# **Projektbericht: Maschinelles Lernen am Spiel „Vier-Gewinnt“**

### Business Analytics Project von Jonathan Cawalla, Lena Gräwe, Ahmad Haschemi und Lena Knickmeier

## 1. Einleitung

Im Rahmen des Business Analytics Projekt sollte anlehnend an die aktuellen Erfolge von Google mit ihrer künstlichen Intelligenz (KI) *AlphaGo* eine lernende KI implementiert werden. Die *AlphaGo-*KI weist im komplexen Spiel „Go“ eine Gewinnquote von 99,8% vor und schlug den amtierenden Europameister. [vgl. 1] Für das Business Analytics Projekt bestand die Aufgabe darin, ein vergleichsweise leichtes Spiel auszuwählen und eine bzw. zwei Methoden des Maschinellen Lernens darauf anzuwenden, um eine KI zu implementieren.

Als erster Schritt sollte ein klassisches Spiel ausgewählt werden, welches ein niedrigeres Komplexitätsniveau als Go hat. Die Auswahl fiel dabei auf das Spiel „Vier-Gewinnt“. Vier-Gewinnt ist ein Spiel bei dem zwei Spieler abwechselnd einen Stein auf dem Spielfeld platzieren. Das Spielfeld besteht aus sieben Spalten, in denen je sechs Steine Platz haben. Spielsteine könne nur von oben in die Spalten geworfen werden und fallen dann soweit nach unten wie möglich. Ein Spieler hat gewonnen, wenn er es schafft, vier seiner Steine direkt neben einander zu platzieren. Dabei ist es egal, ob sie horizontal, vertikal oder auf einer Diagonalen liegen. Es gibt mehrere Möglichkeiten, um die Komplexität eines Spiels zu messen.

Eine Möglichkeit ist, den Spielbaum eines Spiels zu betrachten. Jedes Spiel lässt sich durch einen zusammenhängenden und schleifenlosen Graph bestehend aus Knoten und Ästen abbilden. Für Entscheidungsknoten (mit nach unten weiterführenden Ästen) ist anzugeben, welcher Spieler hier seinen Zug gewählt hat. Der Wert eines Blattes bestimmt, ob der Spieler gewonnen oder verloren hat. Zur Evaluation der möglichen Züge wird ein Spielbaum rekursiv durchgesucht, der aus schätzungsweise möglichen Sequenzen von Zügen besteht, wobei b die Breite des Spiels (Anzahl der legalen Züge pro Position oder Verzweigungsfaktor) darstellt und d ist Tiefe des Spiels (Spieldauer). In den großen Spielen wie Schach gilt b≈35, d ≈80 und für Go gilt b ≈250, d ≈150 [vgl. 1]. Für das Spiel Vier-Gewinnt gilt b≈4, d≈36 [vgl. Victor Allis].

Ein anderer Maßstab für die Komplexität eines Spiels ist die Anzahl der möglichen Spielsituationen im Spiel. Für das klassische Vier-Gewinnt auf einem Spielfeld mit 6 Reihen und 7 Spalten gibt es 4.531.985.219.092 mögliche Positionen für alle Situationen von 0 bis 42 Steine im Spiel [vgl. 2. Ahmad].

## 2. Umsetzung der Trainingsumgebung

**Spielumgebung**

Um Zeit und Arbeit zu sparen, sollte mit einer fertigen Implementierung aus dem Internet begonnen werden. Das entsprechende Programm sollte bestenfalls leicht anpassbar sein und bereits eine einfache KI enthalten.

Nach ausgiebiger Recherche und kleinen Experimenten mit einigen der gefunden Programmen fiel die Auswahl auf eine Implementierung von Vier-Gewinnt die zunächst brauchbar schien [vgl. [hier](http://www.java-online.ch/gamegrid/gamegridEnglish/index.php?inhalt_mitte=gittergames/fourInARow.inc.php) verlinkt]. Um das Programm einsetzbar zu machen, wurden einige Änderungen vorgenommen, um die Spielfeldgröße über globale Variablen einstellen zu können. Außerdem sollte ein dritter Spielmodus eingeführt werden, um zwei KIs gegen einander spielen lassen zu können. Die Umsetzung dieses Modus barg jedoch mehr Tücken als ursprünglich angenommen. Das größte Problem bestand darin, dass, die Reihenfolge der Spieler vertauscht wurde. D.h. immer, wenn Spieler 1 an der Reihe sein sollte, wurde ein Stein von Spieler 1 positioniert und umgekehrt.

Da der Fehler nicht behoben werden konnte und nicht klar war in wie weit er sich auf den Lernerfolg der KIs auswirken würde, fiel letzten Endes der Entschluss, die bisherige Arbeit zu verwerfen und eine eigene Version von Vier-Gewinnt zu programmieren.

Die Klasse *Game* enthält jeweils eine globale Variable für die Spielfeldbreite, eine für die Spielfeldhöhe und eine für die Anzahl von Steinen, die nebeneinander gereiht werden müssen, um zu gewinnen. Das Spielfeld selbst wird in einem zwei dimensionales Array *board* gespeichert, auf das mit einem Getter zugegriffen werden kann. Um die Implementierung von Spiel und KIs zu erleichtern, enthält die Spielklasse außerdem je eine Indikator-Variable, um anzuzeigen, ob das Spielbrett leer ist und ob das Spiel beendet wurde.

Neben einigen Hilfsmethoden mit denen u.a. geprüft wird, ob das Spielende erreicht ist, das Spielbrett zurücksetzt werden kann oder Steine auf dem Spielfeld platziert werden, enthält das Spiel vier unterschiedliche Methoden für die Durchführung von Spielen:

* *PlayGame*Hierbei handelt es sich um eine Methode, die mit zwei übergebenen Spielern ein einzelnes Spiel durchführt und das Ergebnis auf der Konsole ausgibt.
* *generateDataset*Die Methode bekommt zwei Spieler übergeben und eine Anzahl an durchzuführenden Spielen übergeben und gibt nach entsprechend vielen Spielen ein Daten-Set zurück.
* *trainQPlayer*Die Methode bekommt zwei Spieler und die Anzahl zu spielender Spiele übergeben. Außerdem erhält sie eine Information darüber, ob die Spieler abwechselnd beginnen sollen oder nicht.
* *playTournament*Die Methode erhält die gleichen Eingabe-Parameter wie *trainQPlayer*. Auch die Funktionsweise ist im Wesentlichen gleich, mit dem Unterschied, dass falls ein Q-Player am Turnier teilnimmt, keine Werte in seiner Datenbank mehr verändert werden. Diese Methode dient somit vor allem zur Beurteilung der Trainingsergebnisse. Auf der Konsole werden Spielzüge, Ergebnisse von einzelnen Spielen sowie summierte Kennziffern über das gesamte Turnier ausgegeben.

**NormalKI und NormalKI2**

Damit die KIs später automatisiert lernen können, wird eine weitere KI benötigt, die bereits spielfähig ist. Das oben erwähnte Vier-Gewinnt-Spiel enthält bereits eine gute KI, deren Algorithmus für die neue Spiel-Implementierung übernommen wurde.

Der enthaltene Algorithmus basiert auf einfachen if-then-Abfragen und Schleifen. Dabei wird zunächst geprüft, ob das Spielfeld leer ist. In diesem Fall wirft sie in die mittlere Spalte, um eine möglichst gute Ausgangslage zu schaffen. Wenn bereits Steine auf dem Spielfeld liegen, prüft die KI als erstes, ob sie selbst gewinnen kann. Falls nicht, wird abgefragt, ob der Gegner gewinnen kann und der eigene Stein entsprechend so geworfen, um dies zu verhindern.

