfängliche Zustand des Spiels. Knoten ohne Kinder, in denen keine weiteren Züge möglich sind, sind **terminale Knoten**. Der Spielzustand in diesen terminalen Knoten kann bewertet werden um ein Spielergebnis zu erhalten.

Spielbäume sind rekursive Datenstrukturen. Wenn ein Zug gewählt wurde, kann der verbleibende Teilbaum, mit dem gewählten Folgezustand nun in der Wurzel, als neuer Spielbaum wiederverwendet werden.

Ein einfacher Algorithmus, der aber bei großen Spielbäumen aber schnell an seine Grenzen stößt, um den optimalen Spielzug in einem Spielbaum zu wählen ist der **Minimax-Algorithmus**.

8. 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, ISBN: 978-3-540-75537-1 978-3-540-75538-8, doi:10.1007/978-3-540-75538-8_7.

9. Stuart Russell und Peter Norvig, *Artificial Intelligence: A Modern Approach*, 3 edition (Upper Saddle River: Pearson, 11. Dezember 2009), S. 162, ISBN: 978-0-13-604259-4.

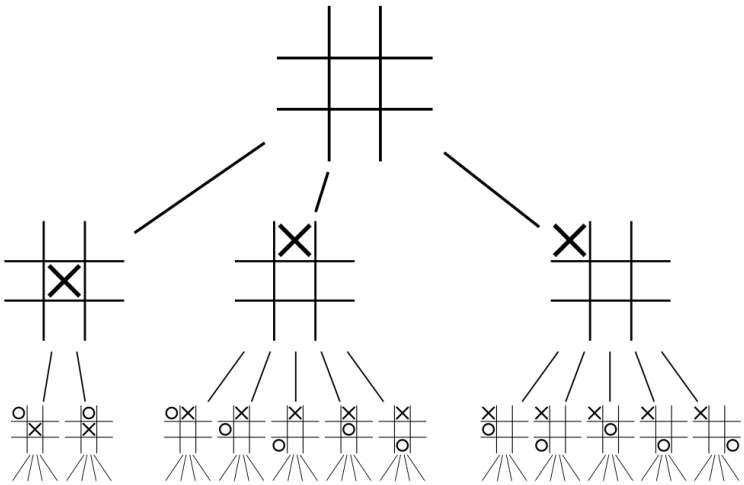

Abbildung 1: Ein Spielbaum mit den ersten zwei Zügen für Tic-Tac-Toe[10]

In einem Zwei-Spieler-Nullsummenspiel sind die Ziele beider Spieler exakt entgegengesetzt. Der eine Spieler versucht seinen (minimalen) Gewinn zu maximieren, während der Gegenspieler versucht, den (maximalen) Gewinn des ersten Spielers zu minimieren - daher der Name Minimax. Dies spiegelt sich auch im Minimax-Spielbaum wieder indem es Max-Knoten und Min-Knoten gibt. Bei Minimax wechseln sich die Knoten strikt ab, also Max-Knoten haben nur Min-Knoten als Kinder und umgekehrt.

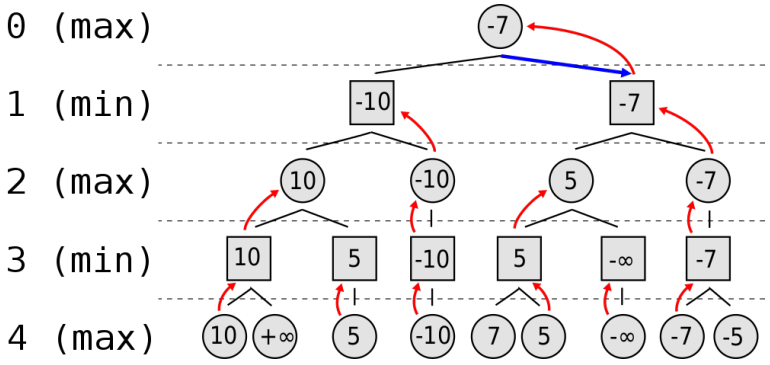

Abbildung 2: Ein Minimax-Spielbaum mit Kreisen für Max-Knoten und Quadraten für Min-Knoten. Die roten Pfeile sind die gewählten Züge in jedem Knoten und der blaue Pfeil ist der gewählte Zug im Wurzelknoten. Die Zahlen in den Knoten stellen den Wert der Knoten dar. [11]

10. 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`

Minimax muss den gesamten Spielbaum erforschen, bis im Wurzelknoten ein optimaler Zug gewählt werden kann, da jeder Knoten von der Bewertung seiner Kindknoten abhängig ist. Bei Spielen mit großem Verzweigungsfaktor wie Schach und Go führt dies aber zu gigantischen Spielbäumen, die einfach nicht komplett aufgebaut werden können.

Eine Lösung für dieses Problem ist es einfach die Suche ab einer gewissen Tiefe zu beenden und den aktuellen Zustand mit einer Bewertungsfunktion zu bewerten. Dies hat sich aber zum Beispiel für Go als sehr schwierig herausgestellt, da es schwer ist eine angemessene Bewertungsfunktion zu definieren.

