Weiterentwicklung von neuronalen Netzwerken als KI für Computerspiele

STUDIENARBEIT

des Studienganges Informationstechnik

an der Dualen Hochschule Baden-Württemberg Ravensburg

von

Tobias Sulzmann, Nicole Graf, Colin Golfels

15.07.2019

Bearbeitungszeitraum	08.10.2018 - 15.07.2019
Matrikelnummer, Kurs	1484247, 3346427, 2674636, TIT16
Gutachter der Dualen Hochschule	Dipl.-Ing. (BA) Claudia Zinser, Prof. Dr. Axel Hoff

Erklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2015.

Ich versichere hiermit, dass ich meine Bachelorarbeit (bzw. Projektarbeit oder Seminararbeit) mit dem Thema:

Weiterentwicklung von neuronalen Netzwerken als KI für Computerspiele

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Friedrichshafen, den 15.07.2019

Tobias Sulzmann	Nicole Graf	Colin Golfels
Unterschrift	Unterschrift	Unterschrift

Inhalt

Abkürzungsverzeichnis

CNN ..... *Convolutional Neural Network*

LSB ..... *Least Significant Bit*

LSTM ..... *Long / Short Term Memory*

MCTS ..... *Monte Carlo Tree Search*

ResNets ..... *Residual Neural Networks*

UCB ..... *Upper Confidence Bound*

Formelverzeichnis

Abbildungsverzeichnis

Abstract

This project deals with the development of an artificial intelligence using a neural network and a Monte Carlo Tree. The main task was focused on the usage of the knowledge gained from a prior project called “Erprobung von neuronalen Netzwerken als KI für Computerspiele” to develop the game Five in a row. This knowledge should be used to create a computer opponent that uses the neural network to present a challenge for a human player.

In the beginning, the project was split between analyzing the status of the previous project and starting to gain the necessary knowledge about neural networks to continue. After the basic knowledge in the used programming language, python in this case, the theoretical basics of neural networks as well as the Monte Carlo Tree was established the coding of the software was started.

During this, no code from the previous work was used, because another goal was to create a more dynamic code. To simplify the project or to easier extend the source code, a structure of several agents was implemented. These agents can be different types of neural networks.

One of these agents was implemented as a combination of a feed-forward neural network with a Monte Carlo Tree. By playing against itself, the neural network learns and gets better. At the same time, it starts teaching the Monte Carlo Tree by sharing the best moves with it. When the Monte Carlo Tree is built up, giving him a fair share of possible best moves, the training can be stopped. Afterwards the Monte Carlo Tree as well as the neural network itself can be used as an opponent.

Zusammenfassung

Diese Projektarbeit behandelt die Entwicklung einer künstlichen Intelligenz, mithilfe eines neuronalen Netzwerks und eines Monte Carlo Baums. Kernaufgabe war, die gewonnen Erkenntnisse aus dem Projekt „Erprobung von neuronalen Netzwerken als KI für Computerspiele“ weiterzuverwenden und das Spiel Fünf in einer Reihe zu implementieren sowie einen Computergegner zu erstellen, welcher durch das neuronale Netz eine Herausforderung darstellt.

Zunächst wurde damit begonnen, den bisherigen Stand zu analysieren und die erforderlichen Kenntnisse in Bezug auf künstliche Intelligenzen zu erlernen. Nachdem diese Kenntnisse in der Programmiersprache Python, sowie den theoretischen Grundlagen zu neuronalen Netzwerken und dem Monte Carlo Baum erarbeitet wurden, ist damit begonnen worden, den Quellcode zu schreiben.

Dabei wurde kein Code aus der Vorarbeit genommen, da ein weiteres Ziel war, den Code dynamischer zu gestalten. Damit das Projekt bzw. der Quellcode einfach erweitert werden kann, wurde eine Struktur bestehend aus mehreren Agenten implementiert. Diese Agenten stellen verschiedene Arten von neuronalen Netzen dar.

Zu diesen neuronalen Netzen zählt die Implementierung eines Feed-Forward-Neural-Network in Kombination mit einem Monte Carlo Baum. Dabei lernt das neuronale Netz beim Spielen gegen sich selbst parallel den Baum mit den besten Zügen an. Wenn der Monte Carlo Baum mit genügend Zügen trainiert wurde, kann die Trainingsphase beendet werden. Anschließend kann dann neben dem neuronalen Netz auch der Monte Carlo Baum als künstliche Intelligenz als Gegner eingesetzt werden.

1 Einführung ins Thema

Spannend und vielseitig, aber auch gefährlich und unberechenbar, so würden wahrscheinlich die meisten Menschen die künstliche Intelligenz beschreiben. Seien es autonom fahrende Autos oder Assistenzsysteme der Medizin, der Einsatzbereich der künstlichen Intelligenz ist groß.

Ziel der meisten Entwicklungen in diesem Bereich der Technik ist vor allem die Unterstützung der Menschen. Seien es beim Auto beispielsweise Einparkhilfen oder autonomes Fahren selbst, können in der Medizin die Computer dabei helfen, Leben zu retten. Durch die Analyse und das Vergleichen der Symptome mit anderen Krankheitsbildern, kann ein Computer anhand einer großen Wissensbasis sinnvolle Entscheidungen treffen und so den Medizinern beim Feststellen der Krankheit helfen.

Aber auch in anderen Bereichen, wie der Spieleentwicklung können künstliche Intelligenzen verwendet werden. Bekannt sind vor allem die Computergegner aus klassischen Brettspielen wie Schach oder Dame. Hierbei wird die künstliche Intelligenz unter Verwendung verschiedenster Methoden so gut in einem Spiel, dass sogar Profispieler keine Chance mehr haben (Braun, 2018).

Betrachtet werden müssen jedoch immer beide Seiten der Medaille. Wo viele nur die Vorteile der künstlichen Intelligenz sehen, kann diese nicht alles problemlos lösen. Hierbei ist wichtig zu erwähnen, dass es sich bei einer künstlichen Intelligenz um von Menschen entwickelter Software handelt, die trainiert werden muss, um effektiv zu funktionieren. Sollten in den Trainingsdaten Fehler vorkommen oder die Trainingsmenge nicht ausreichen, kann dies dazu führen, dass die künstliche Intelligenz zu nicht vorhersagbaren Handlungen oder Entscheidungen kommt.

Betrachtet man die Historie der künstlichen Intelligenz, so beginnen die ersten Entwicklungsschritte um ca. 1950. Um diese Zeit wurden die ersten Programmiersprachen entwickelt und es wurden durch Experimente nachgewiesen, dass Computer in der Lage sind, das menschliche Gehirn nachzuahmen. Um 1970 investierten erstmals auch große Unternehmen in die Technologie, da diese sich einen Vorteil am Markt erhofften. Den großen Durchbruch schaffte die KI aber erst mit der Ära Social Media. Da der Mensch alleine nicht mehr in der Lage war, diese ganzen

Datenmengen zu verwalten und auszuwerten, wurden hierfür KIs erstellt (Calomme, kein Datum).

2 Aufgabenstellung

Als Aufgabenstellung wurde der Auftrag erteilt, sich in die Projektarbeit „Erprobung von neuronalen Netzwerken als KI für Computerspiele“ einzuarbeiten, um festzustellen an welchen Stellen das neuronale Netz, zum Lösen von Computerspielen, weiterentwickelt werden kann.

Nach Möglichkeit soll das neuronale Netz, wenn es weiterverwendet werden sollte, um das Spiel Fünf in einer Reihe erweitert werden. Hierfür soll auch der Ansatz einer Monte Carlo Tree Search untersucht und nach Möglichkeit in die Software implementiert werden.

Sollte es nicht möglich sein, Fünf in einer Reihe zu implementieren, wurde auch die Option offengelassen, ein beliebiges anderes Spiel in das neuronale Netz zu implementieren.

Die Entwicklungen sind zu dokumentieren und auf Schwierigkeiten und Probleme soll ebenfalls eingegangen werden. Das Ergebnis der Arbeit soll abschließend als Ausarbeitung der Prüfstelle und den Betreuern der Dualen Hochschule Baden-Württemberg vorgelegt werden.

3 Vorarbeiten

Als Vorarbeit für diese Studienarbeit dient die Ausarbeitung „Erprobung von neuronalen Netzwerken als KI für Computerspiele“. Diese Arbeit wurde 2018 von den ehemaligen Studenten Patrick Colbaut, Steffen Czymmeck und Christian Retzbach durchgeführt und beschäftigt sich mit der Erprobung von künstlicher Intelligenz im Bereich von Computerspielen.

Um sich mit dem Themenbereich der künstlichen Intelligenz vertraut zu machen, sollten neuronale Netze für die Spiele Tic-Tac-Toe und Vier Gewinnt erstellt werden. Ziel der Projektarbeit von den Herrn Colbaut, Czymmeck und Retzbach war es nicht, eine perfekte KI zu erstellen, sondern einen Einblick in diesen Bereich der Informatik zu bekommen. Ebenfalls sollten die aufgetretenen Probleme und Schwierigkeiten dokumentiert werden, um Grundlage für eine spätere Projektarbeit zu werden (Colbaut, Czymmeck, & Retzbach, Erprobung von neuronalen Netzwerken als KI für Computerspiele, 2018, S. 3).

Um die Künstliche Intelligenz und die beiden Spiele umzusetzen, wurde die Programmiersprache Python und das Framework TensorFlow verwendet.

Das Ergebnis der Studienarbeit „“Erprobung von neuronalen Netzwerken als KI für Computerspiele“ ist ein für Tic-Tac-Toe sehr gut spielendes Neuronales Netz. „*Selbst bei perfekter Spielweise eines menschlichen Spielers nach [10] konnte die selbstlernende künstliche Intelligenz nicht besiegt werden*“ (Colbaut, Czymmeck, & Retzbach, Erprobung von neuronalen Netzwerken als KI für Computerspiele, 2018, S. 42).

In dem Bereich des Vier Gewinnt wurden mittelmäßige Ergebnisse erzielt. Dieser Bereich der Studienarbeit wurde jedoch mit vielen Verbesserungsvorschlägen, wie zum Beispiel der Auswahl einer geeigneten Validierungsstrategie, dokumentiert.

4 Grundlagen

4.1 Neuronale Netze

Die Theorie hinter neuronalen Netzen ist deutlich älter als oftmals angenommen. Der Grundstein wurde bereits 1943 durch Warren McCulloch und Walter Pitts gelegt. In ihrem Artikel „A logical calculus of the ideas immanent in nervous activity“ stellten sie ihr Walter-Pitts-Neuron vor und bewiesen, dass auch einfache neuronale Netze im Prinzip jede arithmetische und logische Operation vornehmen konnten. (Gauglitz & Jürgens, kein Datum)

Neuronale Netze, im heutigen Sinne wurden ab ca. 1985 bekannt. Seitdem forschen immer mehr Wissenschaftler auf diesem Gebiet und es entwickelten sich einige renommierte Fachzeitschriften und Arbeitsgruppen die sich auf neuronale Netze spezialisiert haben.

Prinzipiell lassen sich neuronale Netze in verschiedene Kategorien einteilen. Das Grundkonzept bleibt allerdings das Gleiche.