Sofern keiner dieser Fälle vorliegt, wird eine defensive Spielweise gewählt. Für ein klassisches Vier-Gewinnt-Spiel wird nun geprüft, welche Züge es dem Gegner erlauben würden drei bzw. zwei seiner Steine nebeneinander zu platzieren. Alle entsprechenden Züge werden als Spalten in einem Array *possibleSolutions* zwischengespeichert. Da Züge, die es ermöglichen drei Steine neben einander zu platzieren, natürlich auch zwei Steine neben einander zur Folge haben, werden sie doppelt in das Array aufgenommen. Anschießend werden alle Züge aus *possibleSolutions* entfernt, die es dem Gegner in seinem nächsten Zug ermöglichen (über eine Diagonale oder Horizontale) zu gewinnen. Im nächsten Schritt werden alle verbleibenden Spalten, die nicht am Rand liegen, ein weiteres Mal in das Array kopiert, um dann einen zufälligen Zug auszuwählen. Falls *possibleSolutions* leer ist, wird nach einer Lösung gesucht, die in erster Linie erlaubt ist und dem Gegner nach Möglichkeit nicht ermöglicht, im nächsten Zug zu gewinnen.

Die Klasse *NoramlKI2* enthält außerdem eine Variable *epsilon*, die festlegt wie viel Prozent der Spielzüge rein zufällig ausgewählt werden sollen. Diese Strategie ist bekannt als epsilon-greedy-Strategie und soll eine größere Varianz an Spielzügen ermöglichen. Dies ist der einzige Unterschied zu *NormalKI.*

**Datenbank und Binär-Codierung**

Da die Anzahl der möglichen Spielzustände sehr hoch ist, wird eine effiziente Speicherstruktur für den Q-Player sehr wichtig. Es müssen beliebig viele Zustände gespeichert werden können und der Zugriff auf die Zustände muss sehr schnell ablaufen, da der Q-Player während eines Spieldurchlaufs in jedem Zug sowohl für diesen Zug, als auch für die nächsten Züge Abfragen an die Datenbank generiert. Aus diesem Grund wurde eine HashMap gewählt, die intern von der Klasse *Q\_DB* verwendet wird. Der Schlüssel dieser HashMap ist ein Spielzustand. Der Spielzustand wird von der Klasse *Game* als 2-dimensionales Int Array gespeichert. Da dieser Datentyp sich aufgrund einer fehlenden Hash- und Equalsfunktion nicht als Schlüssel eignet, wird jeder Spielzustand in einen *Array2DWrapper* umgewandelt, der diese Funktionen implementiert. Als Wert für die HashMap wird wiederrum eine neue Datenstruktur verwendet, die eine Aktion und den Wert für diese Aktion enthält. Die Aktion ist die Spalte in die geworfen wird und der Wert ist abhängig von der internen Bewertung des Q-Players und wird von diesem verändert. Die dafür verwendete Datenstruktur ist ebenfalls eine HashMap. Es könnte auch eine ArrayList oder ähnliches verwendet werden, da hier nur eine maximale Anzahl von sieben Aktionen und damit sieben Einträgen möglich ist.

Die Datenbank implementiert weiterhin folgenden Methoden mit denen der Q-Player seine Werte abfragen und verändern kann:

* *get(..)*, *containsState(..)* für Abfragen an die Datenbank
* *getValueOfStateAndAction(..)*: Liefert für einen bestimmten Zustand und die Aktion, die darauf ausgeführt werden soll, den Wert dieser Aktion zurück.
* *put(..)* und *update(..)* zum Verändern der Datenbank.
* Sowie Funktionen zum Speichern und Laden der Datenbank über die Serialisierung der HashMap.

Das Speichern und Laden der HashMap wurde inbesondere dafür verwendet, die Datenbank für den Q-Player nach einer gewissen Anzahl an Trainingsiterationen zu speichern. Damit kann das Training zum späteren Zeitpunkt fortgesetzt werden, indem die Datenbank wieder geladen wird.

## Q-Learning

**Methode**

Beim Q-Learning handelt es sich um eine einfache Form des Reinforcement Learning. Das Prinzip basiert darauf, dass der KI in jedem Zustand eine Reihe von Aktionen zur Auswahl stehen. Je nachdem, welche Aktion sie wählt, erhält sie eine Belohnung oder Bestrafung, die sich zum einen Teil aus dem unmittelbar reichten Zustand und zum andern aus den später erreichbaren Zuständen ableiten lassen. Indem die KI versucht ihre Belohnungen zu maximieren, lernt sie über mehrere Durchläufe, was die bestmögliche Spielstrategie ist. [vgl.1]

Im Falle des Spiels Vier-Gewinnt definieren sich die Zustände durch die aktuelle Position aller Steine auf dem Spielbrett. Die Aktion der KI ist das Werfen eines Spielsteins in eine der Spalten.

**Verwandte Arbeiten**

Olszowka beschäftig sich in seiner Arbeit neben einer allgemeinen Implementierung von Q-Learning für Vier-Gewinnt mit den damit verbundenen Schwierigkeiten, die ihm in einer früheren Arbeit begegnet sind. Unter anderem geht er auf die große Anzahl von Spielzuständen beim klassischen Vier-Gewinnt und beschäftigt sich mit Methoden, um die Anzahl der für den Algorithmus relevanten Zustände zu reduzieren. [vgl. 2]

Șimșek und Barto stellen neben viele anderen mit Q-Learning gelösten Problemen eine KI für Tic-Tac-Toe vor. [vgl. 3] Tic-Tac-Toe ist zwar auf Grund der kleineren Spielfeldgröße weniger komplex als Vier-Gewinnt, die Spielweise ist jedoch ähnlich.

**Implementation des Q-Players**

Für die Umsetzung des Q-Players diente eine Anleitung aus dem Internet als grobe Orientierung. [vgl. ]

Für die Verteilung der Werte der einzelnen Zustände diente die Folgenden Formel als Ausgangspunkt:

Dabei ist alpha der Prozentsatz für den der alte Wert aus der Datenbank bestehen bleiben soll. Es wurde ein Wert von 0,95 gewählt, sodass eine 5 % Veränderung möglich ist, sollte der Q-Player, denselben Zustand noch einmal betreten und den Wert verändern wollen. Das verhindert, dass der Wert jedes Mal überschrieben wird, sobald er neu berechnet wird. Stattdessen konvergiert dieser mit steigender Lernzeit gegen einen festen Wert. Gamma ist der Lernparameter mit dem gewichtet wird, wie stark zukünftige Züge in die Bewertung einfließen. Es wurde ein Wert von 0,8 gewählt. Die Belohnung wird verteilt, sobald das Spiel gewonnen ist.

Um dieses Verfahren an das Vier-Gewinnt Spiel anzupassen, wurden mehrere Veränderungen gemacht. Es wurde nicht nur eine Belohnung für ein gewonnenes Spiel, sondern auch eine Bestrafung für ein verlorenes Spiel gesetzt. Diese werden immer dann verteilt, wenn ein Spiel abgeschlossen ist. Der Q-Player wird über die Methode *reactToWinOrLose* darüber informiert und schreibt den neuen Wert in die Datenbank. Dies passiert getrennt von der Berechnung der restlichen Formel, da immer dann, wenn ein Spiel abgeschlossen ist, die Berechnung von nicht mehr nötig ist, da es keine zukünftigen Züge gibt.