Die Alpha-Beta-Suche ist eine Modifikation der Minimax-Baumsuche, welche versucht nur so viel vom Spielbaum zu durchsuchen, wie es nötig ist. Sobald festgestellt wird, dass ein weiteres Kind in einem Knoten das Ergebnis nicht verbessern kann, wird der gesamte Teilbaum „abgeschnitten“ und nicht weiter berücksichtigt.

2.2 Der MCTS-Algorithmus

Coulom kombinierte erstmals die Erzeugung eines Spielbaumes mit Monte-Carlo Simulationen. Unter Monte-Carlo Simulationen oder Monte-Carlo Experimenten versteht man gleichförmige Zufallsexperimente, die in großer Anzahl durchgeführt werden, um ein Ergebnis zu erhalten welches sich, mit steigender Anzahl der Experimente, dem tatsächlichen Wert immer weiter annähert.

Der von Coulom vorgestellt Algorithmus beginnt mit einem Knoten als Wurzel. Ausgehend von diesem Wurzelknoten werden nun zufällige Simulationen des Spiels gespielt. Das bedeutet in jedem Zustand wird ein zufälliger Zug gewählt und gespielt, bis das Spiel zu Ende ist. Der erste Zustand in einer solchen Simulation, der sich noch nicht im Spielbaum befindet, wird zum Baum hinzugefügt. Alle anderen Knoten, die sich bereits im Spielbaum befinden, merken sich, wie häufig sie bei solchen Simulationen besucht wurden und auch die Summe der Ergebnisse von Simulationen durch diesen Knoten. Die gesammelten Statistiken werden dann bei nachfolgenden Simulationen benutzt, um vielversprechende Knoten häufiger zu besuchen.

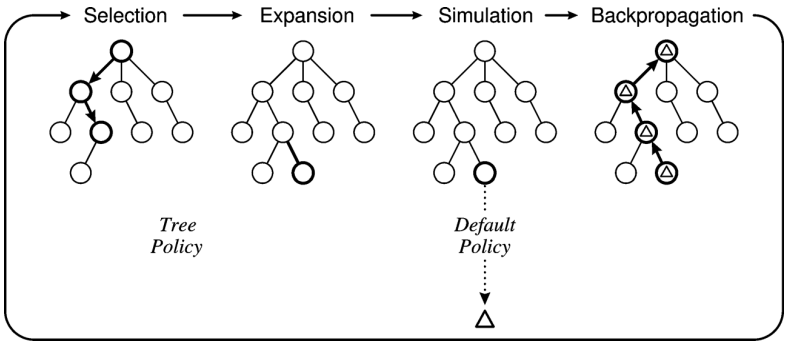

Abbildung 3: Eine Iteration des MCTS Algorithmus[12]

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

Eine Iteration der Monte-Carlo-Baumsuche läuft in vier Schritten ab (siehe Abb. 3). Zunächst wird der bestehende Spielbaum in einer **Selektions-Phase** durchlaufen, bis ein Blattknoten gefunden wurde. Dabei folgt der Algorithmus eine bestimmten Strategie (engl. policy) um die Knoten im Baum zu wählen. Ein Knoten ist ein Blatt des Baumes, wenn er terminal ist, oder noch nicht vollständig erforscht wurde, es also noch potentielle Kindknoten gibt, die noch nicht von diesem Knoten besucht wurden. Wenn dieser Blattknoten nicht terminal ist, so wird er **expandiert** indem ein noch nicht besuchtes Kind hinzugefügt und besucht wird. Ausgehend von diesem neuen Kindknoten wird nun eine **Simulation** gestartet. Es werden so lange zufällige Spielzüge ausgewählt bis das Spiel einen Endzustand erreicht hat. Die Strategie, die während der Simulation befolgt wird, nennt man auch die Default Policy. In der Regel werden die Spielzüge zufällig gewählt, es können aber Heuristiken benutzt werden um die Simulation zu steuern. Das Ergebnis dieser Simulation wird dann in der **Backup-Phase** benutzt, um die Bewertungen der in der Selektion besuchten Knoten zu aktualisieren.

Diese vier Schritte werden so lange wiederholt, bis ein Ressourcenlimit, zum Beispiel ein Zeitlimit, erreicht ist. Danach wird der beste Zug im Wurzelknoten ausgewählt. Die Expansion und Selektion lassen sich logisch noch zu einer Einheit als Tree Policy zusammenfassen. Der Ablauf ist in Algorithmus 1 als Pseudocode erklärt. Das Programm geht von einem Spielzustand s_0 im Wurzelknoten v_0 aus. TreePolicy folgt einer festen Strategie, um ein Blatt v_l im Baum zu finden, welches dann expandiert wird, wenn es möglich ist. SimuliereSpiel arbeitet mit dem Spielzustand im Blatt $s(v_l) = s_l$ und erzeugt daraus eine Bewertung r. Diese Bewertung wird durch Backup vom Blatt bis zur Wurzel propagiert und aktualisiert die Statistiken in den Knoten. Nach Ablauf eines Zeitlimits wird mit BestesKind der am besten bewertete Kindknoten im Wurzelknoten v_0 gewählt und die zugehörige Aktion $a(v)$ zurückgegeben.