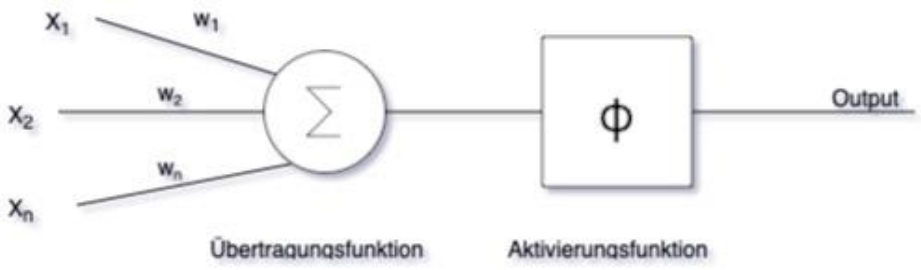

Abbildung 1: Aufbau eines Neurons

Jedes neuronale Netz besteht aus einzelnen Neuronen. Jedes Neuron besitzt ein oder mehrere Inputs (X_1-X_n) welche wiederrum mit Gewichten (w_1-w_n) versehen sind. Das Ergebnis der Übertragungsfunktion wird anhand der Eingaben und der Gewichte ermittelt und zum Schluss in eine Aktivierungsfunktion geleitet. Diese generiert dann den Output für dieses Neuron. (vgl. Abbildung 1)

In einem neuronalen Netz werden viele dieser Neuronen in mehreren Schichten zusammengeschalten. Unter dem Trainieren (vgl. Kapitel 4.1.6) des Netzes versteht man ein Anpassen der Gewichte zur Fehlerminimierung des Outputs.

Es gibt eine ganze Reihe an verschiedenen Netztypen. Im Folgenden sollen einige von ihnen näher beschrieben werden.

4.1.1 Feed-Forward Neural Network

Beim Feed-Forward Neural Network handelt es sich um eine Oberklasse an neuronalen Netzen. Die Besonderheit liegt bei diesen Netzen darin, dass die Struktur des Netzes keinen Zyklus ergibt, sondern die Eingaben stetig vom Eingang in Richtung Ausgang weitergereicht werden. (Zell, 1994)

Ein Austausch zwischen zwei Neuronen derselben Schicht findet nicht statt.

Im einfachsten Fall gibt es eine Eingabeschicht mit einem oder mehreren Neuronen, gefolgt von einer oder mehrerer sogenannter „Hidden Layer“. Daraus wird schließlich in der Output Schicht das finale Ergebnis ausgegeben. Die Durchlaufrichtung ist dabei von der Eingabeschicht in Richtung Ausgabeschicht. (vgl. Abbildung 2)

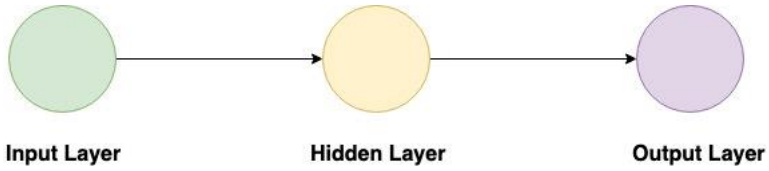

Abbildung 2: Einfachster Aufbau eines neuronalen Netzes

Trainiert wird in Feed-Forward Netzen meist über das Gradient Descent (vgl. Kapitel 4.1.6) Verfahren.

4.1.2 Recurrent Neural Network

Bei den Recurrent Neural Networks liegt die Besonderheit darin, dass auch innerhalb einer Schicht Verbindungen zwischen den Neuronen bestehen können. Die Rückkopplung kann dabei in unterschiedliche Kategorien eingeteilt werden.

Bei **direktem** Feedback verwendet ein Neuron seinen eigenen Ausgang als einen weiteren Eingang.

Bei **indirektem** Feedback wird der Ausgang eines Neurons als ein Eingang eines Neurons in einer vorhergehenden Schicht verwendet.

Beim **lateralen** Feedback wird ein Ausgang als Eingang eines anderen Neurons der gleichen Schicht verwendet.

Bei einer **vollständigen Verbindung** (Full Mesh) besitzt jedes Neuron eine Verbindung zu jedem anderen Neuron des neuronalen Netzes (Haselhuhn, 2018).

Dieser Typ eines neuronalen Netzes eignet sich gut zur Lösung von Problemen die von einer sequenziellen Verarbeitung profitieren. Zum Beispiel zur Schrifterkennung (Skymind Inc., kein Datum).

4.1.3 Long / Short Term Memory Networks

Bei diesen Netzwerken handelt es sich um eine Sonderform eines Recurrent Neural Networks. Bei herkömmlichen Netzwerken steigert sich bei zunehmender Tiefe das Problem des „vanishing gradients“ (vgl. Kapitel 4.1.7).

LSTM Module sollen dieses Problem beheben, indem sie statt einer einzigen Aktivierungsfunktion mehrere sogenannte Gates besitzen, welche jeweils über ihre eigenen Funktionen verfügen.

Zu diesen Gates gehören das Input Gate, welches steuert, inwieweit ein ankommender Wert Einfluss auf die bisherige Wertigkeit hat, sowie das Output Gate, das festlegt, welchen Einfluss der Wert eines Moduls auf dessen Output besitzt vgl. (Hochreiter & Schmidhuber, 1997).

Diese beiden Gates wurden später um Felix A. Gers Forget Gate erweitert, welches beschreibt, wann eine Information in einem Modul an Bedeutung verliert und daher „vergessen“ werden kann. (Gers, Schmidhuber, & Cummins, 1999, S. 16)

4.1.4 **Convolutional Neural Network**

Unter Convolutional Neural Networks (CNNs) versteht man neuronale Netze, in deren Hidden Layern eine „Faltung“ mit Hilfe eines Filterkerns durchgeführt wird. Durch diese Faltung können Muster in den Eingabedaten erkannt werden. Diese sogenannten Feature Sets werden auf Grund ihres explosionsartigem Größenanstieg im Vergleich zum Eingabevektor im Nachhinein oft in „Pooling-Layern“ zusammengefasst, um nur relevante Muster zu betrachten.

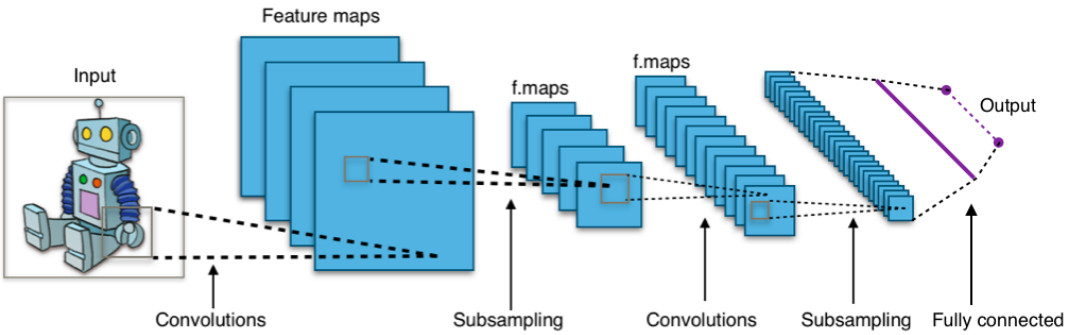

Abbildung 3: Convolutional Neural Network (Aphex34, 2015)

Neuronale Netze dieses Typs eignen sich im Wesentlichen zur Mustererkennung in zweidimensionalen Eingabewerten. Daher finden sie oft Einsatz in der Bilderkennung.

4.1.5 **Residual Neural Network**

Residual Neural Networks (ResNets) sind zumeist eine Sonderform von Feed-Forward-Netzen. Deren Spezialisierung darin besteht, das Vanishing-Gradient-Problem (vgl. Kapitel 4.1.7) zu lösen.

Erreicht wird dies dadurch, dass bei ResNets auf „Skip-Functions“ zurückgegriffen wird.

Konkret bedeutet das, dass bei der Backpropagation Teile des Netzes übersprungen werden können. Das erlaubt eine deutliche höhere Anzahl an Hidden Layern, welche allerdings bei der Backpropagation nicht alle einzeln berücksichtigt werden müssen.

4.1.6 **Gradientenabstieg**

Die Gradienten Abstiegsmethode versucht die Kostenfunktion eines Netzwerkes zu minimieren, um so ein akkurateres Netz zu erhalten. Bei diesem Verfahren vergleicht man die eingegeben Daten mit dem vom Netz berechneten Ergebnis.

Da bei einem neuronalen Netz die Neutronen in den Schichten mit den Neuronen der vorherigen und nachfolgenden Schicht verbunden sind, lassen sich die Gewichtungen an diesen Verbindungen anpassen. In Abbildung 4 wird beispielhaft ein neuronales Netz mit zwei Inputs, einem Hidden Layer und einem Output dargestellt.

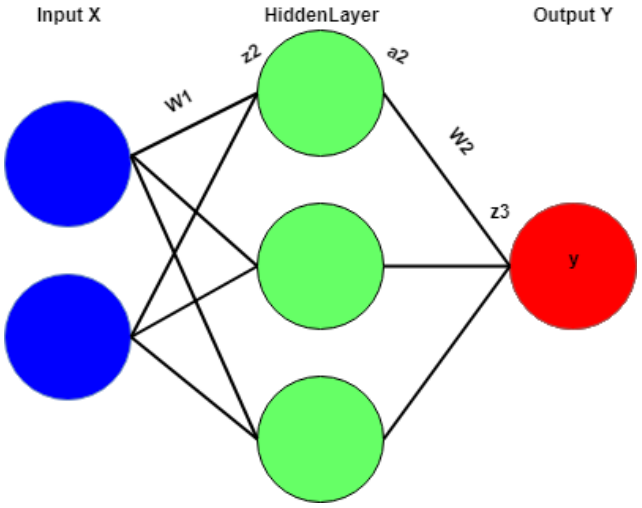

Abbildung 4: Beispiel Gradientenabstieg neuronales Netz

Um einen Wert für die Kostenfunktion zu bestimmen, wird der Durchschnitt aller Trainingsfehler ermittelt, da der Durchschnittswert vieler Fehler aussagekräftiger ist, als ein einzelner. Die Kostenfunktion indiziert bei einem hohen Wert, dass das Netz schlecht trainiert ist, da nicht das erwartete Ergebnis vom Netz ermittelt wurde. Ein niedriger Wert weist jedoch auf ein gut trainiertes Netz hin. Ziel dieser Methode ist es also das Ergebnis der Kostenfunktion auf einen möglichst niedrigen Wert zu bringen. Dieses wird durch das Anpassen der Gewichtungen im Netz erreicht, wobei jedes Gewicht zwischen zwei Neuronen in diese Berechnung einfließt.

Die Kostenfunktion für den Pfad, der im oberen Beispiel dargestellt ist, lautet wie folgt:

$$C = \sum \frac{1}{2} \cdot (y - f(f(X \cdot W_1) \cdot W_2))^2$$

Formel 1: Kostenfunktion

Der Algorithmus prüft nun, in wie weit sich durch die Veränderung der Gewichtungen, das Ergebnis der Kostenfunktion verändert und sucht nach dem Minimum. Bei nur einem Pfad kann es problematischer Weise dazu führen, dass der Algorithmus in einem lokalen Minimum endet und so nicht das globale Minimum eines Gewichtes findet. Betrachtet der Algorithmus jedoch mehrere Gewichte gleichzeitig, so führt durch die mehrdimensionale Ebene immer ein Weg zu dem kleinsten Ergebnis der Kostenfunktion.

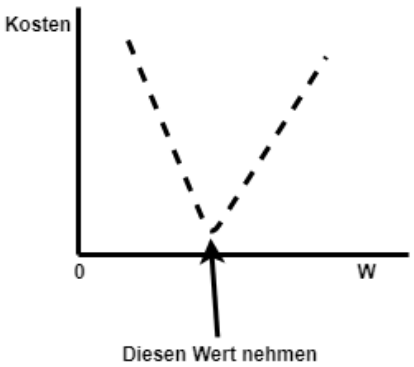

Abbildung 5: Darstellung der Kostenfunktion

In Abbildung 5 ist dargestellt, wie der Algorithmus bei einem Gewicht W die einzelnen Werte für dieses Gewicht prüft und ermittelt, an welcher Stelle der niedrigste Wert der Kostenfunktion vorhanden ist. In diesem Fall ist nur ein Minimum vorhanden, wodurch

der Algorithmus nicht in einem lokalen Minimum stecken bleiben kann (Neural Networks Demystified [Part 3: Gradient Descent], 2014).