Die Formel wurde für das 2-Spieler-Spiel angepasst. Im ursprünglichen Q-Learning wird ein 1-Spieler-Spiel betrachtet, sodass der Spieler nach einem Zug direkt wieder am Zug ist und daher direkt die nächsten Züge aus der Datenbank gelesen werden können. Bei Vier-Gewinnt muss erst der gegnerische Spieler seinen Stein platzieren, bevor der Q-Player selbst wieder an der Reihe ist. Die Berechnung erfolgt in der Methode *turn* und wurde wie folgt angepasst:

wird in der Methode *avgValueForNextStateAllActions* über eine Simulation der nächsten möglichen Züge berechnet. Zunächst werden alle möglichen Züge für den Q-Player generiert. Das Generieren von Zügen wird ermöglicht durch die Methode *generateActions*. Basierend auf diesen Zügen werden die nächsten Züge für den Gegner berechnet. Es wird nur eine Heuristik verwendet, um einen dieser Züge auszuwählen. Dafür wird der Spielzustand invertiert und in der Datenbank des Q-Players geschaut, ob dieser Züge für diesen Spielzustand vorhanden sind. Falls ja, dann wird derjenige mit dem höchsten Wert ausgewählt. Damit wird nur die bestmögliche Aktion für den Gegner betrachtet, was sinnvoll erscheint, da dieser immer diese Aktion machen würde und nicht eine beliebige andere. Falls nein, wird ein zufälliger Zustand ausgewählt. Anschließend werden für diese ausgewählte Aktion des Gegners alle möglichen Züge des Q-Players generiert. Es wird der Durchschnitt der Werte all dieser Aktionen gebildet und in gespeichert. Der Durchschnitt wird aus dem Grund gebildet, da ansonsten die Bestrafungen nicht mit in den Q-Wert einbezogen werden.



Abbildung

## Neuronales Netz

**Methode**

In Sinne des Maschinellen Lernens handelt es sich bei künstlichen Neuronalen Netzen um eine Struktur, welche aus Neuronen und die Neuronen miteinander verbindenden Synapsen besteht, ähnlich dem Zentralnervensystem von Lebewesen. Künstlich Neuronale Netze werden für Einschätzungen und Approximationen von Funktionen angewendet, die mit einer großen Menge von Input-Elementen abhängig sind. [vgl. 1] Diese Neuronalen Netze können durch drei Eigenschaften festgelegt werden: der Architektur, der Aktivierungsregel und den Lernregeln.

Die Architektur bestimmt, welche Variablen im Neuronalen Netz und seiner Topologie beteiligt sind. Die Anzahl der Neuronen in den verdeckten Schichten (Hidden-Layer) sind beispielweise variierbar, im Gegensatz zu der Eingabe- und Ausgabeschicht (Input- und Output-Layer). Es wurde in dieser Arbeit ein Multi-Layer Perzeptron mit einer verdeckten Schicht (Hidden-Layer) ausgesucht.

Die Aktivierungsregel bestimmt, wie sich die Aktivitäten der Neuronen in Reaktion aufeinander verändern. Z. B. verfüget jede verdeckte Schicht und die Ausgabeschicht über eine (eigene) Aktivierungsfunktion. Diese können linear oder nicht linear sein. Nicht lineare Aktivierungsfunktionen machen das Neuronale Netz besonders mächtig. [vgl. 2]

In diesem Projekt wurde die Sigmoidfunktion als Aktivierungsfunktionen angewendet, da der Einsatz von differenzierbaren Funktionen die Verwendung von Lernmechanismen (Backpropagation-Algorithmus) ermöglicht. Als Aktivierungsfunktion eines Neurons wird die Sigmoidfunktion auf die Summe der gewichteten Eingabewerte angewendet, um die Ausgabe des Neurons zu erhalten. Die Sigmoidfunktion wird vor allem aufgrund ihrer einfachen Differenzierbarkeit als Aktivierungsfunktion bevorzugt verwendet. [vgl. 2]

Die Lernregeln bestimmen die Art und Weise, in der die Gewichte des Netzes durch Training im Laufe der Zeit angepasst werden. Es wurde hier Backpropagationals Lernalgorithmus angewendet, welches ein verbreitetes Verfahren für das Erlernen von künstlichen neuronalen Netzen ist. [vgl. 3]

**Verwandte Arbeiten**

**Umsetzung Neuronales Netz als *NNPlayer2***

Nach ersten Recherchen zum Thema Neuronale Netze fiel die Entscheidung auf das Framework „Neuroph 2.7“ von SourceForge, welches eine Java Bibliothek als auch die GUI-Anwendung „Neuroph Studio“ bietet. Mit der Java Bibliothek können Neuronale Netze in Java Programmen erzeugt, trainiert und angewendet werden [vgl. 1]. Es gibt außerdem eine Anleitung zur Erzeugung von Multi-Layer Perzeptronen [vgl. 2], die für das weitere Vorgehen hilfreich war, weil sich an dem Vorgehen von Schneider und Rosa [vgl. 3] orientiert wurde. Für das Neuronale Netz wurde die Struktur von Multi-Layer Perzeptron mit einem Hidden-Layer angewandt. Für das Lernen wurde Backpropagation und die Sigmoid-Funktion als Aktivierungsfunktion gewählt, wie auch bei Schneider und Rosa [vgl. 3]. Danach berechnet sich die Anzahl der Neuronen im Input-Layer als Produkt aus Spalten- und Zeilen-Anzahl, multipliziert mit drei, denn jede Zelle des Spielfelds wird mit drei Eigenschaften beschreiben. Die Anzahl ist also abhängig von der Spielfeldgröße. Eine genauere Erläuterung folgt im nächsten Abschnitt. Die Anzahl der Neuronen im Hidden-Layer kann durch Ausprobieren ermittelt werden. Im späteren Training und Testverlauf wurden mehrere Neuronale Netze mit unterschiedlicher Neuronenanzahl im Hidden-Layer erstellt und deren Auswirkung erfasst. Dabei wurde die Anzahl der Input-Neuronen mit einem Faktor i multipliziert, wobei für Faktor i unterschiedliche Werte gewählt wurden (s. Kapitel 5). Die Anzahl der Neuronen im Output-Layer ergibt sich aus der Anzahl der Spalten des jeweiligen Spielfelds.

Damit das Neuronale Netz lernen kann, werden Daten benötigt. Hierfür wurde die Methode *generateDataSets* in der Klasse *Game* implementiert. Durch Aufruf dieser in der *main*-Methode der Klasse *Game* wird eine bestimmte Anzahl von Spielen zweier *IPlayer* ausgeführt und deren jeweiligen Zustände in eine Textdatei geschrieben, wobei hier zu beachten ist, dass nur neue, unbekannte Zustände in die Datei geschrieben werden. Außerdem werden nur die Zustände von einem bestimmten Spieler und nicht vom gegnerischen Spieler gespeichert.

Bei der Generierung der Textdatei als Daten-Set werden die jeweiligen Spielzustände in Input- und Output-Elemente übersetzt, die das Neuronale Netz später einlesen und im Lernprozess verarbeiten kann. Hierbei wurde die Idee von Schneider und Rosa übernommen, die Input-Elemente binär darzustellen. [vgl. 3] So wird ein einzelnes Feld auf dem Spielfeld durch eine Folge von drei Ziffern ausgedrückt, deren Wert beim Zutreffen eines bestimmten Merkmals eins und bei Nicht-Zutreffen null ist.