Algorithmus 1 Allgemeiner MCTS Algorithmus[13]

```
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))
```

12. 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): S. 6, ISSN: 1943-0698, doi:10.1109/TCIAIG.2012.2186810

13. ebd., S. 5

2.3 Upper Confidence Bounds applied to Trees (UCT)

Parallel zur Arbeit von Coulom haben Kocsis und Szepesvári 2006 den **Upper Confidence Bounds applied to trees** (UCT)-Algorithmus[14] entwickelt. UCT gilt heute als beliebtester MCTS-Algorithmus. Die Besonderheit von UCT liegt in seiner Tree Policy und damit in der Art, wie die Knoten im Baum ausgewählt werden. Kocsis und Szepesvári betrachten dabei die Auswahl eines Kindknotens als ein sogenanntes Banditenproblem.[15]

Banditenprobleme sind eine Klasse von Problemen im bestärkenden Lernen, in denen der Agent zu jedem Zeitschritt t aus K Optionen wählen muss, um eine kumulative Belohnung zu maximieren, indem die optimale Aktion so oft 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.

Dieses Problem, auch bekannt als **exploration-exploitation-Dilemma**, ist ein Grundproblem vieler Algorithmen des bestärkenden Lernens. Die beste Strategie für das Banditenproblem ist die upper confidence bound (UCB) Regel **UCB1** von Auer et.al.[16] 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 $\sqrt{\frac{2 \ln n}{n_j}}$. 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 noch sehr unsicher. Wurde dagegen j sehr häufig gewählt, so ist dieser Anteil gering und somit drücken wir eine hohe Sicherheit in der Schätzung des Wertes $\bar{X}_j$ aus.

14. 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, ISBN: 978-3-540-46056-5, doi:10.1007/11871842_29.

15. 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., ISBN: 978-0-262-03924-6.

16. 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, ISSN: 1573-0565, besucht am 14. August 2020, doi:10.1023/A:1013689704352, https://doi.org/10.1023/A:1013689704352.

Im UCT-Algorithmus wird die Auswahl des Kindknotens zu einem Banditenproblem. Dafür hat jeder Knoten v eine Statistik $Q(v)$ mit der insgesamt erhaltene Belohnung in diesem Knoten und $N(v)$ mit der Anzahl der Besuche des Knotens. Das gewählte Kind ist dann jenes, welches die Formel

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

maximiert. Der Faktor $\sqrt{2}$ wurde aus der ursprünglichen UCB1-Formel (1) herausgezogen und als Parameter C_p variierbar gemacht. Er kann für jedes Problem optimiert werden und bestimme Verbesserungen funktionieren besser mit verschiedenen Werten von C_p. Algorithmus 2 zeigt den vollen Pseudocode des UCT Algorithmus.

Algorithmus 2 Upper Confidence Bound applied to Trees[17]

```
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:
23:    return argmax_{v' ∈ Kinder von v} Q(v')/N(v') + C_p sqrt(ln N(v)/N(v'))
24: function SimuliereSpiel(s)
25:    while s ist nicht terminal do
26:        wähle a ∈ A(s) zufällig
27:        s ← Transition(s, a)
28:    return Belohnung für s
```

Zusätzlich zu $Q(v)$ und $N(v)$ enthalten die Knoten noch Informationen über ihren Spielzustand $s(v)$ und die Aktion die in den Knoten geführt hat $a(v)$ sowie ein Verweis auf den Elternknoten. Die Backup-Regel ist unterschiedlich definiert für Ein-Spieler-Spiele und für Zwei-Spieler-Nullsummenspiele. Für Zwei-Spieler-Spiele in denen die Belohnungen für Spieler 1 gegenteilig zu den Belohnungen für Spieler 2 sind - +1 für Spieler 1 bedeutet −1 für Spieler 2 - kann die Belohnung des anderen Spielers durch Negierung in jedem Backup-Schritt erhalten werden.

