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

Stefan Göppert

8. 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. 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. 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 die Online-Plattform Kaggle vorgestellt, sowie die Regeln des Wettbewerbs erklärt. Nach dieser Einführung werden die verschiedenen Verfahren, die zum Einsatz kommen, vorgestellt. Die Monte-Carlo-Baumsuche wird im Detail erklärt und Optimierungsansätze der Baumsuche betrachtet. Ein kurzer Überblick behandelt die wichtigsten Konzepte des maschinellen Lernens und stellt die beiden eingesetzten Netzwerktypen, MLP und ConvNet, vor.

Die darauf folgenden Abschnitte behandeln die Implementierung des Vier Gewinnt-

Programms. Nach einer Einführung in die Schnittstelle, die durch den Wettbewerb vorgegeben ist, wird zuerst eine einfache Monte-Carlo-Baumsuche ohne Verbesserungen implementiert und Schritt für Schritt erweitert. Die Baumsuche wird dann um neuronale Netze ergänzt. Die beiden ausgewählten Netzwerktypen werden miteinander verglichen und verschiedene Kombinationen aus Baumsuche und neuronalem Netz evaluiert. Zum Schluss werden die gesammelten Ergebnisse noch einmal gegenüber gestellt und ein bestes Programm gewählt, das für den Wettbewerb als finale Version eingereicht 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/ihr gelingt eine Viererreihe von Spielsteinen seiner/ihrer 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. Als Zufalls-freies Spiel mit perfekter Information ist "Vier Gewinnt" ein lösbares Spiel und wurde bereits 1988 von Victor Allis und unabhängig davon im selben Jahr von James D. Allen schwach gelöst. Victor Allis hat mithilfe eines Computerprogramms "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 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.

2 Monte-Carlo Baumsuche

In diesem Kapitel werde ich die theoretischen Hintergründe der Monte-Carlo Baumsuche erklären. Angefangen mit einem kurzen Einstieg in das Reinforcement-Learning und einer Einordnung in das größere Feld des maschinellen Lernens werde ich danach die in dieser Arbeit verwendete Notation erklären und die Monte-Carlo Baumsuche aus Reinforcement-Learning Sicht beschreiben. In diesem Zusammenhang wird auch das Exploration Exploitation Dilemma erklärt.

Die Monte-Carlo Baumsuche wurde 2006 von Coulom "erfunden". Er kombinierte erstmals die schrittweise Erstellung eines Spielbaumes mit Monte-Carlo Simulationen. Spielbäume finden schon seit Jahrzehnten Anwendung in der Spieltheorie und der Entwicklung von Algorithmen für Spiele.

2.1 Spielbäume

Brettspiele wie Vier Gewinnt, Schach und Go lassen sich mit Spielbäumen untersuchen und erklären. Die Knoten eines Spielbaumes sind die (legalen) Zustände des Spiels und die Kanten zwischen den Knoten sind die Aktionen, die ausgeführt werden können um von einem Zustand zu einem anderen Zustand zu wechseln. Die Wurzel des Spielbaumes ist der aktuelle Spielzustand. Hat ein Knoten keine hinausführenden Kanten, so ist er ein Blatt. Blätter sind in der Regel terminale Zustände und haben einen Wert - Sieg, Niederlage oder wenn es das Spiel erlaubt Unentschieden.

2.2 Die vier Schritte des Algorithmus

Die Monte-Carlo Baumsuche ist konzeptionell sehr einfach. Die Suche beginnt beim aktuellen Zustand, dem Wurzelknoten des Suchbaumes. Solange noch Ressourcen für die Berechnung vorhanden sind, werden Iterationen der Baumsuche durchgeführt. Nachdem die Ressourcen aufgebraucht sind, wird die beste Aktion, der beste Kindknoten, ausgewählt.
Eine einzelne Iteration der Baumsuche besteht aus den vier Schritten

- Selektion
- Expansion
- Simulation
- Aktualisierung

Ausgehend von der Wurzel wird der Suchbaum hinabgestiegen bis entweder ein terminaler Knoten, oder ein Knoten, der noch nicht vollständig expandiert ist, erreicht wurde. Das heißt, es wurden noch nicht alle Kinder dieses Knotens besucht. In jedem Knoten entscheidet die Tree policy, welcher Knoten als nächstes besucht werden soll.