Die erste Stelle dieser Ziffern-Folge beschreibt, ob das Feld leer ist, die zweite Stelle, ob dieses Feld von einem Stein des Spieler 1 belegt und die dritte Stelle, ob das Feld von einem Stein des Spieler 2 belegt wird. So steht die Folge [1, 0, 0] für ein leeres Feld, die Folge [0, 1, 0] für ein Feld mit einem Stein des Spieler 1 und die Folge [0, 0, 1] für ein Feld mit einem Stein des Spieler 2. Eine Zeile in der Textdatei einspricht einem Spielzustand (Input für das Neuronale Netz) und die Aktion (Output für das Neuronale Netz) darauf, d. h. es gibt bspw. für ein 6x7 Feld eine Ziffernfolge von 133 Ziffern, wobei die ersten 126 Ziffern den Spielzustand (Input) und die letzten 7 Ziffern die entsprechende Aktion (Output) darstellen. Die sieben Ziffern der Aktion bestehen aus einer Folge von Nullen und Einsen. Diese Folge beschreibt mit einer Eins, in welche Spalte der Stein geworfen wurde, und füllt die restlichen sechs Spalten mit Nullen auf, damit eine eindeutige Spalte zugewiesen werden kann. Ein solches Dataset kann später im Lernprozess des Neuronalen Netzes eingelesen und mit diesen Input-Elementen trainiert werden.

Damit das Neuronale Netz für eine Spielteilnahme genutzt werden kann, wurde die Klasse *NNPlayer2* implementiert. Im Konstruktor dieser Klasse muss manuell angegeben werden, ob durch Aufruf der Methode *learnNNPlayer* ein neues Neuronales Netz erzeugt und trainiert wird oder ob ein schon vorhandenes Netz geladen wird, indem der Name dieses Netzes eingegeben wird.

In der *learnNNPlayer*-Methode wird ein zuvor erstelltes Daten-Set zum Importieren eingegeben und ein auf die jeweilige Spielsituation angepasstes Multi-Layer Perzeptron wird erzeugt. Hierbei können auch passende Werte für den maximalen Fehler, die Lernrate und das Momentum angegeben werden. Das Neuronale Netz wird nach Generierung abgespeichert, damit es zu einem späteren Zeitpunkt geladen werden kann.

Im Spielverlauf selbst wird die *turn*-Methode der Klasse *NNPlayer2* aufgerufen, um einen Spielzug durchzuführen. Hier wird der aktuelle Spielzustand als Input an das an den *NNPlayer2* geknüpfte Netz übergeben. Das Netz kalkuliert daraus dann einen Output und gibt zurück in welche Spalte der nächste Stein geworfen werden soll.

## Test und Auswertung

Um die Funktion der KIs einfacher prüfen zu können, wurden Training und Tests für beide KIs zunächst auf einem 4x5 Feld als „Drei-Gewinnt“ durchgeführt, bevor der Prozess für ein klassisches 6x7 „Vier-Gewinnt“-Feld wiederholt wurde. Für beide Spieler diente die NormalKI als Gegner. Im Folgenden wird zunächst auf die Methode des Q-Learning eingegangen und danach auf die Methode der Neuronalen Netze.

**Q-Learning 4x5**

Der Q-Player wurde mit einem *epsilon* von 0,2 und einem *alpha* von 0,05 schrittweise in insgesamt 128.000 Spielen trainiert. Der anschließende Test wurden in fünf Turnieren mit je 10.000 Spielen durchgeführt.

Abbildung 2

Wie zu erwarten nimmt die Anzahl der Datenbank Elemente mit jedem Training zu, die Anzahl der unbekannten Spielfeldzustände je 10.000 Spiele sinkt. Beides geschieht mit abnehmender Änderungsrate. [siehe Abb. 2]

Abbildung 3

Die Lernkurve steigt zunächst stark an und erreicht ihren Höhepunkt bei 32.000 Trainingsspielen. In diesem Stadium gewinnt sie bei abwechselndem Spielbeginn bis zu 81,8% (auf einer Basis von 10.000 Spielen) und 100% (auf einer Basis von 100.000), wenn sie bei allen Spielen den ersten Zug machen darf. Überraschenderweise führte jeder Versuch, sie weiter zu trainieren, zu einer Verschlechterung der Spielperformance. Dabei wurden sowohl die Parameter als auch die Trainingsintervalle geändert. [siehe Abb. 3]

Hinzuzufügen ist, dass die Anzahl der unentschiedenen Spiele im Schnitt bei 2,56 von 10.000 Spielen liegt und somit vernachlässigbar ist. Genauso liegt der Unterschied zwischen maximal und minimal gewonnen Spielen pro Trainingsstufe im Schnitt bei nur 0,71%.

Besonders interessant ist, dass die KI eine Art strategisches Vorausdenken zeigt, indem sie Zwickmühlen baut. Sie bringt den Gegner also in eine Lage, in der er seine Niederlage nicht ehr verhindern kann. Die folgenden drei Spielausschnitte stammen vom Q-Player mit 32.000 Trainingsspielen und sollen dieses Verhalten veranschaulichen. (Q-Player = ■ NormalKI=X)

|0|0|0|0|0| |0|0|0|0|0| |0|0|0|0|0| |0|0|0|0|0|

|0|0|X|0|0| |0|0|X|0|0| |0|0|X|0|0| |■|0|X|0|0|

|0|0|■|0|0| |0|■|■|0|0| |X|■|■|0|0| |X|■|■|0|0|

|■|X|■|X|0| |■|X|■|X|0| |■|X|■|X|0| |■|X|■|X|0|

*Abbildung 4*

Der Q-Player hat hier eine Situation geschaffen, in der er in der zweiten Zeile sowohl links als auch rechts seinen dritten Stein platzieren kann, um zu gewinnen. Dazu kommt, dass der Gegner, wenn er seinen Stein in die linke Spalte wirft, dem Q-Player die Gewinnmöglichkeit über die Diagonale gibt.

|0|0|0|0|0| |0|0|0|0|0| |0|0|0|0|0| |0|0|0|0|0|

|0|X|X|0|0| |0|X|X|0|0| |0|X|X|0|0| |0|X|X|0|0|

|0|■|■|0|0| |0|■|■|0|0| |0|■|■|X|0| |0|■|■|X|0|

|0|X|■|0|0| |0|X|■|■|0| |0|X|■|■|0| |0|X|■|■|■|

Abbildung 5

In diesem Spiel positioniert der Q-Player seine Steine so, dass er sowohl in der untersten als auch in der darüber liegenden Zeile eine Dreierreihe vervollständigen kann.

|0|X|0|0|0| |0|X|0|0|0| |0|X|0|0|0| |0|X|0|■|0|

|0|X|■|0|0| |0|X|■|0|0| |0|X|■|X|0| |0|X|■|X|0|

|0|■|X|0|0| |0|■|X|■|0| |0|■|X|■|0| |0|■|X|■|0|

|X|■|X|■|0| |X|■|X|■|0| |X|■|X|■|0| |X|■|X|■|0|

*Abbildung 6*

Auch hier hat der Q-Player im zweiten gezeigtenZzustand zwei Möglichkeiten zu gewinnen. Zum einen in der vierten Spalte. Zum anderen über eine Diagonale von Spalte drei bis fünf. Zusätzlich ermöglicht der Gegner eine weitere Option, wenn er in die vierte Spalte wirft.