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
```

Dieser normale UCT (engl. plain UCT) Algorithmus, wie er von Kocsis und Szepesvári vorgestellt wurde, wird in Kapitel 3 implementiert und als Ausgangslage für alle weiteren Verbesserungen genommen.

2.4 Verbesserungen

Die verschiedenen Schritte der Baumsuche können verändert und verbessert werden und werden dadurch besser für die jeweiligen Anwendungsfälle geeignet. Im Paper „A survey of monte carlo methods“ von Browne u.a.[18] wird eine große Menge von Verbesserungen vorgestellt und eine Übersicht darüber gegeben, für welche klassischen Spiele sie nützlich sein könnten. Im folgenden werde ich eine Handvoll ausgewählter Verbesserungen vorstellen.

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 (Abb. 4).

Eine andere Zugfolge, die zum selben Spielzustand führt, ist d1,d2;c1,e1. Durch diese unterschiedlichen Pfade werden die beiden identischen Zustände normalerweise als sepa-

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

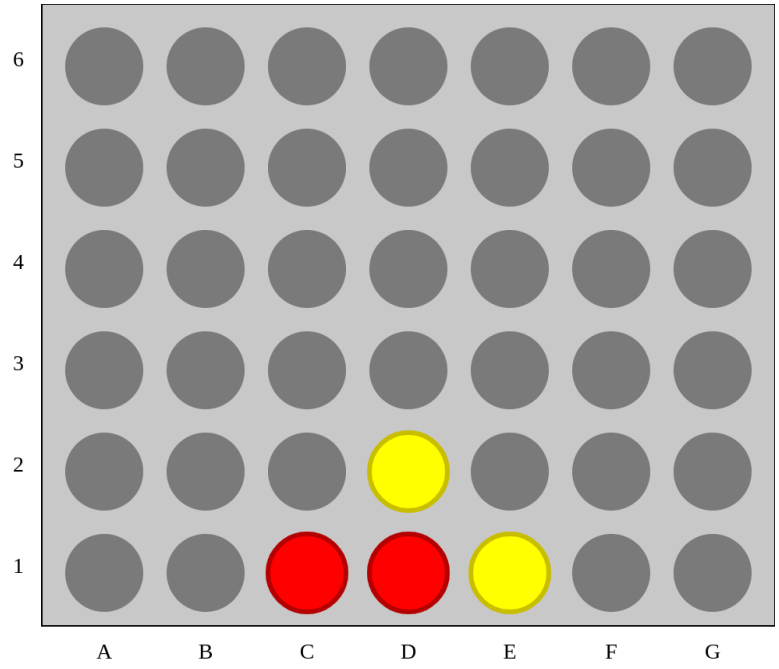

Abbildung 4: Die Spielposition kann auf mehrere Wegen erreicht werden. Die Züge d1,e1;c1,d2 sowie d1,d2;c1,e1 und c1,e1;d1,d2 führen alle zum selben Ergebnis.

rate Knoten mit separaten Statistiken gespeichert. Es würde aber sehr viel Sinn ergeben, diese Knoten zu kombinieren, denn der Weg, wie ein Zustand erreicht wird, hat keinen Einfluss darauf, welches die beste Aktion in diesem Zustand ist. Childs u.a. schlagen in ihrem Paper „Transpositions and Move Groups in Monte Carlo Tree Search“[19] drei Anpassungen des UCT-Algorithmus vor, um mit Transpositionen umzugehen.

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

In Gleichung 2 verwende ich die Statistiken eines Knotens $Q(v)$ und $N(v)$ direkt, also die Summe der Belohnungen aus Simulationen und die Anzahl an Simulationen durch diesen Knoten. Zusätzlich dazu kann diese Statistik abhängig von der gewählten Aktion a im Knoten v, $Q(v, a)$ und $N(v, a)$ gespeichert werden. Wenn keine Transpositionen identifiziert werden, so sind die Werte identisch $v' = \textsc{Transition}(s(v), a) \Rightarrow Q(v') = Q(v, a)$. Werden Transpositionen aber erkannt, kann es mehrere Aktionen aus verschiedenen Knoten geben, die zum selben Kindknoten führen.

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

19. 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), S. 390 f., doi:10.1109/CIG.2008.5035667.

allem den Vorteil, dass der gesamte Baum kleiner wird. Die von Childs u.a. vorgeschlagene einfache Auswahlregel UCT1 ist:

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

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 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. Für die Berechnung des Erkundungsfaktors

Sie können ignoriert werden (UCT0), sie können identifiziert werden und ihre Informationen die in der Selektionsphase notwendig sind, werden über alle identischen Knoten kumuliert (UCT1), anstatt der Information Q(s,a) wird die Information Q(g(s,a)) geteilt, wodurch die Informationen aus den Kindern eines Knotens verwendet werden anstatt der Informationen aus dem Elternknoten über diese Kinder. Dies verfeinert die Schätzung, da sowohl die Informationen aus Elternknoten A in diese Bewertung einfließen als auch die aus anderen Elternknoten B .. p (UCT2). Die letzte Variante UCT3 berechnet den Wert der Kinder Q(g(s,a)) rekursiv als gewichteter Durchschnitt aller Kinder des Knotens. Offen bleibt die Frage, wie die Backup-Phase des MCTS Algorithmus aussehen soll. Die vorgeschlagenen Vorgehensweisen sind "Update-All", welches ausgehend vom Blatt den Baum wieder hinaufsteigt und alle Vorfahren jedes Knotens aktualisiert, und "Update-Descend" welches nur die Knoten aktualisiert, die auf dem Abstiegspfad liegen.

Durch das Kombinieren der Knoten entsteht ein gerichteter azyklischer Graph. Es kann nützlich sein, zur Bewertung eines Knotens nicht nur die direkten Kinder und Eltern zu betrachten, sondern beliebig tief im Graph auf- und abzusteigen. Cazenave, Mehat und Saffidine schlagen einen parametrisierbaren UCT Algorithmus, den sie Upper confidence bound for rooted directed acyclic graphs (UCD) nennen, vor.

2.4.2 Scorebounded MCTS