4.1.7 Vanishing Gradient Problem bei neuronalen Netzen

Mit zunehmender Tiefe eines neuronalen Netzes steigt die Gefahr des „vanishing gradient“. Damit ist gemeint, dass bei der Backpropagation die Gewichte der Fehlerfunktion aufmultipliziert werden. Bei Fehlerwerten zwischen null und Eins nähert sich daher der Gradient immer weiter an null an, je weiter man bei der Backpropagation von der Ausgabeschicht entfernt ist.

Die Folge dieses Problems bei tieferen Netzen ist, dass sich die Gewichte der Anfangsschichten nicht mehr bzw. in zu geringem Maße verändert werden und das neuronale Netz auch nach einem langen Lernprozess keine richtigen Ausgabedaten produziert.

4.2 Monte Carlo Tree Search

Monte Carlo Tree Search (kurz MCTS) ist ein neues Verfahren welches bekannt wurde, (Chaslot, Bakkes, Szita, & Spronck, 2008) als Googles DeepMind (eine künstliche Intelligenz) den Weltsieger im asiatischen Brettspiel Go klar besiegte (Silver, Schrittwieser, Simonyan, Antonoglou, & et al., 2017). Das Spiel galt bis dahin als die größte Hürde für ein rechnerbasiertes System, da es deutlich mehr Spielzüge und Möglichkeiten als Schach besitzt (Coulom, 2006). MCTS nutzt stochastische Simulationen, um bei einem finiten Spiel die beste Spieltechnik zu ermitteln.

Der Algorithmus besteht aus vier Phasen, durch die ein Entscheidungsbaum aufgebaut wird.

Die erste Phase ist die Selektionsphase. Bei einem existierenden Baum wird der momentane Knoten auf Kindknoten überprüft. In dieser Phase wird zwischen oft besuchten und weniger besuchten Knoten entschieden. Dazu wird der Upper Confidence Bound (kurz UCB) verwendet. Der UCB berechnet für jeden Knoten einen Confidence (dt. Vertrauen) Wert bezüglich der Gewinnchance und gibt diesen Wert dann zurück (Peng, 2018). Dadurch wird das Exploration-Exploitation System umgesetzt. Bei diesem System wird eine Balance zwischen dem Verwenden bereits bekannter Knoten und deren Gewinnchance (Exploitation, dt. Ausnutzung) sowie dem

Verwenden neuer Knoten, bzw. nicht so oft verwendeter Knoten (Exploration, dt. Erkundung) gefunden. Um diese Balance zu schaffen, wird Formel 2 verwendet.

$$\frac{w_i}{n_i} + c\sqrt{\frac{\ln t}{n_i}}$$

Formel 2: Auswahlformel für den Monte Carlo Tree Search Algorithmus

Der erste Term teilt die Anzahl der gewonnen Züge durch die Anzahl der durchlaufenen Simulationen im i-ten Durchlauf. C ist ein Explorationsfaktor. Durch ihn kann eingestellt werden, ob die Funktion die Tendenz zur Erkundung neuer Züge oder zur Ausnutzung bekannter Züge hat. T ist die Gesamtzahl der durchlaufenen Simulationen (Levente Kocsis, 2006).

Jeder Knoten besitzt einen UCB Wert. Der Kindknoten mit dem höchsten UCB Wert wird ausgewählt. Dies wird so lange wiederholt, bis es keine Kindknoten mehr gibt.

Wird ein Blattknoten aufgefunden, wird die nächste Phase des MCTS Algorithmus eingeleitet.

In Abbildung 6 ist beispielhaft ein Entscheidungsbaum aufgeführt. Die erste Zahl in dem Knoten ist die Anzahl der Siege, die bei Besuch des Knoten aufgetreten sind, die rechte Zahl ist die Gesamtzahl der Besuche. In der Expansionsphase wird nun Formel 2 verwendet, um den passenden Knoten für den nächsten Zug auszuwählen. Hier wird dem Baum ein neuer Knoten mit einem neuen Zug hinzugefügt.

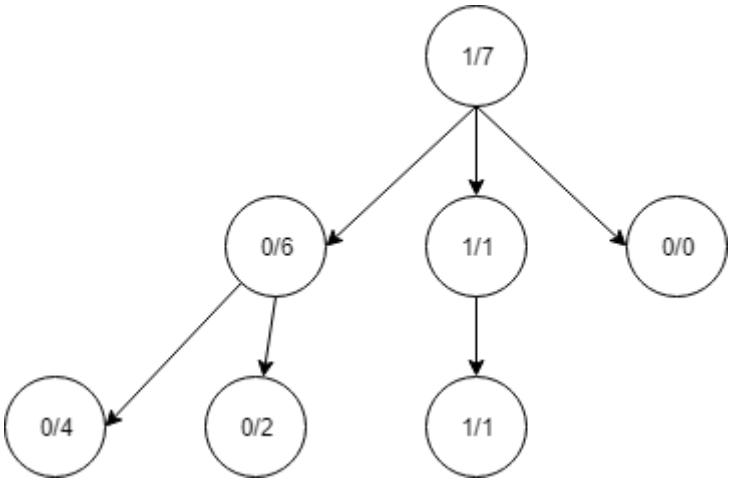

Abbildung 6: Beispiel eines Entscheidungsbaums

Im Beispiel von Abbildung 6 kann beim nächsten Aufruf des Baums der letzte, noch unbenutzte Knoten in der Selektionsphase verwendet werden. Nach diesem Zug gibt es keine weiteren Knoten mehr. Ist das Spiel zu diesem Zeitpunkt noch nicht vorbei, wird nun aus der möglichen Auswahl an Zügen, ein neuer Zug als Knoten dem Baum hinzugefügt. Dies ist in Abbildung 7 durch den geschwungenen Pfeil sichtbar.

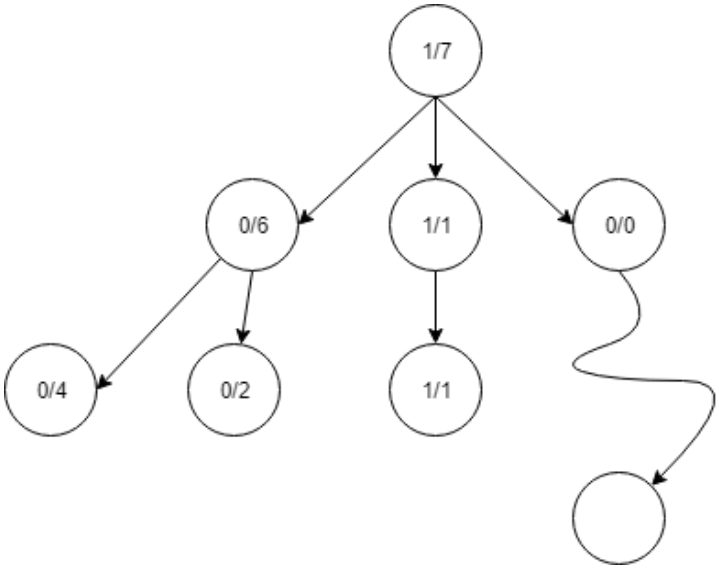

Abbildung 7: Beispiel eines Entscheidungsbaums (Expansionsphase)

Die nächste Phase des Algorithmus ist die Simulation. Hierbei wird das Spiel durch zufällige Selektion weiterer Züge zu Ende gespielt. Das Ergebnis davon wird in der letzten Phase des Algorithmus durch den Baum vom Blatt zurück zur Wurzel iteriert und nennt sich Backpropagation. Diese ist in Abbildung 8 dargestellt.

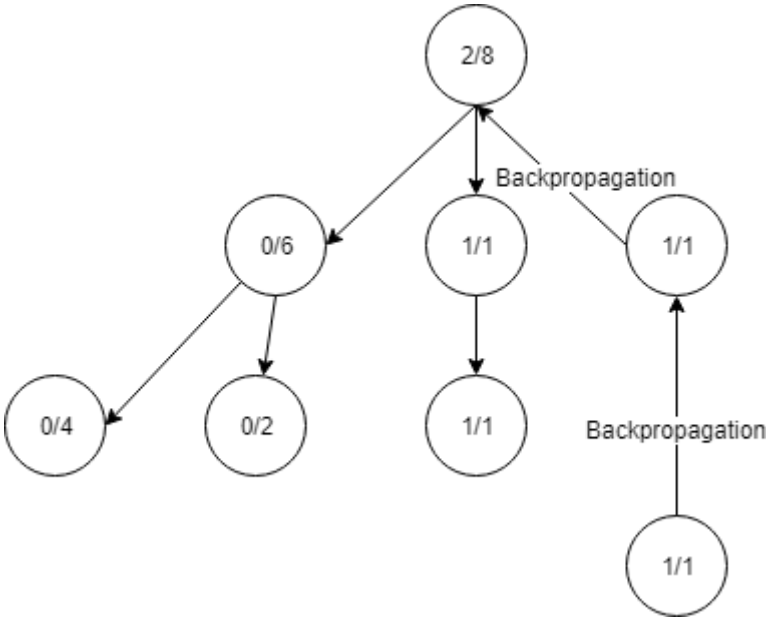

Abbildung 8: Beispiel eines Entscheidungsbaums (Backpropagation)

In diesem Beispiel wurde das Spiel gewonnen. Deswegen wird die Anzahl der Siege, sowie die Gesamtbesuche des Knotens erhöht. Auch die Anzahl der Gewinne und Besuche der Elternknoten werden aktualisiert (Chaslot, Bakkes, Szita, & Spronck, 2008) (Browne, Powley, Whitehouse, Lucas, & al., 2012).

4.3 Python

Bei der Programmiersprache Python handelt es sich um eine von Guido van Rossum entwickelte Sprache. Python ist eine plattformunabhängige Programmiersprache, wodurch diese auch unter verschiedenen Betriebssystemen funktioniert. Die aktuellste Version der Python Programmiersprache ist zurzeit die Version 3.7.4. Da Python unter einem offenen Standard entwickelt wird, bietet es sich neben kleineren Projekten, auch für Unternehmen an, diese Programmiersprache zu verwenden. Das Ziel, welches Guido van Rossum mit seiner Entwicklung erreichen wollte, war die Schaffung einer einfach zu verstehenden und vor allem einfach zu lesenden Sprache. Wo andere Programmiersprachen Klammern verwenden, löst Python dies über Einrückungen des Quellcodes. Dadurch muss der Benutzer nicht ständig Klammern setzen, sondern kann durch Einrücken des Quellcodes die Programmbereiche zuweisen.

Ein weiterer Vorteil ist, dass am Ende einer Codezeile kein Semikolon verwendet werden muss, um den Befehl abzuschließen. Ebenso lassen sich die dynamischen

Datentypen in Python als Vorteil hervorheben. Wo in anderen Programmiersprachen, wie C++ oder Java, die Variablen alle vor Verwendung ein Datentyp brauchen, ermittelt Python diesen dynamisch. Im Folgenden ist ein Beispiel, um zu veranschaulichen wie Quellcode in Python aussieht, verglichen mit Quellcode aus der Programmiersprache C++.

```
int addieren (int x, int y) {
    if(X > y) {
        return x + y;
    }
    return 0;
}

def addieren(x,y):
    if x > y:
        return x + y

    return 0
```

Abbildung 9: C++ Quellcode vs. Python Quellcode

In Abbildung 9 befindet sich oben der C++ Code und zum Vergleich unten nochmal die gleiche Methode in Python geschrieben.