Wenn ein nicht vollständig expandierter Knoten erreicht wurde, so wird dieser expandiert, indem ein noch nicht besuchtes Kind dem Suchbaum hinzugefügt wird, und mit diesem Knoten verbunden wird.

Danach wird von diesem Kind eine Simulation durchgeführt. In der Simulationsphase werden von der Default policy so lange Aktionen gewählt, bis das Spiel einen Endzustand erreicht hat. Das Ergebnis dieser Simulation wird danach verwendet, um die Werte der Knoten im Spielbaum zu aktualisieren.

Die Monte-Carlo Baumsuche erzeugt einen asymmetrischen Spielbaum. Knoten, die eine gute Bewertung haben werden häufiger besucht und der Baum in dieser Richtung mehr erweitert.

2.3 Tree policy

In jedem Knoten muss das Exploration Exploitation Dilemma gelöst werden. Der tatsächliche Wert eines Kindknotens ist nicht bekannt. Durch die Simulationen kann die Bewertung nur geschätzt werden. Der Algorithmus muss also eine Entscheidung mit unvollständigen Informationen treffen. Werden die Knoten gewählt, die in der Vergangenheit gute Ergebnisse geliefert haben oder werden Knoten ausgewählt, die mit den aktuellen Informationen zwar schlechter bewertet sind, deren geschätzter Wert aber noch sehr unsicher ist?

Die Tree policy versucht dieses Problem zu lösen. Das Problem gehört zur Klasse der Banditen-Probleme. Der Name stammt von den Slot-Maschinen, auch "einarmige Banditen" genannt, in Spielkasinos. Jeder "Arm" hat eine Wahrscheinlichkeit p einen Gewinn R auszuschütten, wenn er gezogen wird. In den theoretischen Banditen Problemen ist diese Wahrscheinlichkeit fest und dem Spieler unbekannt. Wie entscheidet der Spieler nun, welcher der beste Arm ist? Der naive Ansatz probiert zunächst jeden Arm einmal aus und merkt sich für jeden Arm die erhaltene Belohnung. Dann nimmt man immer den Arm mit der besten durchschnittlichen Belohnung, man ist gierig (engl. greedy).

Beispiel eines Banditenproblems: Bandit 1 (B1) gibt beim ersten Zug eine Belohnung von 0 und Bandit 2 (B2) eine Belohnung von 10. Die durchschnittlichen Belohnungen sind also entsprechend 0 und 10. Ein greedy Agent wählt jetzt so lange B2 bis die Bewertung von B2 geringer ist, als die von B1. Da die minimale Belohnung allerdings 0 ist, wird der Wert nie auf 0 sinken. Der Agent würde also immer den suboptimalen Arm wählen. Wir müssen ihn dazu bringen zu erkunden.

ϵ-greedy ϵ-greedy ist ein häufig gewählter Ansatz um dieses Problem zu lösen. Ein ϵ-greedy Algorithmus verhält sich greedy mit einer Wahrscheinlichkeit von ϵ und zufällig mit $1-\epsilon$. Die zufällige Auswahl ist Einheitlich aus allen vorhandenen Möglichkeiten, auch der greedy Option. Das bedeutet über eine Laufzeit von t Zeiteinheiten wird der Agent mindestens $(\epsilon + (1-\epsilon)/k) * t$ mal die greedy Option wählen und $((1-\epsilon) - (1-\epsilon)/k) * t$ eine rein zufällige andere.