Score bounded MCTS ist konzeptionell ähnlich zu Alpha-Beta Cuts in Minimax Suchalgorithmen. Jeder Knoten speichert eine optimistische und pessimistische Einschätzung. Ein Knoten gilt als gelöst, wenn er terminal ist, oder alle Kindknoten gelöst sind. Ein terminaler Knoten hat eine optimistische = pessimistische Grenze = tatsächliche Bewertung dieses Knotens.

Diese Grenzen können benutzt werden, um die Auswahl der Kindknoten zu steuern. Ein Kindknoten dessen optimistische Grenze schlechter ist, als die pessimistische Grenze des Elternknotens kann getrost ignoriert werden, da er zu keiner Verbesserung der Bewertung dieses Knotens beitragen kann.

Wenn sich die Bewertung eines Knotens ändert, muss die Änderung der pess. und opt. Grenzen den Baum hinauf propagiert werden. Dafür schlagen Cazenave und Saffidine zwei einfache Algorithmen vor.

2.4.3 Rapid Action Value Estimation

All Moves as First (AMAF) gehört zu den sogenannten History Heuristiken. Anhand der besuchten Knoten während der Selektion und Simulation werden andere Knoten im Baum oder die Simulationsphase beeinflusst. AMAF nimmt an, dass das Spielen eines Spielzuges X aus dem Zustand A einen ähnlichen Wert hat, wie wenn der Spielzug X aus dem Zustand A2 gespielt wird. Dies gilt sowohl für Spielzüge innerhalb des Suchbaumes als auch für Spielzüge, die in der Simulation gewählt werden.

Verschiedene History heuristic Verfahren unterscheiden sich darin, wie sie diese zusätzlichen Informationen verwenden. Move Average Sampling nutzt die Daten um den Simulationsschritt in der Default Policy zu steuern. Rapid Action Value Estimation (RAVE) nutzt die Daten zum Bootstrapping der Knoten. Die zusätzlichen Informationen geben bereits ein gewisses Vorwissen über die Werte eines Knotens, solange der Knoten noch nicht ausreichend oft besucht wurde.

Unterschieden wird hier zwischen Algorithmen, die die Bewertung der Knoten direkt verändern und solchen, die zusätzliche Informationen in den Knoten speichern.

3 Implementierung der Monte-Carlo-Baumsuche

4 Künstliche Neuronale Netze

Künstliche neuronale Netze (KNN), häufig auch nur neuronale Netze genannte, haben in den letzten Jahren das Forschungsfeld des maschinellen Lernens revolutioniert. Die Ideen hinter KNNs stammen aber bereits aus den 1940ern und 1950ern.McCulloch und Pitts haben 1943 ein Modell der Neuronen als Recheneinheit vorgestellt, 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. Eine Maschine, die binäre Inputs verarbeitet und einen binären Output erzeugt. Die Besonderheit des Perzeptrons war, dass es lernen konnte, seine Parameter anzupassen wenn der erzeugte Output falsch war - es konnte lernen. Das Perzeptron ist auch heute noch die Basis für die Einheiten, aus denen neuronale Netze bestehen.

Ein Perzeptron alleine kann einfache aussagenlogische Verknüpfungen wie AND, OR und NAND implementieren, scheitert aber an XOR und anderen sogenannten nicht linear separierbaren Problemen[20], dies haben Minsky und Papert in ihrem Buch Perceptrons 1969 gezeigt.

Dieses Problem wurde so schlimm von der Forschungsgemeinde aufgenommen, dass das Interesse an Perzeptrons verebbte.

Abhilfe schaffte der sogenannte Multi layer Perceptron (MLP), ein mehrschichtiges Netzwerk aus Perzeptronen, allgemeiner einfach nur Neuronen bezeichnet. Durch die Verknüpfung mehrerer Neuronen konnten 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 sogenannten Backpropagation Algorithmus mit Hilfe von automatischer Differenzierung möglich auch tiefere neuronale Netze zu trainieren.

Die Erfolge beim Kombinieren von Monte-Carlo Baumsuche mit neuronalen Netzen um Brettspiele wie Go, Shogi und Schach zu lernen von Silver et. al**silverMastering**, macht mich zuversichtlich, dass neuronale Netze auch Vier Gewinnt gut lernen können.

4.1 Neuronen

Neuronen sind kleine Einheiten die Inputs (**Features**) entgegennehmen und einen Output erzeugen. Sie unterscheiden sich vom Perzeptron dadurch, dass die Inputs und Outputs reelle Zahlen, in der Regel zwischen 0 und 1, sind. Jedes Neuron hat eine Menge von **Gewichten** w und einen **Bias-Wert** b. Die Gewichte werden verwendet um eine Gewichtete Summe der Eingangssignale zu berechnen. Der Bias-Wert dient dazu zu steuern, wie leicht oder schwer das Neuron „aktiviert“ wird, also wie leicht es ist einen positiven Output zu erzeugen. Man nennt die Gewichte zusammen mit dem Bias auch die **Parameter** eines Neurons. Ein Neuron mit 3 Inputs hat also 3 Gewichte und 1 Bias also 4 Parameter.