4.4 TensorFlow und Keras

TensorFlow stellt sich selbst als die Basisbibliothek zur Erstellung von Machine Learning Algorithmen vor (tensorflow, kein Datum). Diese wurde von Google entwickelt und 2015 frei zur Verfügung gestellt (Diedrich, 2015). TensorFlow stellt seine Funktionen in Python, JavaScript, C++ und Java bereit. Die aktuellste stabile Version der Kernfunktionen ist Version r1.14. s

Eine weitere API zum Trainieren von neuronalen Netzen ist Keras. Ähnlich wie TensorFlow hat auch Keras eine Implementierung in Python, aber auch in Theano. Ziel von Keras ist es, dass der Benutzer schnell in der Lage ist, seine Planung zu Code umzusetzen. Angestoßen wurde Keras durch François Chollet (keras, kein Datum) (Chollet, 2019). Mittlerweile wurde eine Keras Implementierung in TensorFlow integriert (tensorflow, kein Datum).

4.5 Git

Eine der wichtigsten Funktionen, die ein Programmierer bei seiner Arbeit benötigt, ist die Versionsverwaltung. Die Versionsverwaltung ist eine Möglichkeit ältere Stände, beispielsweise der Software, wiederherzustellen. Hierfür speichert die entsprechende Verwaltungssoftware die jeweiligen Stände in einer Datenbank ab.

Eine Möglichkeit der Versionsverwaltung ist, diese lokal zu verwenden. Bei dieser Variante speichert die Software die Versionen lokal auf der Festplatte ab und der Benutzer kann bei Bedarf zwischen den verschiedenen Versionen wechseln. Nachteil dieser Variante ist jedoch, dass die Daten verloren gehen, wenn die Festplatte defekt ist. Ebenfalls wird so das gemeinsame Arbeiten an einer Software deutlich erschwert, da die Softwarestände immer wieder zwischen allen Projektteilnehmern ausgetauscht werden müssen.

Eine bessere Variante, ist die zentrale Lösung. Die gesamte Verwaltung der Softwarestände wird hierbei von einem Server übernommen, bei dem Benutzer sich vorher anmelden müssen. So kann bereits von Anfang an festgelegt werden, wer auf die Daten zugreifen kann. Ein weiterer positiver Punkt ist der Informationsaustausch. Der Projektleiter kann so einfach auf dem Server schauen, wie weit das Projekt bisher vorangekommen ist. Aber genau wie bei der lokalen Varianten, sind bei der zentralen Verwaltung, Probleme vorhanden. Das wohl größte Problem ist die Verfügbarkeit der Daten. In diesem Zusammenhang ist auch der Begriff “Single Point of Failure” zu erwähnen. Sollte der Server Probleme haben oder die Verbindungsstrecke zum Server ausfallen, erhalten die Benutzer auch keinen Zugriff mehr auf ihre Daten.

Um die Vorteile beider Systeme zu vereinen, wird immer mehr auf die Kombination beider Varianten gesetzt. Es wird ein zentraler Server verwendet, um die Projektversionen zu verwalten und die Daten werden bei Bearbeitung für den jeweiligen Benutzer lokal heruntergeladen und gespeichert. Der Benutzer kann dann an seinen Projektbereichen arbeiten und anschließend seine Version der Software wieder auf dem Server ablegen (Chacon & Straub, 1.1 Los geht’s - Was ist Versionsverwaltung?, 2014).

Eines dieser kombinierten Systeme ist Git. Neben der hohen Geschwindigkeit zur Datenverarbeitung und Kontrolle hat sich Git auch noch das Ziel gesetzt, durch ein einfaches Design dem Benutzer eine möglichst einfache Handhabung, mit der Software, zu bieten. Ebenfalls wurde bei der Entwicklung von Git darauf bestanden, nicht lineare Entwicklung zu unterstützen. Damit ist gemeint, dass die Anwender nicht zwangsläufig alle auf dem gleichen Softwarestand arbeiten müssen, sondern dass jeder Benutzer an seinen eigenen Features arbeiten kann. Die Softwarestände müssen dann anschließend vom Verwaltungstool wieder entsprechend zusammengeführt werden. Ein weiterer wichtiger Punkt, der beachtet wurde, beschäftigt sich mit der Projektgröße. Da Git als Versionsverwaltungssoftware nicht nur für kleine Anwendungen funktionieren soll, haben die Entwickler entsprechende Funktionen eingebaut, um auch mit großen Datenmengen und Dateien umzugehen (Chacon & Straub, 1.2 Los geht's - Kurzer Überblick über die Historie von Git, 2014).

Der wohl größte Unterschied zwischen Git und anderen Verwaltungstools wie CVS, Subversion, Perforce oder Bazaar, ist die Betrachtung der Daten. Git verwaltet die Daten anders als die genannten Programme mit Referenzen. Sollten sich in einem neuen Softwarestand nur einige der Dateien verändern, speichert Git für nicht veränderte Dateien, eine Referenz auf ihren vorherigen Stand. Durch dieses Vorgehen speichert Git bei jedem Update der Daten nicht alle Dateien neu ab, egal ob diese verändert wurden oder nicht, was zu einer besseren Performance führt (Chacon & Straub, 1.3 Los geht's - Git Grundlagen, 2014).

Um Git zu verwenden, muss zuerst ein Projekt aufgesetzt und die erforderlichen Anwender diesem hinzugefügt werden. Anschließend können die Anwender den Softwarestand vom Server laden und anfangen diesen lokal zu bearbeiten. Um die Übersichtlichkeit zu wahren, können die Anwender eigene Zweige (Branches), der heruntergeladenen Version erstellen. Nachdem das neue Feature oder das Update des Quellcodes fertig ist, kann der Anwender seinen lokalen Stand zurück an den Server schicken. Dieser prüft, ob der Anwender einen neuen Branch zum Projekt hinzufügen möchte oder ob seine Veränderungen auf einem bereits vorhandenen Branch gespeichert werden sollen. Ebenfalls ist es möglich mit Git das verschiedene Braches wieder zusammenzuführen. So können die Anwender jeweils an eigenen

Bereichen arbeiten und müssen nicht befürchten, dass ein anderer Benutzer ihren Stand überschreibt.

4.6 Spiele

4.6.1 Tic-Tac-Toe

Bei dem Spiel Tic-Tac-Toe handelt es sich um ein Zweipersonen-Strategiespiel. Ziel dieses Spiels ist es, drei seiner „Spielfiguren“ in einer Reihe anzuordnen. Zu Beginn des Spiels wählen die beiden Spieler zwischen den zur Verfügung stehenden Zeichen „X“ und „O“. Der Spieler, der die Figuren mit dem X-Zeichen bekommt, beginnt das Spiel.

Das Spielfeld eines Tic-Tac-Toe Spiels besteht aus einem 3x3 Feld und bietet somit den Spielern, zu Beginn des Spiels, die Möglichkeit eines der neuen Felder auszuwählen. Abgebildet ist ein solches Feld in Abbildung 10.

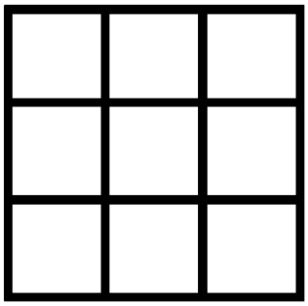

Abbildung 10: Tic-Tac-Toe Spielfeld

„Spieler 1“ beginnt nun das Spiel, indem er eines der Felder mit seiner Spielfigur belegt. Dieses Feld ist somit nicht mehr in der Anzahl der verfügbaren Felder und kann auch nicht vom Gegenspieler markiert oder überschrieben werden. Nachdem „Spieler 1“ seinen Zug gemacht hat, stehen „Spieler 2“ nur noch 8 Felder zur Auswahl. Das Spiel wird nun abwechselnd fortgeführt, bis einer der beiden Spieler 3 Spielfiguren in einer Reihe hat oder keine freien Felder mehr zur Verfügung stehen.

Bei der Gewinnerkennung eines Tic-Tac-Toe Spieles, werden sowohl die horizontale Ebene, als auch die vertikale und diagonale berücksichtigt. Deutlich gemacht wurde dies in Abbildung 11.

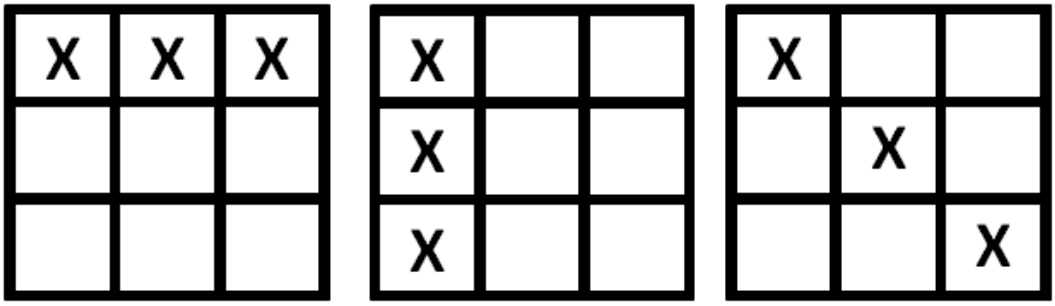

Abbildung 11: Tic Tac Toe Ebenen

Betrachtet man die Komplexität eines Tic-Tac-Toe Spiels wird schnell klar, dass es viele verschiedene Varianten gibt, um einen Sieg zu erzielen.

4.6.2 Vier Gewinnt

Bei dem Spiel Vier Gewinnt oder auch Connect Four genannt, geht es wie bei dem Spiel Tic-Tac-Toe darum, seine Spielsteine in einer vertikalen, horizontalen oder diagonalen Linie anzuordnen. Das Spielfeld besteht aus einem 6 Felder hohen und 7 Felder breiten, hochkant stehendem Feld. Dieses wird abwechselnd von beiden Spielern oben mit ihren jeweiligen Spielsteinen gefüllt. Zu Beginn erhalten beide Spieler jeweils 21 Spielsteine ihrer Farbe, wobei in der Ursprungsversion die beiden Farben Gelb und Rot verwendet werden. Die beiden Spieler lassen dann jeweils abwechselnd einen ihrer Spielsteine in eine der sieben Spalten fallen, wo dieser Spielstein dann, durch die Schwerkraft, bis auf den untersten freien Platz der Spalte fällt. Sollte eine Spalte gefüllt sein, also alle 6 Felder dieser Spalte mit einem Spielstein belegt sein, so gilt diese Spalte als voll und es darf kein weiterer Spielstein in diese Spalte gelegt werden. Dargestellt werden das Spielfeld und die Spielmechanik beispielhaft in folgender Abbildung.

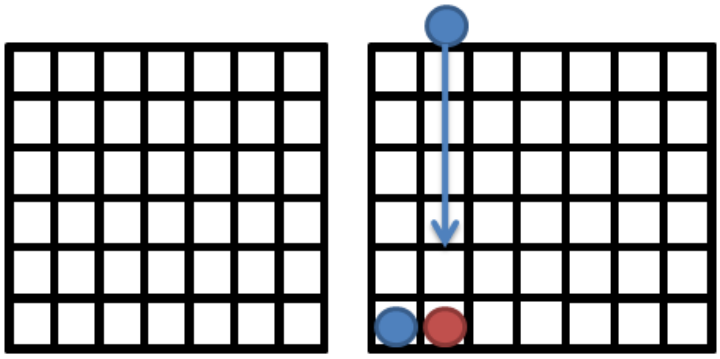

Abbildung 12: Vier Gewinnt

Sollten die beiden Spieler alle ihrer Spielsteine benutzt haben, das Spielfeld ist also komplett gefüllt und es befinden sich keine vier Steine in einer Reihe, so geht das Spiel unentschieden aus.

Sollte ein Spieler jedoch gewinnen, also vier oder mehr seiner Steine in einer horizontalen, vertikalen oder diagonalen Ebene verbunden haben, werden auch die Steine des Gegners betrachtet. Sollten die Steine des Gegenspielers alle verbunden sein, so erhält dieser auch einen Punkt, wodurch das Spiel wieder zu einem Unentschieden führt.