Regret Unter Regret versteht man den Verlust, der dadurch entsteht, dass nicht die optimale Aktion in einem Zustand ausgewählt wird. Der Wert der optimalen Aktion v^* ist definiert als $v^* = q(r|a)$. Der Regret L_t ist der kumulierte Verlust über die gesamte Laufzeit des Agenten

$$L_t = \sum_i^t v^* - q(a_i) \tag{1}$$

Bei der Betrachtung des Regrets ist nicht der absolute Wert entscheidend, sondern sein Verhalten mit steigendem n. Deshalb ist die Big-O Notation nützlich um über das Wachstum des Regrets zu reden. Es wurde gezeigt, dass der Regret nicht langsamer als $O(\log n)$ wachsen kann. Ein epsilon-greedy Algorithmus hat einen linear wachsenden Regret $O(n)$.

Upper confidence bound Der Upper Confidence Bound Algorithmus versucht die Exploration von der Anzahl der durchgeführten Simulationen abhängig zu machen. Zur durchschnittlichen Bewertung eines Armes $Q(a_t)$ kommt noch ein Explorations-Term $U(a_t)$ der die Unsicherheit in der Genauigkeit des geschätzten Wertes angibt. Der Term wird so gewählt, dass $q(a_t) \leq Q(a_t) + U(a_t), \forall a_t$. Damit erhalten wir für jede Aktion $a_t \in A$ den optimistisch geschätzten Wert, der aufgrund unserer Unsicherheit höher liegen muss, als der tatsächliche Wert q(at).
Die UCB Formel wurde von XXX entwickelt.

$$UCB(a_t) = Q(a_t) + U(a_t) \tag{2}$$

mit

$$Q(a_t) = \text{durchschnittliche Belohnung erhalten wenn at gewählt} \tag{3}$$

und

$$U(a_t) = \sqrt{\frac{2\log(n)}{N(a_t)}} \tag{4}$$

was häufig allgemeiner als $U(a_t) = C_p \times \sqrt{\frac{\log(n)}{N(a_t)}}$ geschrieben wird wobei $n =$ die Anzahl aller Simulationen und $N(a_t) =$ die Anzahl der Simulationen, in denen at gewählt wurde, ist. Es wurde gezeigt, dass der Regret von UCB logarithmisch steigt.

UCT Tree Policy Die UCT Tree Policy ist die gängigste Tree Policy für die Monte-Carlo Baumsuche. UCT bedeutet Upper-confidence Bound for Trees und ist eine Adaption des UCB algorithmus (siehe Kap. X.X) für die Monte-Carlo Baumsuche.

$$UCT(s_t, a_t) = Q(s_t, a_t) + C_p \times \sqrt{\frac{\log(N(s_t))}{N(s_t, a_t)}} \tag{5}$$

Wobei $Q(s_t, a_t)$ der durchschnittliche Wert eines Knotens ist, C_p ist die UCB Explorations-Konstante, $N(s_t)$ ist die Anzahl der Besuche des Knotens s_t zum Zeitpunkt t und $N(s_t, a_t)$ ist die Anzahl wie häufig Aktion a_t im Zustand s_t gewählt wurde.

2.4 Default policy

Die Default Policy wird verwendet, um den Simulationsschritt der Baumsuche zu steuern. Die normale Policy wählt aus den möglichen Aktionen mit gleicher Wahrscheinlichkeit bis das Spiel zu Ende ist. Einige Personen haben gezeigt, dass eine Verbesserung der Default Policy eine beachtliche Steigerung der Effizienz der Baumsuche bewirken kann, andere haben aber gezeigt, dass in bestimmten Fällen eine zufällige default Policy die beste Wahl ist. Insbesondere, weil eine komplexere default Policy zu einer erheblichen Steigerung des Rechenaufwands pro Simulation führen kann.