**Q-Learning 6x7**

Den Q-Player für ein 6x7 Spielfeld zu trainieren ist hingegen sehr viel schwerer. Zwar lernt der Spieler erkennbar. Jedoch benötigt das Training bei dieser Spielfeldgröße deutlich mehr Spiele; mit 3,2 Mio. Trainingsspielen liegt die Performance der KI hier gerade einmal bei durchschnittlichen 11,2%. [siehe Abb. 7]

Abbildung 7

Das eigentliche Problem stellt hier jedoch nicht die Anzahl der benötigten Trainingsspiele dar, sondern die große Anzahl an Spielzuständen und Datenbank-Elementen. [siehe Abb. 8] Nach 1,6 Mio. Spielen enthält die Datenbank gut 11 Mio. Elemente und ist 4,3 GB groß. Nach 3,2 Mio. Spielen belief sich die Anzahl der Elemente auf ca. 21 Mio. Eine genaue Speichergröße liegt nicht vor, da die Datenbank auf Grund fehlender Arbeitsspeicherkapazität nicht gespeichert werden konnte. Eine Schätzung auf 8,2 GB scheint aber realistisch.

Ein weiteres Training auf 4,8 Mio. bzw. 6,4 Mio. Trainingsspielen wurde zwar versucht, jedoch abgebrochen, da der Trainingsprozess zunehmend langsamer wurde. Zuletzt lag die Anzahl der durchgeführten Spiele bei ca. 200 pro Minute, womit das Training mehre Tage gedauert hätte. Es ist sogar denkbar, dass es mehrere Wochen gedauert hätte, da die Geschwindigkeit mit Sicherheit weiter abgenommen hätte.

Abbildung 8

Trotz der geringen Erfolgsquote nach 3.2 Mio. Lerndurchläufen lassen sich einige geschickte Züge in den Turnierspielen finden. Genauso wie auf dem 4x5 Feld konstruiert der Q-Player Zwickmühlen, um seinen Gegner zu besiegen.

|  |  |  |  |
| --- | --- | --- | --- |
| |0|0|0|0|0|0|0|  |0|0|0|0|0|0|0|  |0|0|0|X|0|0|0|  |0|0|0|■|0|0|0|  |0|■|0|■|0|0|0|  |X|X|■|X|X|0|0| | |0|0|0|0|0|0|0|  |0|0|0|0|0|0|0|  |0|0|0|X|0|0|0|  |0|0|0|■|0|0|0|  |0|■|■|■|0|0|0|  |X|X|■|X|X|0|0| | |0|0|0|0|0|0|0|  |0|0|0|0|0|0|0|  |0|0|0|X|0|0|0|  |0|0|0|■|0|0|0|  |X|■|■|■|0|0|0|  |X|X|■|X|X|0|0| | |0|0|0|0|0|0|0|  |0|0|0|0|0|0|0|  |0|0|0|X|0|0|0|  |0|0|0|■|0|0|0|  |X|■|■|■|■|0|0|  |X|X|■|X|X|0|0| |

*Abbildung 9*

Abbildung 9 zeigt eine einfache Form der Zwickmühle, bei der der Q-Player eine Reihe aus drei seiner Steine mit zwei offenen Enden erzeugt. Ähnliche Spielsituationen finden sich mehrfach in den aufgezeichneten Turnierausschnitten.

|  |  |  |  |
| --- | --- | --- | --- |
| |0|0|X|0|0|0|0|  |0|0|■|0|0|0|0|  |0|X|■|X|0|0|0|  |0|■|■|■|0|0|0|  |0|■|X|X|0|0|0|  |X|X|■|X|0|0|0| | |0|0|X|0|0|0|0|  |0|0|■|■|0|0|0|  |0|X|■|X|0|0|0|  |0|■|■|■|0|0|0|  |0|■|X|X|0|0|0|  |X|X|■|X|0|0|0| | |0|0|X|0|0|0|0|  |0|0|■|■|0|0|0|  |0|X|■|X|0|0|0|  |0|■|■|■|0|0|0|  |X|■|X|X|0|0|0|  |X|X|■|X|0|0|0| | |0|0|X|0|0|0|0|  |0|0|■|■|0|0|0|  |0|X|■|X|0|0|0|  |■|■|■|■|0|0|0|  |X|■|X|X|0|0|0|  |X|X|■|X|0|0|0| |

*Abbildung 10*

Im zweiten Beispiel ist eine komplexere Strategie zu sehen. Der Q-Player positioniert zunächst drei seiner Steine auf einer Diagonalen. Der Gegner versucht im nächsten Zug den Sieg des Q-Players zu verhindern und wirft seinen Stein in die linke Spalte. Dadurch wird zwar der Sieg über die Diagonale für den Q-Player verhindert, gleichzeitig ist es ihm nun aber möglich eine Vierer-Reihe in der Horizontalen zu vervollständigen.

**NNPlayer2 4x5**

Zunächst wurden für ein Spielfeld der Größe 4x5 im „Drei-Gewinnt“-Spiel mehrere Datasets erstellt, sowohl für Spiele des 1. Spielers (Beginner eines Spiels) als auch Spiele des 2. Spielers. Die Zeileneinträge der Daten-Sets wurden jeweils auf 200, 300 und 400 Einträge beschränkt, wobei nur unterschiedliche Einträge gespeichert werden, die aber zufällig in einem Turnier von zwei *NormalKI*-Spielern, die gegeneinander spielten, erzeugt wurden. Es wurden so insgesamt sechs Datasets erzeugt, jeweils drei pro Spieler. Es wurden mehrere Neuronale Netze erzeugt, die auf den zuvor generierten Datasets trainiert wurden. Dabei lag maximale Fehler bei 0,01, die bei Lernrate 0,2 und das Momentum bei 0,7. Um die Auswirkung der Neuronen-Anzahl im Hidden-Layer näher zu betrachten, wurden dafür unterschiedlich Anzahlen übergeben, indem die Anzahl der Neuronen im Input-Layer mit einem Faktor i= {½, 1, 2, 3, 4} multipliziert wurden. Dies entspricht einer Neuronen-Anzahl von 30, 60, 120, 180 und 240. Bei dieser Vorgehensweise wurde sich an der Arbeit von Schneider und Rosa orientiert [vgl. 1].

Im weiteren Verlauf wurden mehrere Turniere des *NNPlayer2* mit den unterschiedlichen Neuronalen Netzen gegen die *NormalKI* gespielt, wobei pro Turnier 10.000 Spiele gespielt wurden.