Ähnlich wie beim Tic-Tac-Toe ist auch hier die beste Position in der Mitte, da von hieraus die meisten Möglichkeiten bestehen, eine erfolgreiche Spielstrategie umzusetzen. Betrachtet man die möglichen Zustände auf einem 7x6 Felder großem Spielfeld, so erhält man die Zahl 4.531.985.219.092. Von diesen über 4 Billionen möglichen Zügen gelten 1.904.587.136.600 als Gewinnzustände und 713.298.878 als Unentschieden (Haran, 2013).

4.6.3 **Fünf in einer Reihe**

Eines der bekanntesten Strategiespiele ist das „Fünf in einer Reihe“. Überall in der Welt kennt man das Spiel unter einem anderen Namen. Beispielsweise wird es in China Wuziqi genannt, Omok in Korea oder Gomoku narabe in Japan. In Deutschland wird das Spiel oft mit einem Stift auf Karopapier gespielt und es wird meist vorher ein Spielbereich festgelegt. In China wird das Spiel z.B. mit Go Steinen auf einem 15x15, 17x17 oder 19x19 Feld gespielt.

Wie auch schon bei den Spielen Tic-Tac-Toe und Vier Gewinnt handelt es sich bei Fünf in einer Reihe um ein Spiel, welches von 2 Spielern gespielt wird. Zu Beginn wählen beide Spieler eine Markierungsmöglichkeit, wobei in den meisten Fällen die Zeichen „X“ und „O“ verwendet werden. Diese sind aber nicht fest vorgegeben und können frei gewählt werden. Nachdem das Spielfeld festgelegt wurde und die beiden Spieler sich jeweils für eine Markierungsmöglichkeit ihrer Felder entscheiden haben, wird entschieden wer das Spiel beginnen darf.

Ziel ist es, als erster, fünf gleiche Einheiten desselben Spielers in einer horizontalen, vertikalen oder diagonalen Ebene anzuordnen. Hierbei ist darauf zu achten, dass nicht wie beim Vier Gewinnt die Spielsteine durch die Schwerkraft nach unten fallen, sondern dass hier die Auswahl des Feldes frei erfolgt. Sollte das ganze Spielfeld markiert sein und es wurde keine Reihe von fünf gleichen Steinen erzielt, endet das Spiel unentschieden.

Bei den beiden Spielen Vier Gewinnt und Fünf in einer Reihe lassen sich bei Gewinnerkennungsalgorithmus bereits einige Vereinfachungen vornehmen. Da die Reihe beim „Fünf in einer Reihe“ mindestens fünf Steine lang sein muss, braucht der Algorithmus bei der diagonalen Prüfung nicht die Ecken des Feldes prüfen, da diese in der diagonalen Ebene keine Reihe mit fünf Einheiten zulassen. In Abbildung 13 wird von einem 15x15 Feld ausgegangen, um den Vorgang besser veranschaulichen zu können.

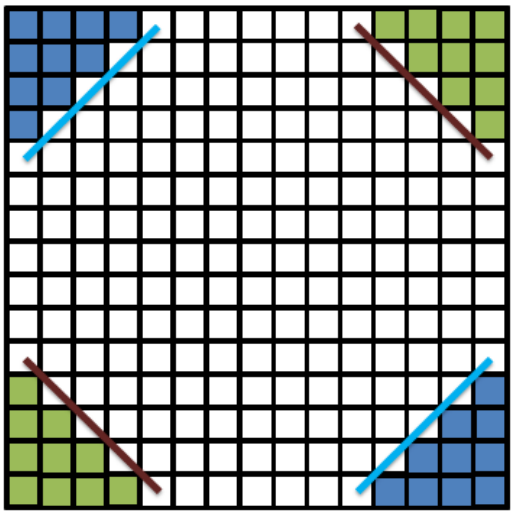

Abbildung 13: Fünf in einer Reihe, diagonale Ebene

Für die Überprüfung der diagonalen Ebene verwendet der Algorithmus zwei Durchläufe. In dem ersten Durchlauf prüft der Algorithmus von links unten nach rechts oben, also die positive Diagonale. Die in der Abbildung eingezeichneten blauen Ecken werden hierbei übersprungen, da diese nicht in der Lage sind eine vollständige Reihe aus fünf gleichen Elementen zu besitzen. Die erste und letzte Möglichkeit einer vollständigen Reihe in positiver diagonaler Richtung sind durch die türkisenen Striche dargestellt.

Im zweiten Durchlauf prüft der Algorithmus nun die negative Diagonale, wobei er von links oben nach rechts unten eine Reihe sucht. Die hier in grün dargestellten Ecken werden bei diesem Vorgang übersprungen, da diese wie bei der Prüfung der positiven Diagonalen, keine vollständige Reihe zulassen. In der Abbildung sind auch für die Überprüfung der negativen Diagonalen die erste, sowie letzte Möglichkeit einer vollständigen Diagonale durch die brauen Striche veranschaulicht.

5 Projektdurchführung

5.1 Implementierung der Spiele / Frontend

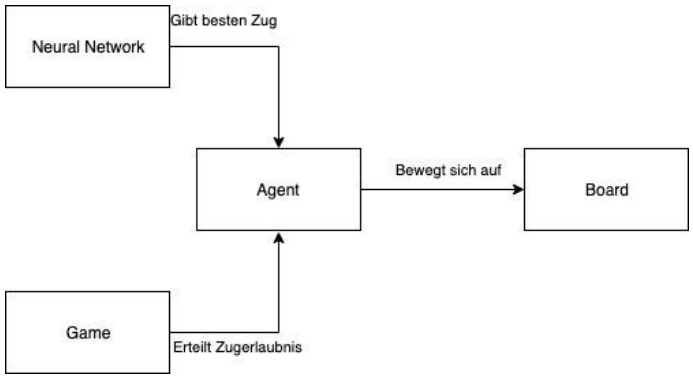

Abbildung 14: Grobarchitektur der Software

Die Grobarchitektur der Software kann Abbildung 14 entnommen werden.

Das `Game` fordert den `Agenten` auf, einen Zug auf dem Board durchzuführen. Der angesprochene `Agent` lässt sich daraufhin vom Neuronalen Netz bzw. dem Monte-Carlo-Baum alle möglichen Züge aus seiner aktuellen Situation heraus mit den dazugehörigen Gewinnwahrscheinlichkeiten pro Zug geben. Nach Entfernen der unerlaubten Züge wählt er den Besten aus. Diesen führt er daraufhin auf dem `Board` aus. Danach ist der andere Spieler (in Form eines weiteren `Agenten`) an der Reihe.

Das `Game` speichert Informationen über Gewinn und Verlust eines Spielers, um eine statistische Auswertung möglich zu machen. Die detaillierte Architektur kann Abbildung 15 entnommen werden.

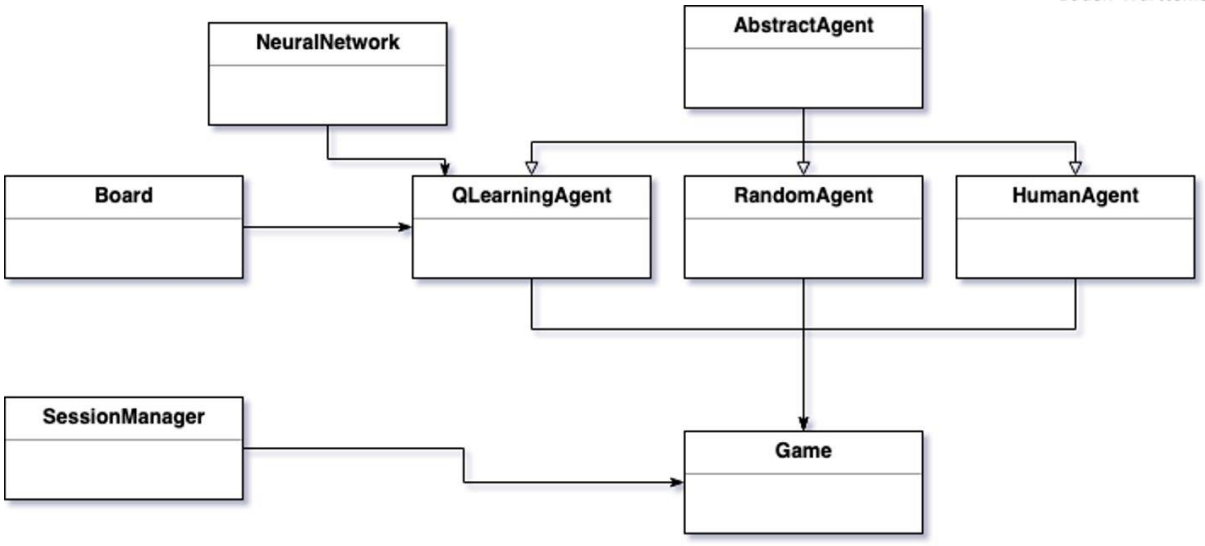

Abbildung 15: Softwarearchitektur im Detail

Bei der Entwicklung der Architektur wurde darauf geachtet, eine möglichst hohe Erweiterbarkeit zu erhalten. Daher wurde eine abstrakte Klasse `AbstractAgent` definiert, welche du zwingend notwendigen Methoden aller Agenten beinhaltet.

Jeder Agent, erbt von dieser Mutterklasse und erweitert sie um die von ihm spezifischen Methoden (beispielsweise die Integration eines neuronalen Netzes).

Zu den wichtigsten Agenten gehört der `QLearningAgent` welcher mit Hilfe des Q-Learning Verfahrens seine Zugwahrscheinlichkeiten ermittelt, oder der `HumanAgent`, welcher dem Nutzer die Möglichkeit gibt, selbst gegen das Netz zu spielen.

Auch das Spielfeld selbst, innerhalb der `Board` Klasse lässt sich dynamisch auf verschiedene Größen einstellen.

Sollte sich das Regelwerk ändern, ließe sich auch dieses in der `Board` Klasse durch Austausch der Gewinnprüfmethode umsetzen.

Der `SessionManager` dient dem Managen der einzelnen neuronalen Netze innerhalb des TensorFlow Graphen.

5.2 Umsetzung Neuronales Netz

Im Rahmen dieser Studienarbeit wurden eine Reihe von neuronalen Netzen auf ihre Eignung geprüft.

Grundsätzlich lässt sich mit jedem neuronalen Netz fast jedes Problem lösen. Der Unterschied kommt erst bei der Effektivität zum Vorschein.

Auf Grund der Einfachheit wurde zunächst auf ein simples Feed-Forward Neural Network gesetzt. (vgl. Kapitel 4.1.1)

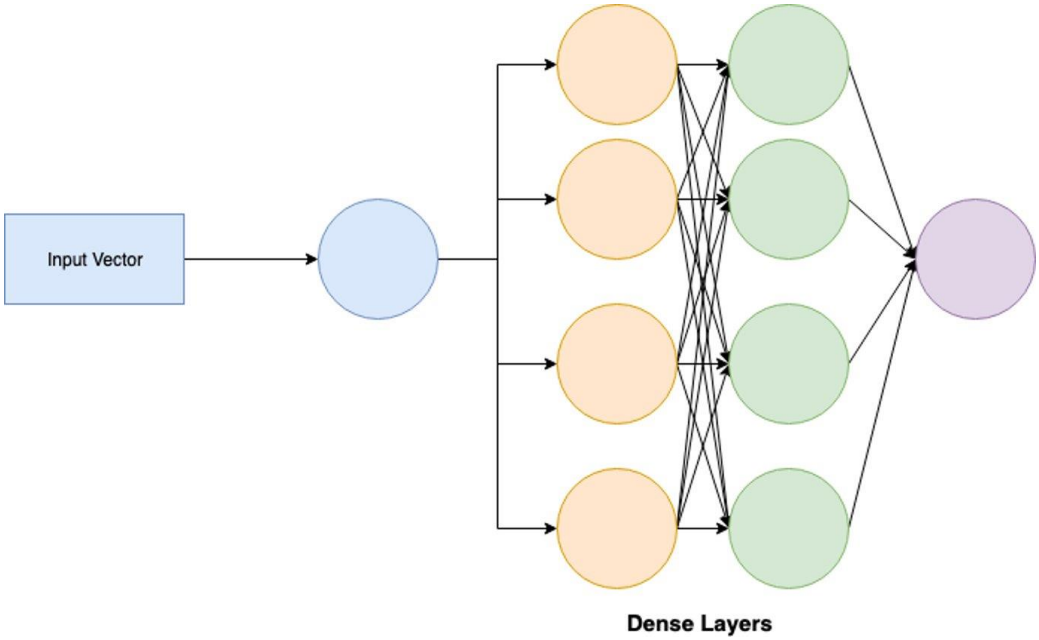

Abbildung 16: Implementiertes neuronales Netz

Es besitzt einen Eingabevektor welcher in einem Hidden Layer verarbeitet wird. Die Ausgabeschicht besitzt n^2 Neuronen wobei n die Dimension des Spielfelds darstellt. Ein 50x50 Spielfeld besitzt damit also 2.500 Ausgabeneuronen.