Silver et al. haben in ihrem Paper über AlphaGo gezeigt, dass das komplette Ersetzen des Simulationsschrittes durch eine Auswertung durch ein neuronales Netz die Fähigkeiten eines Spielers basierend auf der Monte-Carlo Baumsuche im Brettspiel Go auf übermenschliches Niveau anheben kann.

3 Monte-Carlo Baumsuche Verbesserungen

Im Paper A Survey of MCTS methods wird eine Reihe von Verbesserungen der Baumsuche vorgestellt und ihre Anwendungsfälle für verschiedene klassische Brettspiele beschrieben. Im Folgenden werden ein paar dieser Verbesserungen aufgegriffen, die ich sinnvoll für Vier Gewinnt erachte und in dieser Bacherlorarbeit evaluieren möchte.

3.1 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.

Platzhalter Bild

Eine andere Zugfolge, die zum selben Spielzustand führt, ist d1,d2;c1,e1. Durch diese unterschiedlichen Pfade werden die beiden identischen Zustände normalerweise als separate Knoten mit separaten Statistiken gespeichert. Es würde aber sehr viel Sinn ergeben, diese Knoten zu kombinieren, denn der Weg wie ich zu einem Zustand gekommen bin hat keinen Einfluss darauf, welches die beste Aktion in diesem Zustand ist. Childs et al. schlagen drei Anpassungen der UCT Tree policy vor, um mit Transpositionen umzugehen. Sie können ignoriert werden (UCT0), sie können identifiziert werden und ihre Informationen die zur move selection 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.

3.2 All Moves as First(AMAF)

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.

3.3 Score bounded 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.

3.4 Temporal Difference Learning

Temporal difference learning ist der klassische Reinforcement Learning Ansatz der letzten Zeit. Das Ziel ist es, einen Q-Wert (State-Action Value) oder V-Wert (State-Value) zu lernen. Das Grundprinzip hinter TD Learning entstammt der Gleichung der Monte Carlo Methoden

$$V(S_t) \leftarrow V(S_t) + [G_t - V(S_t)] \tag{6}$$

wobei G_t die tatsächliche Belohnung nach Zeit t mit der konstanten Schrittgröße a ist. Am Ende einer Episode aktualisieren Monte Carlo Methoden den State-Value V(St)mit dem Monte-Carlo Fehler Gt-V(St). Monte-Carlo Methoden können Aktualisierungen erst am Ende einer Episode durchführen, weil erst dann G_t bekannt ist. Der Fehler berechnet wie weit die aktuelle Bewertung des Zustandes vom tatsächlichen Ergebnis der Simulation G_t abweicht und nähert sich dem Ergebnis G_t um einen Schritt alpha an.

Temporal Difference Methoden funktionieren ähnlich, müssen aber nicht warten bis die Episode vorbei ist. Im nächsten Zeitschritt $t+1$ können bereits die Bewertungen des

Zeitschritts t aktualisiert werden

$$V(S_t) \leftarrow V(S_t) + [R_{t+1} + V(S_{t+1}) - V(S_t)] \tag{7}$$

Diese Temporal Difference Methode ist auch als TD(0) oder one-step TD bekannt. TD(0) basiert die Aktualisierung des geschätzten Wertes auf bereits existierenden Schätzungen und wird dadurch als Bootstrapping Methode bezeichnet. Der Discountfaktor (0,1]kann benutzt werden, um Bewertungen aus näheren Zeitschritten höher zu gewichten ($\gamma \to 0$) oder alle Zeitschritte gleichmäßig zu beteiligen ($\gamma \to 1$).

Ansätze aus dem TD Learning können auch in den MCTS Backups Anwendung finden. Anstatt das tatsächliche Ergebnis einer Simulation den gesamten Baum vom Blatt zu propagieren, kann ein TD-Fehler verwendet werden. t=Rt+1+Q(St+1,at+1)-Q(St,at) Der Wert eines Knotens wird dann durch Q(St,at)Q(St,at)+t aktualisiert, wobei alpha im einfachen Fall =1N(S,a)entspricht.