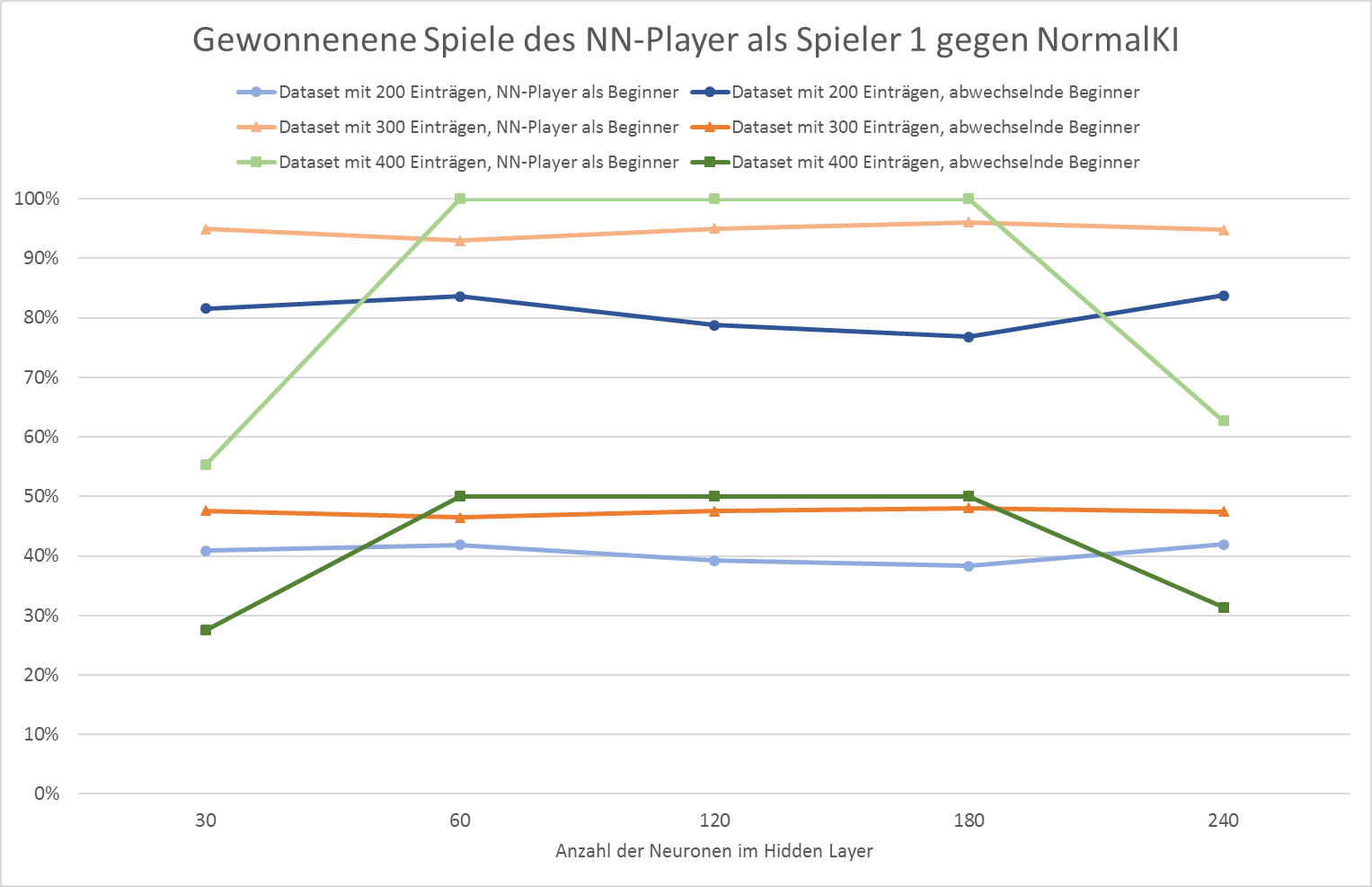


Abbildung 11

Abbildung 11 zeigt die Anzahl der im Mittel gewonnenen Spiele des *NNPlayer2* mit Neuronalen Netzen unterschiedlicher Neuronen-Anzahl im Hidden-Layer gegen die *NormalKI*. Die angewendeten Netze, wurden jeweils auf den Datasets von Spieler 1 trainiert.

Es ist erkennbar, dass sich die Kurven der NN-Player, die auf dem gleichen Dataset gelernt haben, aber unterschiedlich im Turnier beginnen, ähneln. Sehr eindeutig sieht man das auf den Kurven des Datasets mit 400 Einträgen: Die schlechtesten Ergebnisse liefert hier das Neuronale Netz mit 30 Neuronen im Hidden-Layer, während die Netze mit 60, 120 oder 180 Neuronen im Abwechselnd-Modus 50 % der Spiele gewinnt bzw. als Beginner der Spiele sogar 100 % der Spiele gewinnt, aber bei der Anzahl von 240 Neuronen wieder sinkt. Die Netze, die auf diesem Dataset mit 400 Einträgen gelernt haben und eine Neuronen-Anzahl von 60, 120 oder 180 aufweisen, können also vermutlich sehr gut einschätzen, wie sich die *NormalKI* als Spieler verhalten wird. Dass im Abwechselnd-Modus nur noch 50 % der Spiele gewonnen werden, statt 100 % als Beginner der Spiele, deutet daraufhin, dass das Netz alle Vorgehensweisen der *NormalKI* imitiert, also auch die des Verlierens. Was durchaus möglich ist, da in den Datasets sowohl gewonnene, verlorene als auch unentschiedene Spiele verzeichnet werden konnten.

Wenn man sich die Ergebnisse der Daten-Sets mit 200 und 300 Einträgen ansieht, so sind die Abweichungen der gewonnenen Spiele eines jeweiligen Daten-Sets im Vergleich zur Neuronen-Anzahl im Hidden-Layer meist nah beieinander und es sind nur leichte Schwankungen zu verzeichnen. Demnach ist ein Neuronales Netz, das gute Ergebnisse liefert, eines, welches auf dem Daten-Set mit 400 Einträgen gelernt hat und einer Neuronen-Anzahl von 60, 120 oder 180 im Hidden-Layer aufweist.