Als Eingabevektor wurde ein eindimensionales Array gewählt, welches die Positionen der beiden Spieler und der noch leeren Felder kodiert.

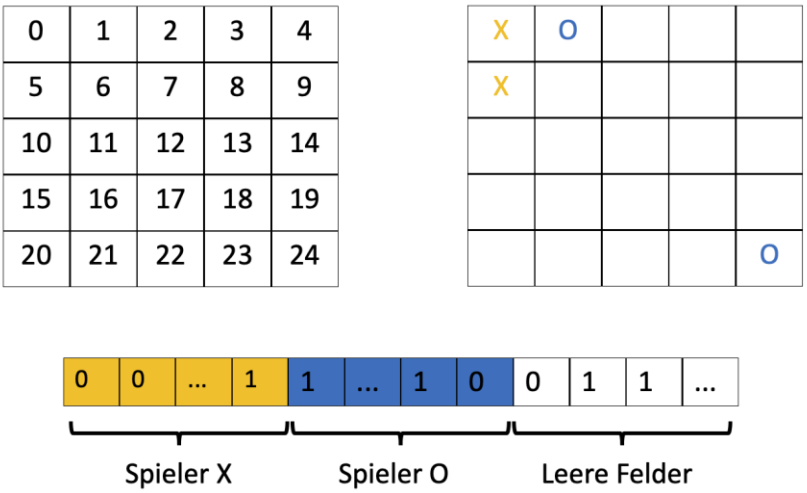

Abbildung 17: Kodierung des Spielzustands

Abbildung 17 zeigt die Umsetzung dieser Kodierung. Das Array besitzt die Länge von $n^2 * 3$ wobei n die Dimension des Spielfeldes ist. Bei einer Feldgröße von $5 * 5$

Ist das Array demnach 75 Felder lang. In den ersten n^2 Feldern werden die Positionen von „Spieler 1“ gespeichert. Die Position im Array entspricht dabei der Position auf dem Spielfeld, wobei das LSB der Position Null entspricht. Analog wird der Vorgang bei „Spieler 2“ und den leeren Feldern durchgeführt.

Dieses neuronale Netz erfüllt zwar seinen Zweck, ist allerdings sehr simpel und daher auch vergleichsweise schwach. Eine bessere Alternative wird unter Kapitel 7.3 erläutert.

5.3 Umsetzung Monte Carlo Tree Search

In dieser Arbeit wurde der MCTS Algorithmus verwendet. Implementiert wurde dieser in Zwei Klassen. Einer `Tree` sowie einer `Node` Klasse. Der `Tree` besitzt `Nodes`, was in Abbildung 18 zu sehen ist. Ein `Node` kann weitere `Nodes` besitzen. Dadurch entsteht eine hat-Beziehung. Dies ist dadurch bedingt, dass ein Knoten seinen Elternknoten und seine Kindknoten, wenn er welche besitzt, kennen muss. Die Kindknoten werden in einer Liste gespeichert.

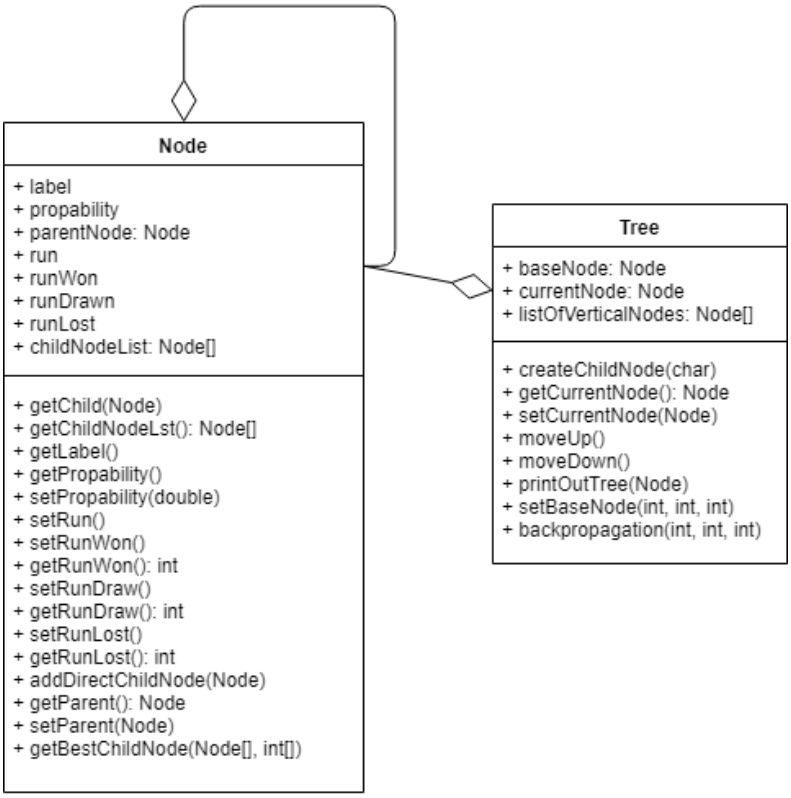

Abbildung 18: Klassendiagramm MCTS

Der `Tree` wird als Speicherobjekt der Knoten betrachtet und besitzt die Funktionen, um sich durch den gesamten Baum bewegen zu können. Bei der Initialisierung besitzt der `Tree` einen `baseNode`, welche die Wurzel darstellt. In dieser Wurzel wird gespeichert, wie oft der `Tree` trainiert wurde und wie oft dabei gewonnen, verloren oder unentschieden gespielt wurde.

Dies ist ein Hinweis auf die Güte des Baumes. Je besser die Gewinnratio zu der Verlustratio oder dem Unentschieden ist, desto besser ist der MCTS Algorithmus.

Der `Tree` ist zunächst als einfach verkettete Liste von den Blattknoten bis zum Wurzelknoten implementiert. Dies ist bedingt durch das `parentNode` Attribut in der `Node` Klasse. Die Wurzel ist der einzige Knoten, welche kein solches Attribut besitzt. Die einfach verkettete Liste wurde ausgewählt, da ein Pfad im Baum, von Wurzel bis Blatt, ein Spiel bedeutet. Beim Trainieren ist es durch diese Art von verketteter Liste einfach, die Backpropagation, wie in Kapitel 10 beschrieben, durchzuführen, da nur auf den Elternknoten zugegriffen und dessen Werte aktualisiert werden müssen. Da

der Wurzelknoten keine Elternknoten hat, muss dieser extra aktualisiert werden, dies ist durch die `setBaseNode` Funktion gegeben.

Durch eine Liste mit Kindknoten kann eine doppelte Verkettung jedoch angedeutet werden, wodurch eine vertikale Bewegung Richtung Blätter möglich ist.

Die Nodes sind die Speicherobjekte, welche die tatsächlichen Informationen speichern.

Das Label eines Nodes gibt in dieser Arbeit das Feld, bzw. den Zug an, der im Spiel durchgeführt werden muss. Weiterhin kennt ein Knoten seine eigene Gewinnchance, die `propability`. Dieses Attribut wird aus der Anzahl der Spiele insgesamt und aus den Gewinnen berechnet und gibt die Güte dieses Spielzuges, für diesen Moment an.

Da jeder Knoten Kindknoten besitzen kann, müssen diese gespeichert werden. Umgesetzt ist das in diesem Fall über eine Liste von Kindknoten. Beim Hinzufügen eines neuen Kindknotens kann demnach einfach die Liste überprüft und durchlaufen werden. Deshalb macht eine klassisch doppelt verkettete Liste in diesem Fall keinen Sinn.

Beim Verwenden dieser Datenstruktur wird im Neuronalen Netz zwischen der Trainingsphase und der Spielphase unterschieden. Beim Training dieses MCTS wurde sich gegen das komplette Selbstlernen entschieden. Die Auswahl, welche Züge beim Training gemacht und durchlaufen werden, werden von einem Neuronalen Netz zuvor bewertet. Diese Informationen erhält das neuronale Netz und wählt damit den besten Zug aus und fügt diesen, falls er nicht bereits existiert, dem Baum zu. Dies wird so lange durchgeführt, bis das Spiel zu Ende ist. Damit wird pro Spiel ein vertikaler Ast aufgebaut. Somit wird die, aus dem klassischen MCTS Algorithmus bekannte, Simulationsphase übergangen.

Diese Entscheidung hat Performancegründe. Bei großen Feldern kann nicht jeder Zug durchgeführt und mehrfach durchlaufen werden. Der Baum würde viel zu groß werden. Daher muss die Vorauswahl der besten Züge durch das Neuronale Netz gefunden und verwendet werden. Am Ende jedes Spieldurchlaufs wird die `backpropagation` Methode aufgerufen. Diese Methode ist dafür zuständig, die Position im Baum wieder

auf den Wurzelknoten zu setzen und alle Werte entlang des vertikalen Pfades zu setzen, bzw. zu aktualisieren.

Beim Spielen wird bei der Auswahl des Zuges der Kindknoten verwendet, welcher die höchste `propability` hat. Dem Baum wird eine Liste mit möglichen Zügen übergeben. Diese wird mit den Kindknoten des momentanen Knoten verglichen. Bei Auffinden von Gleichheiten wird aus den in der Liste existierenden Zügen der Zug ausgewählt, der im Baum als Knoten mit der besten `propability` gespeichert ist.

5.4 Spiel aufsetzen

Die fertigen Methoden zur Spieldurchführung befinden sich in der `Game.py` um zwei beliebige `Agents` gegeneinander spielen zu lassen, müssen diese zunächst erstellt werden.

```
#Erstellen eines Agents mit QLearning
nnplayer = SimpleQLearningAgent("QLearner1")
#Erstellen eines Agents, der zufällige Züge durchführt
rndplayer = RandomAgent()
#Erstellen eines Agents, der Nutzereingaben verarbeitet
human = HumanAgent()
#Erstellen eines Agents, der einen Monte-Carlo-Baum verwendet
MCTSAgent = MonteCarloAgent("MCTSA")
```

Danach kann der Lernmodus durch Aufruf der Methode `test_players(agent_1, agent_2)` gestartet werden. Dabei stehen `agent_1` und `agent_2` für die zuvor erstellten Agenten.

Die Methode liefert als Rückgabe die Spielnummer, die Anzahl der Gewinne des jeweiligen Spielers und die Anzahl der unentschiedenen Spiele.

```
game_number, p1_wins, p2_wins, draws = test_players(nnplayer, MCTSAgent)
```

Optional kann der Methode noch eine spezifische Anzahl an zu spielenden Epochen oder eine spezifische Anzahl an Spielen innerhalb einer Epoche festgelegt werden.

```
def test_players(p1 : AbstractAgent, p2 : AbstractAgent, games_per_epoch = 100,
                 num_epochs = 100):
```

Wie im Code zu sehen sind die Standardwerte jeweils auf 100 gesetzt.