20. 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.

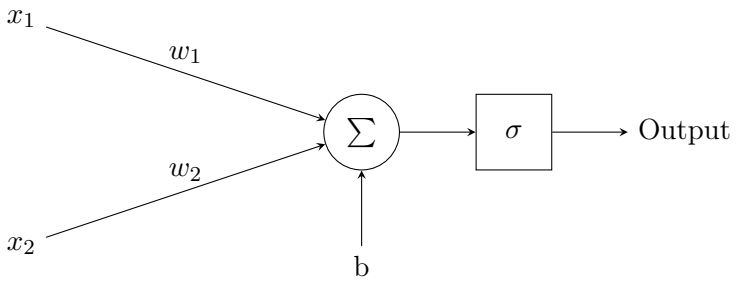

Abbildung 5: Neuron mit zwei Inputs und Bias

Der berechnete Wert des Neurons $z = x \cdot w + b$ wird dann durch eine **Aktivierungsfunktion** geschickt, um den Output des Neurons zu bestimmen. Im klassischen Perzeptron ist diese Aktivierungsfunktion die Stufenfunktion

$$STEP(z) = \begin{cases} 0 & z \leq 0 \\ 1 & z > 0 \end{cases}. \tag{4}$$

Die Stufenfunktion macht es aber schwer, die Parameter des Neurons präzise zu trainieren. Idealerweise führt eine kleine Anpassung der Gewichte zu einer kleinen Änderung des Outputs, die Stufenfunktion sorgt aber dafür dass die Änderung entweder nichts bewirkt oder eine große Änderung (± 1) auslöst.

Deswegen werden heute viele verschiedene Aktivierungsfunktionen eingesetzt. Die erste viel genutzte ist die Sigmoid Funktion

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{5}$$

Sie wird, genauso wie die Stufenfunktion, für große positive Werte 1 und für große negative Werte 0, dazwischen steigt sie aber stetig an. Sie hat somit das gewünschte Verhalten, dass kleine Veränderungen im Input nur einen kleinen Einfluss auf den Output haben.

Ein Neuron ohne Aktivierungsfunktion würde auch funktionieren, verliert dann aber kombiniert in einem Netzwerk seine Wirkung. Jedes Neuron für sich berechnet eine Linearkombination seiner Inputs. Eine Kombination dieser Neuronen 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 nicht-linear, wodurch sich jede beliebige Funktion approximieren lässt.

Es gibt noch viele weitere Aktivierungsfunktionen die für neuronale Netze benutzt werden. Nachfolgend ein paar Beispiele:

tanh Die Sigmoid Funktion ist eigentlich ein gestauchter und verschobener Tangens hyperbolicus $tanh(z) = \frac{e^z - e^{-z}}{e^z + e^{-z}}$. $tanh(z)$ hat daher eine ähnliche Form wie die Sigmoid Funktion, ihr Wertebereich ist aber $(-1, 1)$

ReLU ReLU steht für **Re**ctified **L**inear **U**nit und ist eine sehr einfache Funktion: $RELU(z) = max(0, z)$. Der Wert von ReLU ist 0 für alle $z < 0$ und sonst z. Diese Aktivierungsfunktion leidet an dem Problem, dass negative Werte und Werte nahe 0 dafür sorgen, dass das Neuron keine Aktivierung mehr hat, das Neuron „stirbt“.

Leaky ReLU Leaky ReLU versucht das Sterben des Neurons zu verhindern, indem auch der negative Teil eine kleine Steigung hat. $LEAKY(z) = max(\alpha z, z)$ mit einem kleinen α.

Die Aktivierungsfunktionen müssen differenzierbar sein, da die Ableitung für das Training im Backpropagation Algorithmus verwendet wird.

4.2 Architektur eines Neuronalen Netzes

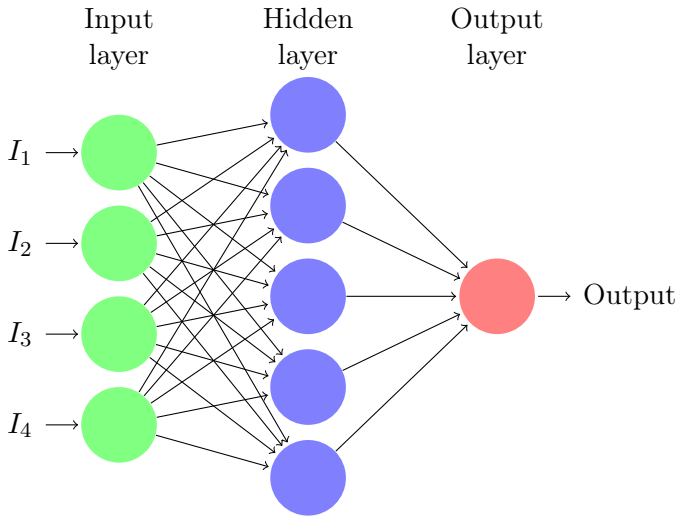

Abbildung 6: Ein neuronales Netz mit 2 Schichten, die Eingabeschicht wird nicht mitgezählt

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 reichen nur ihren jeweiligen Input an die nächste Schicht weiter. Generell wenn die Rede ist von einem neuronalen Netz mit n Schichten, so wird die Eingabeschicht nicht mitgezählt (siehe Abb. 6).

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 der Output. Generell 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 ist davon abhängig, welche Aufgabe das neuronale Netz lösen soll.

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 einfach die Aktivierung einer Schicht L mit Aktivierungsfunktion σ berechnen durch

$$a^L = \sigma(w^L \cdot a^{L-1} + b^L). \tag{6}$$

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 die Schicht.

Neuronale Netze werden für die verschiedensten Aufgaben verwendet, die Hauptanwendungsfälle lassen aber in Klassifizierung und Regression aufteilen.

Klassifizierung Das Ziel der Klassifizierung ist es, die Beispiele, die das System verarbeitet, in Klassen einzuteilen - es sollen diskrete Werte vorhergesagt werden. Unterschieden wird hierbei zwischen **binärer Klassifizierung**, **multi-label Klassifizierung** und **multi-class Klassifizierung**.
Die *binäre Klassifizierung* stellt die Frage, ob das Beispiel zur Klasse gehört oder nicht, der gewünschte Output also 1 ist.