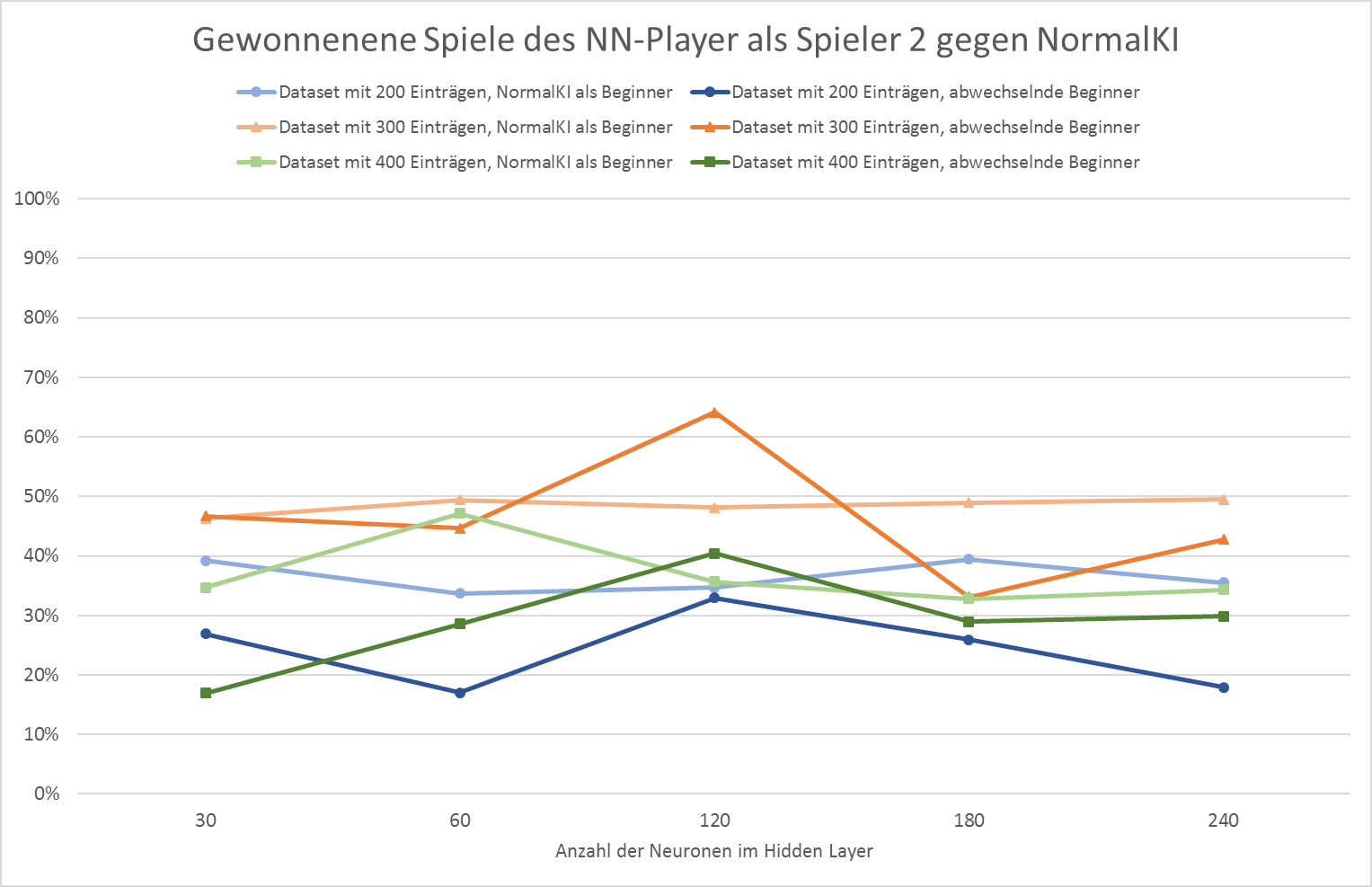


Abbildung 12

Abbildung 12 zeigt die Ergebnisse der im Mittel gewonnenen Spiele des *NNPlayer2* mit Neuronalen Netzen unterschiedlicher Neuronen-Anzahl im Hidden-Layer gegen die *NormalKI*, wobei hier die angewandten Netze, jeweils auf den Datasets von Spieler 2 gelernt haben. Zuerst fällt auf, dass die Ergebnisse insgesamt schlechter ausfallen als die Ergebnisse bei den Datasets mit Einträgen des 1. Spielers, welches die Vermutung stützt, dass der 1. Spieler (Beginner des Spiels) im Vorteil ist. Einzig das Neuronale Netz mit 120 Neuronen im Hidden-Layer, das auf einem Dataset mit 300 Einträgen des 2. Spielers gelernt hat, kann im Turnier eine Gewinnquote von 64 % im Abwechselnd-Modus aufweisen, während alle anderen Netze die 50 %-Schwelle nicht überschritten. Diese Beobachtung lässt vermuten, dass das Neuronale Netz aufgrund der Datasets des 2. Spielers wahrscheinlich hauptsächlich gelernt hat, auf eine Spielsituation zu reagieren und daher vermutlich eher die Strategie verfolgt die Gewinnchancen des Gegners zu verhindern, um diese Vermutung zu stützen, müsste man sich allerdings die Spielverläufe genauer ansehen.

Es ist außerdem zu beobachten, dass im Abwechselnd-Modus für die Neuronalen Netze, die auf den drei unterschiedlichen Datasets gelernt haben, eine Neuronen-Anzahl von 120 im Hidden-Layer im Test die besten Ergebnisse lieferten.

Spielt der *NNPlayer2* aber nur als 2. Spieler und die *NormalKI* beginnt im Turnier (Beginner-Modus für die *NormalKI*), so sind die Beobachtungen nicht mehr so eindeutig und schwanken stärker, was die Brauchbarkeit Neuronen-Anzahl im Hidden-Layer betrifft.

Im Test der Neuronalen Netze gab es ein paar Turnierverläufe, in denen es auch unentschiedene Spiele gab, allerdings waren hier nur die Spiele betroffen, von den Datasets mit 300 und 400 Einträgen, welche vom 2. Spieler aufgezeichnet wurden und hier waren maximal 6 % der Spiele in einem Turnier unentschieden und ist nicht sehr aussagekräftig, da es sein kann, dass es bei Generierung der anderen Datasets vielleicht nie zu unentschiedenen Spielen kam und das Netz diesen Spielverlauf vielleicht gar nicht kennt.

Wenn man sich ein paar Spielverläufe näher ansieht, fällt auf, dass der NNPlayer2 die Strategie der Zwickmühlen gelernt hat und dadurch ein Spiel für sich entscheiden kann. Die folgenden Spielverläufe beziehen sich auch einen NNPlayer2. Bei dem Neuronalen Netz handelt es sich um das, welches 240 Neuronen im Hidden-Layer hat und auf einem Daten-Set vom 1. Spieler mit 200 Einträgen gelernt hat. ( NNplayer2 = ■ )

|  |  |  |
| --- | --- | --- |
| |0|0|0|0|0|  |0|0|0|0|0|  |0|0|0|0|0|  |0|0|■|X|0| | |0|0|0|0|0|  |0|0|0|0|0|  |0|0|0|■|0|  |0|0|■|X|0| | |0|0|0|0|0|  |0|0|0|0|0|  |0|0|0|■|0|  |0|0|■|X|0| |
| |0|0|0|0|0|  |0|0|0|0|0|  |0|0|0|■|■|  |0|0|■|X|X| | |0|0|0|0|0|  |0|0|0|0|0|  |0|0|X|■|■|  |0|0|■|X|X| | |0|0|0|0|0|  |0|0|0|0|■|  |0|0|X|■|■|  |0|0|■|X|X| |
| *Abbildung 13* | | |

In Abbildung 13 ist ein Spiel zu sehen, in dem der NNPlayer2 beginnt und innerhalb weniger Züge eine Zwickmühle baut.

|  |  |  |
| --- | --- | --- |
| |0|0|0|0|0|  |0|0|■|0|0|  |0|0|X|0|0|  |0|■|■|X|0| | |0|0|0|0|0|  |0|0|■|0|0|  |0|0|X|0|0|  |X|■|■|X|0| | |0|0|0|0|0|  |0|0|■|0|0|  |■|0|X|0|0|  |X|■|■|X|0| |
| |0|0|0|0|0|  |0|0|■|0|0|  |■|X|X|0|0|  |X|■|■|X|0| | |0|0|0|0|0|  |0|■|■|0|0|  |■|X|X|0|0|  |X|■|■|X|0| | |0|0|0|0|0|  |0|■|■|0|0|  |■|X|X|X|0|  |X|■|■|X|0| |
| *Abbildung 14* | | |

Abbildung 14 zeigt die letzten Züge eines Spielverlaufs des gleichen NN-Players. Der *NNPlayer2* verliert hier aufgrund einer Zwickmühle. Allerdings ist erkennbar, dass das Neuronale Netz die Zwickmühle – wenn auch zu spät - zu erkennen scheint (rot markiert), da er letzten Zug die Diagonale (gelb markiert) blockiert.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| |0|0|0|0|0|  |0|0|0|0|0|  |0|0|0|0|0|  |0|■|0|0|0| | |0|0|0|0|0|  |0|0|0|0|0|  |0|X|0|0|0|  |0|■|0|0|0| | |0|0|0|0|0|  |0|0|0|0|0|  |0|X|0|0|0|  |0|■|■|0|0| | |0|0|0|0|0|  |0|0|0|0|0|  |0|X|0|0|0|  |X|■|■|0|0| | |0|0|0|0|0|  |0|0|0|0|0|  |0|X|0|0|0|  |X|■|■|■|0| |
| *Abbildung 15* | | | | |

In Abbildung 15 wird ein Spiel verlauf gezeigt, bei dem das Neuronale Netz mit 120 Neuronen im Hidden-Layer auf einem Daten-Set vom 2.Spieler mit 300 Einträgen gelernt hat. Auch hier wurde innerhalb weniger Züge eine Zwickmühle gebaut, die zum Sieg führte.

|  |  |  |
| --- | --- | --- |
| |0|0|0|0|0|  |0|X|0|0|0|  |0|X|■|0|0|  |0|■|X|0|0| | |0|■|0|0|0|  |0|X|0|0|0|  |0|X|■|0|0|  |0|■|X|0|0| | |0|■|0|0|0|  |0|X|X|0|0|  |0|X|■|0|0|  |0|■|X|0|0| |

|  |  |  |  |
| --- | --- | --- | --- |
| |0|■|0|0|0|  |0|X|X|0|0|  |0|X|■|0|0|  |■|■|X|0|0| | |0|■|X|0|0|  |0|X|X|0|0|  |0|X|■|0|0|  |■|■|X|0|0| | |0|■|X|0|0|  |0|X|X|0|0|  |■|X|■|0|0|  |■|■|X|0|0| | |0|■|X|0|0|  |X|X|X|0|0|  |■|X|■|0|0|  |■|■|X|0|0| |
| *Abbildung 16* | | | |

Abbildung 16 zeigt die letzten Züge eines Spielverlaufs des gleichen NN-Players, er auch in Abbildung 15 zu sehen ist. Es ist ersichtlich, dass das Neuronale Netz alle Gefahren (rot markiert) erkennt und sie innerhalb des nächsten Zuges verhindert (gelb markiert), aber leider aufgrund einer Zwickmühle des Gegners am Ende verliert.