Es kann zwischen zwei Speicherarten gewählt werden. Sollte ein MCTS Agent eingesetzt werden, so ist nach jeder Epoche ein der folgende Code aufzurufen, um den Monte Carlo Tree zu speichern:

```
pickle.dump(MCTSAgent.MCTS,open("save.p", "wb"))
```

Beim Q-Learning Agenten sollte das Netz hingegen nur dann gespeichert werden, wenn es besser ist, als das vorhergehende. Erreicht wird dies durch ein:

```
if(x_count>most_wins and train):
        saver = tf.train.Saver()
        saver.save(SessionManager.get_session(),"models/best.ckpt")
```

5.5 Herausforderungen

Da dieses Projekt auf einem anderen aufbauen sollte, wurde zunächst der Fokus daraufgelegt, die Arbeit der Vorgänger nachzuvollziehen und anschließend darauf aufzubauen. Dabei ist aufgefallen, dass, obwohl die mathematischen Grundlagen verständlich und anschaulich erklärt wurden, die Dokumentation der Umsetzung lückenhaft gewesen ist. Denn es waren kaum Angaben zur Umsetzung gegeben. Bei Betrachtung des Codes fiel auf, dass nicht genügend Kommentare geschrieben wurden, um ein zusammenhängendes Gesamtbild des Umfangs des Programms zu aufzubauen.

Ein ähnliches Problem bildete die Einarbeitung in TensorFlow. Das Projekt auf dessen Basis diese Arbeit entstanden ist, hatte sich mit reinem TensorFlow Code beschäftigt. Um auf den Vorgängern aufzubauen, wurde sich deshalb zunächst darauf fokussiert und eine andere API nicht betrachtet. Zu spät wurde deshalb bemerkt, dass Keras für ein solches Projekt besser geeignet gewesen wäre, da die Oberfläche leichter zu verwenden ist und Models sowie Datenobjekte leichter abgespeichert werden können.

Ein weiteres Problem bei TensorFlow ist der praktische Einstieg. Im Internet existieren viele Quellen und Anleitungen zur Benutzung der Bibliothek oder zu neuronalen Netzen. Diese Anleitungen können in zwei Kategorien aufgeteilt werden. Zum einen sind dies Anleitung, welche die mathematischen Grundlagen der neuronalen Netze beschreiben, sowie den theoretischen Ablauf der Funktionen. Zum Anderen existieren die praktischen Umsetzungen, welche jedoch meist keine genaue Beschreibung der

Architektur des umgesetzten Programms besitzen und bei der Implementierung nicht auf die verwendeten Funktionen der Bibliothek eingehen. Dadurch wird nicht erklärt, was diese bewirken oder warum diese verwendet werden. Dadurch fällt es sehr schwer, sich in das Thema einzuarbeiten. Der Übergang von Theorie und Praxis ist daher eine Herausforderung.

Eine letzte Herausforderung in diesem Projekt war den Zusammenhang zwischen dem MCTS Algorithmus und dem neuronalen Netz zu finden. Der MCTS Algorithmus reicht bei weniger umfangreichen Spielen aus, um passable bis gute Ergebnisse zu erhalten. Dies kann man bei einer einfachen Tic-Tac-Toe Implementierung sehen. Bei größeren Feldern, wie ein 15x15 Feld benötigt der Algorithmus deutlich mehr Kapazität und Speicher. Die Kombination der beiden Techniken umzusetzen hat dadurch viel Zeit gekostet. Es war auch nicht von Anfang an klar, in welcher Form die beiden Technologien zusammenarbeiten können.

6 Diskussion der Ergebnisse

Im Rahmen des Projektes entstand ein funktionsfähiges Spieleframework, welches simple Brettspiele in Form von „verbinde X Steine miteinander“ auf beliebiger Spielfeldgröße spielen kann.

Das mit Hilfe von Tensorflow erstellte neuronale Netz ermittelt anhand eines gegebenen Inputvektors die Gewinnwahrscheinlichkeiten für jeden möglichen Zug, allerdings konnte auf Grund des zeitlichen Rahmens keine abschließende Verifikation der Präzision dieser Wahrscheinlichkeiten durchgeführt werden.

Statistisch gesehen besiegt das neuronale Netz einen Spieler der wahllos Züge wählt zu ca 92%, wie in Abbildung 19 zu sehen.

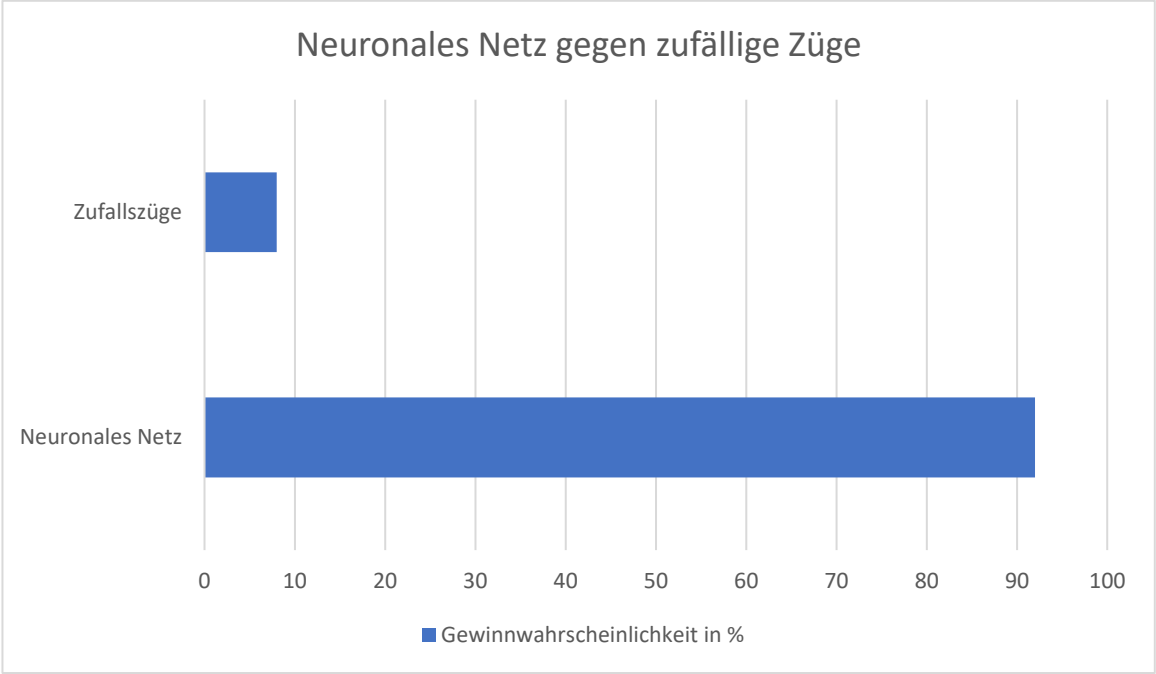

Abbildung 19: Diagramm neuronales Netz gegen zufällige Züge

Dieses Ergebnis belegt zumindest die grundsätzliche Korrektheit der empfohlenen Züge.

Ein trainiertes neuronales Netz gewinnt gegen ein beinahe untrainiertes Netz allerdings nur in ca. 65-70% der Fälle, sichtbar in Abbildung 20. Dies deutet darauf hin, dass die bisherige Trainingszeit noch nicht ausreichend ist.

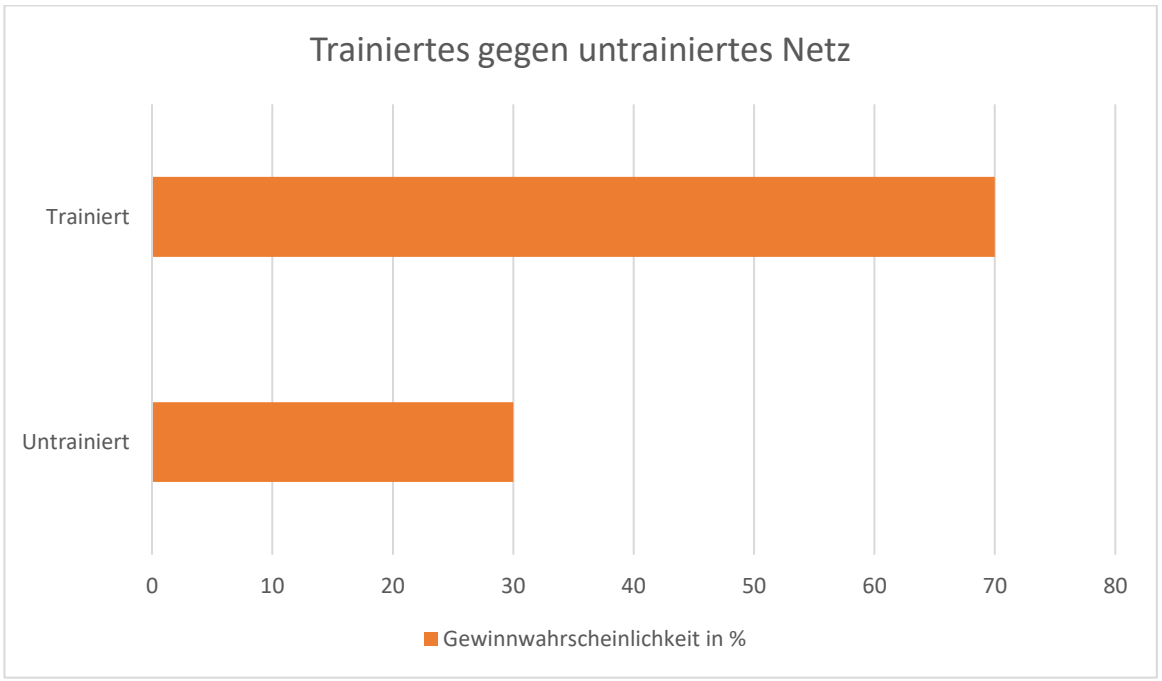

Abbildung 20: Diagramm trainiertes gegen untrainiertes Netz

Derzeit kann der menschliche Spieler die künstliche Intelligenz noch sehr einfach besiegen, da diese nicht versucht, gegnerische Fortschritte zu unterbrechen. Auch das kann auf eine nicht ausreichende Trainingszeit zurückgeführt werden.

7 Ausblick

7.1 Monte Carlo Tree Search

Der Monte Carlo Algorithmus wurde in diesem Projekt nicht vollständig umgesetzt. In einer zukünftigen Arbeit sollten die fehlenden Funktionalitäten hinzugefügt werden. Besonders wichtig eingeschätzt wird dabei das Hinzufügen der UCB Formel. Beim Spielen sucht der Baum sich im Moment den Knoten, welcher die höchste Gewinnwahrscheinlichkeit besitzt. Dieser wird ausgewählt und benutzt. Bei mehr Gewinnen wird seine Gewinnwahrscheinlichkeit immer höher, wodurch dieselbe Knotenfolge immer wieder durchgeführt wird. Nur durch Verluste oder unentschieden gespielten Spielen kann sich die Gewinnwahrscheinlichkeit senken, wodurch möglicherweise ein anderer Knoten verwendet wird. Durch den UCB könnte man eine Exploration in der Trainingsphase sowie der Spielphase einbauen, sodass nicht nur die Züge mit der besten Wahrscheinlichkeit verwendet werden, sondern auch Züge, die sonst nie in Betracht gezogen worden wären.

Weiterhin könnte dadurch das Problem gelöst werden, dass wenn kein Knoten für die Liste an möglichen Zügen existiert, das Programm nicht weiterlaufen kann.