Da bei klassischen Spielen wie Vier Gewinnt die Belohnung allerdings erst zum Ende eines Spiels bestimmt wird, setzt sich der TD-Fehler nur im Fall der Blattknoten wie oben beschrieben zusammen. Alle inneren Knoten aktualisieren sich relativ zu ihren Kindknoten
innent=Q(St+1,at+1)-Q(St,at)
Der Discountfaktor gamma steuert dabei wie sehr Knoten aktualisiert werden sollen, die sich weit vom Blattknoten entfernt befinden. Außerdem kann ebenfalls die Anzahl an Schritten während der Simulation benutzt werden, um die Belohnung Rt+1bereits im Blatt abhängig von der Länge der Simulation zu reduzieren. Wenn die Simulation in einem Blatt bereits nach 2 Schritten zu Ende ist, ist das Ergebnis vermutlich aussagekräftiger als wenn die Simulation erst nach 10 Schritten endet.

4 Kaggle Spielumgebung

Die Spielumgebung die von Kaggle zur Verfügung gestellt wird, orientiert sich in ihrer Programmierschnittstelle an der des OpenAI Gym. Das OpenAI Gym wurde entwickelt, um Forschung im Bereich des Reinforcement Learnings voranzutreiben. Insbesondere der Bedarf nach besseren Benchmarks und einheitlichen Umgebungen in veröffentlichten Papern waren ein Anreiz dafür.

Während andere Bereiche des maschinellen Lernens Benchmarks wie zum Beispiel ImageNet haben, gibt es kein äquivalent für reinforcement learning. Außerdem wird die Reproduzierbarkeit von veröffentlichten Forschungsergebnissen dadurch erschwert, dass es subtile Unterschiede in den verwendeten Umgebungen und Problemdefinitionen gibt, die die Schwierigkeit der Aufgabe drastisch verändern können.

Durch eine einheitliche Schnittstelle, die es leicht macht verschiedenste Umgebungen zu erzeugen und benutzen, wird die Entwicklung neuer Reinforcement learning Algorithmen deutlich vereinheitlicht.

4.1 Inhalt der Spielumgebung

Das Python Package `kaggle_environments`[1] liefert einige Spielumgebungen wie zum Beispiel TicTacToe, ConnectX und Halite mit einheitlicher Schnittstelle. Durch ein "make('ConnectX')" wird ein Objekt der Spielumgebung erzeugt.

Mit env.run([agent1, agent2]) kann das Environment ausgeführt werden, bis es terminiert. Die Methode gibt ein Array aller Zustände von Anfang bis Ende und die zugehörigen Belohnungen für beide Spieler zurück.

Mit env.render(mode) wird das Environment dargestellt. Der Mode gibt dabei an, wie die Darstellung geschieht. Dies kann z.B. interaktiv in einem ipython Notebook oder statisch auf einem Terminal geschehen.

Durch env.train([gegner, None]) kann eine Trainingsumgebung wie das OpenAI Gym erstellt werden. Durch das Keyword None wird angegeben, als welcher Spieler trainiert werden soll. Ein so erzeugter "trainer" kann dann genauso verwendet werden, wie es mit einem OpenAI Gym möglich ist. Durch reset() wird der Trainer initialisiert und zurückgesetzt, außerdem wird so die initiale Observation (Beobachtung) erstellt (siehe Kap. 3.2). Diese Observation wird vom zu trainierenden Agenten verwendet, um eine Entscheidung über die gewählte Aktion zu treffen. Durch Aufruf der Methode trainer.step(action) wird die Aktion durch die Umgebung verarbeitet und die nächste Observation, Belohnung, Spielende und weitere Informationen werden zurückgegeben. trainer.step() kann in einer Schleife aufgerufen werden, bis das Spielende erreicht ist.