Die *multi-label Klassifizierung* ist eine Art parallele binäre Klassifizierung. Es gibt mehrere Zielklassen und das Beispiel kann zu einer oder mehreren dieser Klassen gehören, entsprechend gibt es in diesem Fall so viele Outputs wie es Klassen gibt und die Outputs der Zielklassen müssen 1 werden.
Die *multi-class Klassifizierung* hat ebenfalls mehrere Zielklassen, allerdings gehört das Beispiel nur zu einer der Klassen.

Eine typische Aktivierungsfunktion für die Klassifizierung ist die Sigmoid Funktion. Für die multi-class Klassifizierung wird in der Regel die Softmax Funktion eingesetzt. Ihr Ergebnis ist eine Wahrscheinlichkeitsverteilung über alle Zielklassen die sich zu eins aufsummiert.

Regression 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 hat das neuronale Netz einen Output für jeden Zielwert, der vorhergesagt werden soll.

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

Ein neuronales Netz mit nur einer versteckten Schicht wird in der Literatur häufig als flaches Netzwerk und eines mit mehreren Schichten als tiefes Netzwerk (deep network) bezeichnet. Heutzutage sind Netzwerke mit 3 bis 10 Schichten nicht selten, weshalb der Begriff eines „tiefen Netzwerks“ etwas schwammig geworden ist.

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** (engl. epoch) 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. Also wie sehr das Netzwerk im Durchschnitt mit seinen Vorhersagen daneben lag. Die Fehlerfunktion (engl. cost function) muss abhängig von der Aufgabe und der Art der Trainingsdaten gewählt werden.

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

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{7}$$

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i| \tag{8}$$

Eine typische Fehlerfunktion für Klassifizierung ist die (binäre) Kreuzentropie (engl. cross entropy loss). Damit werden Wahrscheinlichkeitsverteilungen miteinander verglichen.

$$CrossEntropy = -\frac{1}{m}\sum_{i=1}^{m}[y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i)] \tag{9}$$

Der Fehler ist gering, wenn die Vorhersage sehr nahe an dem erwarteten Wert ist, und wird groß wenn die Vorhersage stark abweicht. Wichtig ist, dass diese Fehlerfunktionen den Fehler über den gesamten Trainingsdatensatz berechnen.

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. Effektiv ist die Fehlerfunktion also eine Funktion der Parameter des Netzwerks. Das Ziel ist es, diese Parameter so anzupassen, dass die Fehlerfunktion minimiert wird. Das verwendete Optimierungsverfahren ist auch als Gradientenverfahren (engl. gradient descent) bekannt.

4.3.2 Gradientenverfahren und Backpropagation

Stell dir vor du stehst auf einem Berg umrandet von Bäumen und willst das Tal erreichen, du siehst aber nicht wo genau es sich befindet. Der Boden unter deinen Füßen ist leicht geneigt. Solange du dich immer in Richtung der größten Steigung bewegst, wirst du irgendwann ein Tal erreichen. Nach diesem Prinzip funktioniert auch das Gradientenverfahren.

Der Berg, dessen Tal wir erreichen wollen, ist die Fehlerfunktion und die Steigung dieser Funktion wird durch die Gewichte des neuronalen Netzes bestimmt. Wenn wir für jedes Gewicht, also jede mögliche „Richtung“ in einem n-dimensionalen Raum, die Steigung an der aktuellen Position (mit den aktuellen Parametern) berechnen, so können wir alle Gewichte ein kleines bisschen verändern und uns dem Tal Schritt für Schritt nähern.

Die Steigung ist, wie aus der Analysis bekannt, die Ableitung der Funktion in Abhängigkeit vom Parameter $C'(x) = \frac{\delta C}{\delta x}$. Für jeden Parameter des Netzwerks muss diese partielle Ableitung berechnet werden. Ein Vektor mit allen diesen partiellen Ableitungen wird auch als Gradient bezeichnet $\nabla C(w_{1,1}, w_{2,1}, ..., w_{k,n}) = (\frac{\delta C}{\delta w_{1,1}}, \frac{\delta C}{\delta w_{2,1}}, ..., \frac{\delta C}{\delta w_{k,n}})^T$, daher der Name Gradientenverfahren.

Die Berechnung der Ableitungen geschieht im Backpropagation Algorithmus. Nachdem die Trainingsbeispiele durch das Netzwerk geführt wurden und der Fehler bestimmt wurde, wird jetzt der Fehler rückwärts durch das Netzwerk zurückgeführt. Zunächst wird für jeden Output bestimmt, welchen Einfluss er auf den Fehler hatte. Durch Anwendung der Kettenregel wird dann die Beteiligung jeder Verbindung - die Ableitung des Fehlerfunktion in Abhängigkeit vom Gewicht der Verbindung - der 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 bestimmt sind, kann nun ein Schritt des Gradientenverfahrens durchgeführt und die Parameter etwas verbessert werden.