Eine andere Möglichkeit eine Auswahl zu treffen wäre, eine exakte Verlustwahrscheinlichkeit zu berechnen und mit der Gewinnwahrscheinlichkeit abzugleichen. Dadurch können noch weitere Regeln zur Auswahl der Knoten hinzugefügt werden, welche die Chance haben, den Auswahlalgorithmus weiter zu verbessern. Beispielsweise könnte eingegrenzt werden, wie hoch die maximale Gewinnwahrscheinlichkeit sein darf oder wie viel besser die Gewinnwahrscheinlichkeit im Gegensatz zur Verlustwahrscheinlichkeit ist.

Letztlich wird der MCTS Algorithmus stetig durch mehr Training besser. Durch ein gutes neuronales Netz, welches sinnvolle Züge liefert, kann ein sinnvoller Monte Carlo Tree aufgebaut werden. Deshalb wird empfohlen die Kombination von MCTS und neuronalem Netz beizubehalten, das Netz auszubauen und dadurch einen besseren Tree zu generieren.

7.2 Spiellogik

Um das Trainieren und das Spielen zu beschleunigen, können Änderungen am Algorithmus vorgenommen werden, welcher prüft, ob eine der beiden Parteien gewonnen hat.

Der derzeitige Algorithmus durchsucht nach jedem Zug das ganze Feld. Dieser Vorgang ließe sich mit einer kompakteren Abfrage schneller erledigen. Es könnte zum Beispiel in einem neuen Algorithmus nur noch ab der gerade gesetzten Position kontrolliert werden. Hierzu könnte die neue Prüfmethode dann alle Felder um das neu gesetzte Feld prüfen und müsste entsprechend nur maximal die nächsten vier Felder in alle Richtungen kontrollieren. Betrachtet man die Möglichkeit bei Fünf in einer Reihe zu gewinnen, so ergibt sich die Untersuchung von 8 Richtungen ab der aktuellen Position. Eine schematische Darstellung dieses Algorithmus ist in Abbildung 21 zu sehen.

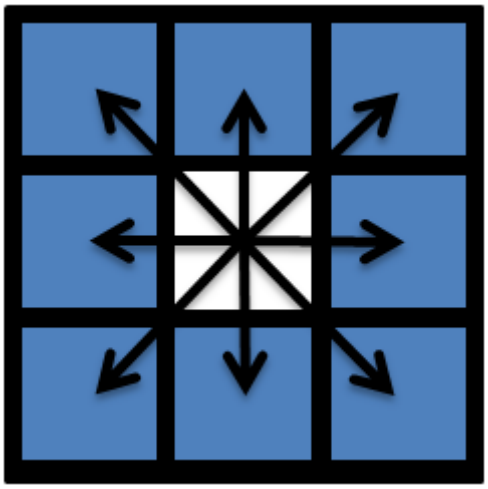

Abbildung 21: Neuer Algorithmus zur Bestimmung eines Gewinners

Sollte der Algorithmus feststellen, dass sich in einer Richtung kein Stein desselben Spielers befindet oder dass in einer Richtung eine Wand liegt, so können diese Richtungen bereits ausgeschlossen werden. Durch diese Vorgehensweise wird mit sehr hoher Wahrscheinlichkeit eine bessere Performance erzielt, als es bisher der Fall ist.

7.3 Andere Neuronale Netze

Im Projektverlauf wurde der Einsatz komplexerer Netzwerke analysiert. Auf Grund der zweidimensionalen Charakteristik eines Spielbretts eignet sich ein Convolutional Neural Network mit Pooling Schichten. Abbildung 22 zeigt einen Entwurf dieses Netzes.

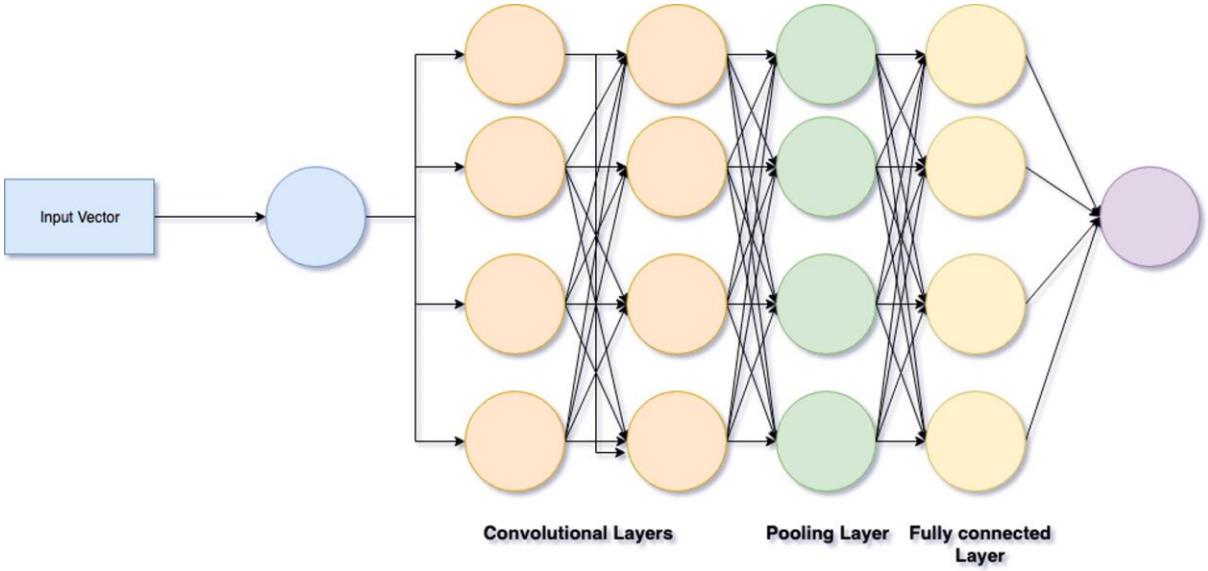

Abbildung 22: Vorschlag zukünftiges Netz

Die Ausgabeschicht, hier als „Fully connected Layer“ bezeichnet, besteht wie zuvor aus n^2 Neuronen. Hintergrund dieser Überlegung ist, dass sich Muster, die sich beim taktischen Spielen abzeichnen könnten, mit einem Convolutional Neural Network schneller erkannt werden. Daraus folgt eine schnellere Lernkurve bei gleichbleibendem Rechenaufwand.

7.4 Literaturverzeichnis

Aphex34. (16. Dezember 2015). *wikipedia*. Abgerufen am 12. Juli 2019 von https://de.wikipedia.org/wiki/Convolutional_Neural_Network#/media/Datei:Typical_cnn.png

Braun, H. (8. Dezember 2018). Abgerufen am 10. Juli 2019 von Wie Googles KI Schach veränderte: https://www.heise.de/newsticker/meldung/Wie-Googles-KI-Schach-veraenderte-4245938.html

Browne, C. B., Powley, E., Whitehouse, D., Lucas, S. M., & al., e. (März 2012). A Survey of Monte Carlo Tree Search Methods. *IEEE Transactions on Computational Intelligence and AI in Games, 4*(1).

Calomme, V. (kein Datum). Abgerufen am 10. Juli 2019 von Die Geschichte der Künstlichen Intelligenz: https://www.welove.ai/de/blog/post/geschichte-kuenstlicher-intelligenz.html

Chacon, S., & Straub, B. (2014). Abgerufen am 12. Juli 2019 von 1.1 Los geht's - Was ist Versionsverwaltung?: https://git-scm.com/book/de/v2/Los-geht%E2%80%99s-Was-ist-Versionsverwaltung%3F

Chacon, S., & Straub, B. (2014). Abgerufen am 11. Juli 2019 von 1.2 Los geht's - Kurzer Überblick über die Historie von Git: https://git-scm.com/book/de/v2/Los-geht%E2%80%99s-Kurzer-%C3%9Cberblick-%C3%BCber-die-Historie-von-Git

Chacon, S., & Straub, B. (2014). Abgerufen am 11. Juli 2019 von 1.3 Los geht's - Git Grundlagen: https://git-scm.com/book/de/v2/Los-geht%E2%80%99s-Git-Grundlagen

Chaslot, G., Bakkes, S., Szita, I., & Spronck, P. (2008). Monte-Carlo Tree Search: A New Framework for Game AI. *Proceedings of the fourth artificial intelligence and interactive digital entertainment conference* (S. 216-2017). Menlo Park, California: AAAI Press.

Chollet, F. (2019). *github*. Abgerufen am 11. Juli 2019 von https://github.com/fchollet

Colbaut, P., Czymmeck, S., & Retzbach, C. (2018). *Erprobung von neuronalen Netzwerken als KI für Computerspiele.* Friedrichshafen.

Colbaut, P., Czymmeck, S., & Retzbach, C. (2018). *Erprobung von neuronalen Netzwerken als KI für Computerspiele.* Friedrichshafen.

Coulom, R. (2006). Efficient Selectivity and Backup Operators in Monte-Carlo Tree Search. *Computers and Games* (S. 72-83). Lille, France: Springer.

Diedrich, O. (9. November 2015). Abgerufen am 11. Juli 2019 von heise online: https://www.heise.de/newsticker/meldung/Google-will-das-maschinelle-Lernen-voranbringen-2912530.html

Gauglitz, G., & Jürgens, C. (kein Datum). *Neuronale Netze - Einführung*. Abgerufen am 10. Juli 2019 von Neuronale Netze - Einführung: http://www.chemgapedia.de/vsengine/vlu/vsc/de/ch/13/vlu/daten/neuronalenetze/einfuehrung.vlu/Page/vsc/de/ch/13/anc/daten/neuronalenetze/snn1_6.vscml.html

Gers, F. A., Schmidhuber, J., & Cummins, F. (1999). Learning to Forget: Continual Prediction with LSTM. *Learning to Forget: Continual Prediction with LSTM*.

Haran, B. (1. Dezember 2013). Connect-Four-Numberphile. Abgerufen am 10. Juli 2019 von https://www.youtube.com/watch?v=yDWPi1pZ0Po

Haselhuhn, A. (12. Januar 2018). *Universität Leipzig.* Abgerufen am 10. Juli 2019 von Universität Leipzig: https://dbs.uni-leipzig.de/file/Haselhuhn_Hausarbeit.pdf

Hochreiter, S., & Schmidhuber, J. (1997). LONG SHORT-TERM MEMORY. *Neural Computation 9 (8)*, S. 1735-1780.

keras. (kein Datum). Abgerufen am 11. Juli 2019 von https://keras.io/

Levente Kocsis, C. S. (2006). Bandit Based Monte-Carlo Planning. *Machine Learning: ECML 2006* (S. 282-293). Springer.

Neural Networks Demystified [Part 3: Gradient Descent] (2014). [Kinofilm]. Abgerufen am 12. Juli 2019 von https://www.youtube.com/watch?v=5u0jaA3qAGk&feature=youtu.be

Peng, M. (17. Januar 2018). *The Startup*. Abgerufen am 7. Juli 2019 von https://medium.com/swlh/tic-tac-toe-at-the-monte-carlo-a5e0394c7bc2

Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., & et al. (9. Oktober 2017). Mastering the game of Go without human knowledge. (p. o. Macmillan Publishers Limited, Hrsg.) *Nature, 550*, S. 354 ff.

Skymind Inc. (kein Datum). *A Beginner's Guide to LSTMs and Recurrent Neural Networks*. Abgerufen am 9. Juli 2019 von A Beginner's Guide to LSTMs and Recurrent Neural Networks: https://skymind.ai/wiki/lstm

tensorflow. (kein Datum). Abgerufen am 11. Juli 2019 von https://www.tensorflow.org/

tensorflow. (kein Datum). Abgerufen am 11. Juli 2019 von https://www.tensorflow.org/guide/keras

Zell, A. (1994). *Simulation neuronaler Netze.* Bonn [u.a.]: Addison-Wesley.