4.2 Repräsentation des Spielfeldes

Die Observation der ConnectX Umgebung enthält zwei Elemente, das Spielfeld und die Markierung des ziehenden Spielers. Das Spielfeld wird als ein-dimensionales Array dargestellt. Eine 0 markiert einen freien Platz und eine Eins oder Zwei einen Spielstein

[1] https://github.com/Kaggle/kaggle-environments

des jeweiligen Spielers. Die Reihenfolge der Elemente im Array ergibt sich aus einer Nummerierung der Felder im Spielbrett von oben links nach unten rechts. Das heißt die erste(oberste) Zeile des Spielbretts (7 Felder im klassischen Spielbrett) sind auch gleich die ersten Felder in der Observation. Die letzte Zeile des Spielbretts sind die letzten Felder der Observation.

Diese Darstellung lässt sich leicht in eine zwei-dimensionale Darstellung umwandeln, um darauf Algorithmen durchzuführen, die die räumlichen Eigenschaften des Spieles ausnutzen.

5 Implementierung der Monte-Carlo Baumsuche

Die entwickelten Agenten werden mit der Programmiersprache Python entwickelt, da der Kaggle-Wettbewerb nur das Einreichen von Python-Code erlaubt und das Tooling für maschinelles Lernen für Python sehr umfangreich ist.

5.1 Überblick über die Klassen

Die Funktionalität des Agenten ist auf mehrere Klassen aufgeteilt.

5.1.1 ConnectFour

Diese Klasse kapselt den Spielzustand. Intern wird der Spielzustand als 1D-Array gespeichert. Ein Stein für Spieler 1 wird durch eine 1 und ein Stein von Spieler zwei durch eine 2 repräsentiert. Freie Felder werden durch eine 0 dargestellt.

Die notwendigen Funktionen um ein Spiel zu spielen sind

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

5.1.2 Node

Dies ist ein Knoten im Spielbaum. Unterschiedliche Implementationen speichern unterschiedliche Daten in den Knoten. Minimal muss aber eine Baumstruktur abgebildet werden können. Das heißt jeder Knoten kann Kinder und ein oder mehrere Elternknoten haben und muss einen Verweis auf diese Knoten speichern.

Alle Knoten erben von einer `AbstractTreeNode` Klasse, die eine Methode `best_child` vorgibt. Diese Methode soll den, abhängig vom Algorithmus, besten Kindknoten des aktuellen Knotens zurückgeben.

5.1.3 Player

Der `Player` ist die Haupt-Klasse jedes Agenten. Sie erben alle von der `AbstractPlayer` Klasse. Ein Aufruf der Methode `get_move(observation, configuration)` führt die Baumsuche aus, bis ein gesetztes Resourcenlimit erreicht ist, und gibt dann die Aktion des Agenten zurück.
Die Player-Klasse verwaltet einen Spielbaum aus `AbstractTreeNode` Knoten, der auch zwischen Spielzügen erhalten bleiben kann.

5.2 Normale Monte-Carlo Baumsuche

Die normale Baumsuche implementiert die MCTS ohne jegliche Verbesserungen. Jeder Knoten speichert seine eigenen Informationen, die Simulation geschieht durch eine zufällige Default-Policy und die Tree-Policy benutzt die UCT-Formel, um das beste Kind zu bestimmen.

Diese Implementierung dient als Vergleichswert zu allen weiteren Varianten und Verbesserungen der Baumsuche.

`Pseudocode für den normalen MCTS Algorithmus nochmal aufschreiben`

5.3 Transpositionen

Die Erweiterung der Baumsuche, sodass Transpositionen berücksichtigt werden ist relativ einfach. Wenn ein neuer Kindknoten erzeugt wird muss nur geprüft werden, ob er sich bereits in einer Transpositionstabelle befindet und wenn ja muss der Knoten aus der Tabelle verwendet werden, anstatt einen neuen anzulegen. Befindet er sich nicht in der Tabelle wird ein neuer angelegt und dieser wird in der Tabelle hinterlegt.