Es gibt viele Optimierungen des Gradientenverfahrens François Chollet, *Deep Learning with Python*, 1st (Shelter Island, New York: Manning Publications, 22. Dezember 2017), Seite 50, ISBN: 978-1-61729-443-3. Eine der gängigsten Erweiterungen fügt ein Momentum hinzu, was konzeptionell ähnlich ist wie wenn nicht ein Mensch einen Berg hinabsteigt, sondern ein Ball hinab rollt. Der Ball bewegt sich noch ein bisschen in die Richtung des letzten Updates weiter, er verhält sich also träge. Dadurch kann der Algorithmus aus lokalen Minima hinausrollen und schafft es so mit größerer Wahrscheinlichkeit in ein globales Minimum. Andere Verfahren verändern die Lernrate des Algorithmus für individuelle Parameter.

4.4 Convolutional Neural Networks

Convolutional neural Networks (ConvNets) orientieren sich am menschlichen Sehen und der Funktionsweise des visuellen Kortex. In einem großen Bild, das das Auge sieht, hat jedes Neuron des visuellen Kortex nur ein kleines rezeptives Feld. Es ist nur Empfindlich auf Muster, die in diesem Feld auftauchen. Dieses Konzept wird durch Kernelfilter auf neuronale Netze übertragen. Kernelfilter finden unter anderem in der Bildbearbeitung und Signalverarbeitung Anwendung. So wird zum Beispiel die Schärfe und Unschärfe-Funktion eines Bildbearbeitungsprogramms durch einen Kernelfilter umgesetzt.

Ein Kernelfilter ist eine kleine quadratische Matrix, meistens 3x3 oder 5x5, die Pixel für Pixel über ein Bild geschoben wird. Für jeden Pixel werden alle umliegenden Pixel mit den Werten des Filters multipliziert und dann der Durchschnitt gebildet, um den neuen Wert für den mittleren Pixel zu bestimmen.

Klassische neuronale Netze mit einem Neuron für jedes Input Feature können auch auf Bilder angewandt werden indem für jedes Pixel und jeden Farbkanal ein Input-Neuron verwendet wird. Dadurch steigt allerdings die Größe des Netzwerks explosionsartig, gerade wenn Bilder mit Millionen von Pixeln verarbeitet werden. Ein solches neuronales Netz erkennt globale Muster in den Eingabedaten, so sind zum Beispiel die ersten 5 Neuronen der ersten verstecken Schicht aktiv, wenn ein bestimmtes Muster in der oberen linken Ecke auftaucht. Kernelfilter und Kernelfilter-Schichten suchen dagegen nach lokalen Mustern. Da der Filter über das gesamte Bild bewegt wird, ist er an allen Stellen aktiv, an denen das lokale Muster im Fenster auftaucht. Kernelfilter werden auch als Translationsinvariant bezeichnet.

5 Implementierung der Neuronalen Netze

6 Vergleiche und Ergebnisse

7 Fazit und Ausblick

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, 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. ISSN: 1573-0565, besucht am 14. August 2020. doi:10.1023/A:1013689704352. https://doi.org/10.1023/A:1013689704352.

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. ISSN: 1943-0698. doi:10.1109/TCIAIG.2012.2186810.

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. doi:10.1109/CIG.2008.5035667.

Chollet, François. *Deep Learning with Python*. 1st. Shelter Island, New York: Manning Publications, 22. Dezember 2017. ISBN: 978-1-61729-443-3.

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. ISBN: 978-3-540-75537-1 978-3-540-75538-8. doi: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.

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.

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.

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. ISBN: 978-3-540-46056-5. doi:10.1007/11871842_29.

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

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

OpenAI, Christopher Berner, Greg Brockman, Brooke Chan, Vicki Cheung, Przemysław Dębiak, Christy Dennison u. a. „Dota 2 with Large Scale Deep Reinforcement Learning“, 13. Dezember 2019. Besucht am 13. August 2020. arXiv: 1912.06680 [cs, stat]. http://arxiv.org/abs/1912.06680.

Russell, Stuart, und Peter Norvig. *Artificial Intelligence: A Modern Approach*. 3 edition. Upper Saddle River: Pearson, 11. Dezember 2009. ISBN: 978-0-13-604259-4.

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. ISBN: 978-0-262-03924-6.