`Pseudocode für MCTS mit transposition`

Die Transpositionstabelle ist eine Hash-Map mit einem Hash des Spielzustandes als Schlüssel um dem Knoten selbst als Element.

Durch Einsatz der Transpositionstabelle teilen sich identische Knoten an unterschiedlichen Stellen im Baum ihre Statistiken. Dadurch wird es notwendig, die Statistiken eines Knotens in Abhängigkeit vom Elternknoten zu speichern, da sonst die UCT-Formel verfälscht wird (siehe Kapitel 3.1).

Der TranspositionPlayer implementiert mehrere Varianten des UCT Algorithmus mit Anpassungen an Transpositionen. Beim Erstellen eines Spieler-Objektes kann konfiguriert werden, welche Funktion verwendet werden soll.

6 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[2], 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[**?**], macht mich zuversichtlich, dass neuronale Netze auch Vier Gewinnt gut lernen können.

6.1 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.

[2] 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.

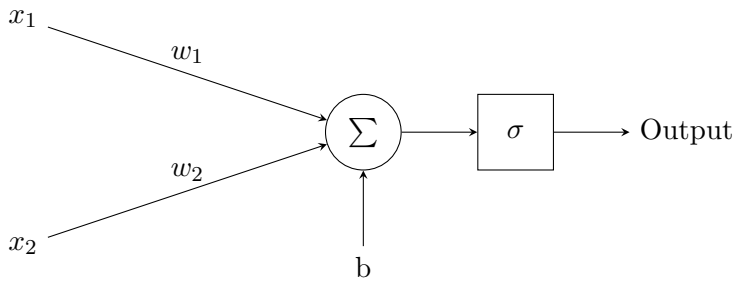

Abbildung 1: Neuron mit zwei Inputs und Bias

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

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

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

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

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.

6.2 Architektur eines Neuronalen Netzes

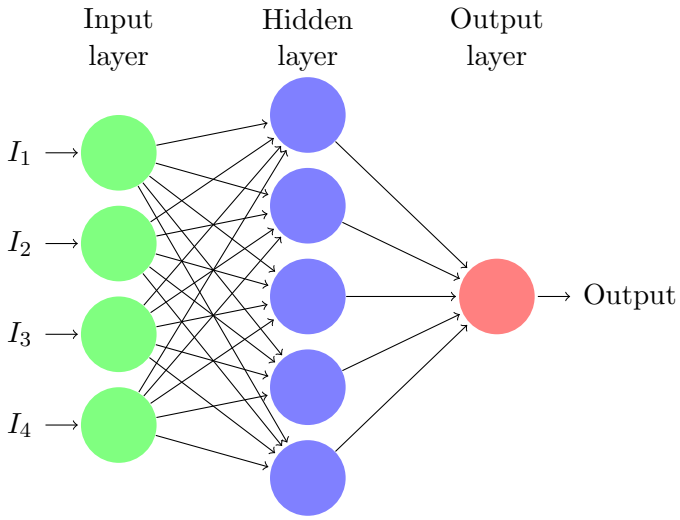

Abbildung 2: Ein neuronales Netz mit 2 Schichten, die Eingabeschicht wird nicht mitgezählt

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. 2).

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

$$a^L = \sigma(w^L \cdot a^{L-1} + b^L). \tag{10}$$

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.

6.3 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.

6.3.1 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.

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{11}$$

$$MAE = \frac{1}{m}\sum_{i=1}^{m}|y_i - \hat{y}_i| \tag{12}$$

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

$$CrossEntropy = -\frac{1}{m}\sum_{i=1}^{m}[y_i \ln(\hat{y}_i) + (1 - y_i)\ln(1 - \hat{y}_i)] \tag{13}$$

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.

6.3.2 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 [1, Seite 50]. 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.

6.4 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.

Literatur

[1] F. Chollet. *Deep Learning with Python*. Manning Publications, Shelter Island, New York, 1st edition, Dec. 2017.