Abschließend lässt sich also sagen, dass die Neuronalen Netze des *NNPlayer2* in der Lage zu sein scheinen, selbst Zwickmühlen zu bauen, aber auch auf mögliche Gefahren des Gegners zu reagieren, was ein strategisches Vorausdenken zeigt.

**NNPlayer2 6x7**

Da das Spiel auf einem 6x7 Feld und einer Gewinnbedingung von 4 Steinen deutlich komplexer wird, sind nicht die gleichen Ergebnisse wie auf dem 4x5 Feld zu erwarten gewesen. Trotzdem zeigt sich ein Lernerfolg durch Neuronalen Netzes und eine interessante Spielstrategie, die immer wieder zu gewonnen Spielen für den NN-Player führten.

Um zu bestätigen, dass die KI trotz geringerer Gewinnquote eine deutliche Verbesserung aufweist, wird die Gewinnrate von einer zufälligen KI betrachtet: Diese gewinnt bei 10.000 Turnierspielen gerade einmal 138 also ca. 1 % der Spiele. Das Neuronale Netz ist wieder ein Multi-Layer Perzeptron und es wurden unterschiedliche Daten-Sets sowie unterschiedliche Parameter getestet. Die Anzahl der Input-Neuronen ergab 126, die Output-Neuronen 7 und die Hidden-Neuronen wurden während der Tests bei 252 belassen.

Es wurde mit unterschiedlich große Daten-Sets, welche mit der *generateDataSets*-Methode erstellt wurden, getestet:

1. 50 Trainingsspiele mit ca. 700 Einträgen
2. 100 Trainingsspiele mit ca. 1250 Einträgen
3. 150 Trainingsspiele mit ca. 2000 Einträgen
4. 500 Trainingsspiele mit ca. 6000 Einträgen

Dabei wurden für den maximalen Fehler Werte von 0,3 bis 0,01 verwendet. Eine Reduzierung auf einen Fehler auf unter 0,1 war allerdings nur bei einem kleineren Daten-Set möglich, da die Fehlerrate bei den größeren Sets davor bereits konvergiert hat. Auch Veränderungen an anderen Parameter, wie der Lernrate oder dem Momentum, konnten Erfolge verzeichnen. Bei 50 und 100 Trainingsspielen blieb die Gewinnquote bei ca. 10 % mit nur leichten Schwankungen bei unterschiedlichen Fehlerraten. Mit 150 Trainingsspielen, also 2000 Einträgen, konnte das beste Ergebnis erzielt werden. Mit mehr Einträgen wurde das Ergebnis wieder schlechter. Das Ergebnis mit 100 Trainingsspielen lässt darauf schließen, dass ein zu geringer erlaubter Fehler zu einem Overfitting führt und der NN-Player daher schlecht spielt. Dies bestätigt das Ergebnis mit 150 Trainingsspielen. Bei einem Fehler von 0,3 gewann der Spieler ca. 25 % der Spiele und 10 % gingen unentschieden aus. Im Vergleich zu den anderen Trainingsversuchen und dem Spielen gegen eine zufällig spielende KI ist das eine deutliche Mehrleistung.

Im Folgenden soll die Spielweise des NN-Players mit dem eben beschriebenen, besten neuronalen Netz betrachtet werden (Q-Player = ■ NormalKI=X)

|  |  |  |
| --- | --- | --- |
| Spiel 1 | Spiel 2 | Spiel 3 |

|  |  |  |
| --- | --- | --- |
| |0|0|■|X|X|0|0|  |0|0|X|■|X|0|0|  |0|0|■|■|■|0|0|  |0|0|■|■|X|0|0|  |0|X|■|X|■|■|0|  |■|X|X|■|X|X|0|  |0|0|■|X|X|0|0|  |0|0|X|■|X|0|0|  |0|0|■|■|■|0|0|  |0|0|■|■|X|X|0|  |0|X|■|X|■|■|0|  |■|X|X|■|X|X|0|  |0|0|■|X|X|0|0|  |0|0|X|■|X|0|0|  |0|0|■|■|■|■|0|  |0|0|■|■|X|X|0|  |0|X|■|X|■|■|0|  |■|X|X|■|X|X|0| | |0|0|X|0|■|0|0|  |0|0|■|X|X|0|0|  |0|0|X|■|■|0|0|  |0|0|■|■|■|0|0|  |0|0|■|X|X|0|0|  |0|X|X|■|X|■|0|  |0|0|X|0|■|0|0|  |0|0|■|X|X|0|0|  |0|0|X|■|■|0|0|  |0|0|■|■|■|0|0|  |0|0|■|X|X|X|0|  |0|X|X|■|X|■|0|  |0|0|X|0|■|0|0|  |0|0|■|X|X|0|0|  |0|0|X|■|■|0|0|  |0|0|■|■|■|■|0|  |0|0|■|X|X|X|0|  |0|X|X|■|X|■|0| | |0|0|0|0|0|0|0|  |0|0|X|X|0|0|0|  |0|0|■|■|0|0|0|  |0|0|■|X|■|0|0|  |0|■|■|■|X|0|0|  |0|X|X|■|X|0|0|  |0|0|0|0|0|0|0|  |0|0|X|X|0|0|0|  |0|0|■|■|0|0|0|  |0|0|■|X|■|0|0|  |0|■|■|■|X|0|0|  |X|X|X|■|X|0|0|  |0|0|0|0|0|0|0|  |0|0|X|X|0|0|0|  |0|0|■|■|0|0|0|  |0|0|■|X|■|0|0|  |■|■|■|■|X|0|0|  |X|X|X|■|X|0|0| |

*Abbildung 17*

Diese drei Spielen zeigen, dass das Neuronale Netz gelernt hat, Spielsituation zu erschaffen, in denen der Gegner in eine Zwickmühle gerät. Diese Situationen treten in nahezu allen Spielen auf, die der NN-Player gewinnt.

|  |  |  |
| --- | --- | --- |
| Spiel 1 | Spiel 2 | Spiel 3 |
| |X|■|■|X|X|0|X|  |■|■|X|■|X|0|■|  |X|X|■|■|X|X|X|  |■|■|■|X|■|■|■|  |X|X|■|■|X|X|X|  |■|X|X|■|X|■|■|  |X|■|■|X|X|0|X|  |■|■|X|■|X|■|■|  |X|X|■|■|X|X|X|  |■|■|■|X|■|■|■|  |X|X|■|■|X|X|X|  |■|X|X|■|X|■|■|  |X|■|■|X|X|X|X|  |■|■|X|■|X|■|■|  |X|X|■|■|X|X|X|  |■|■|■|X|■|■|■|  |X|X|■|■|X|X|X|  |■|X|X|■|X|■|■| | |0|0|■|X|0|0|0|  |0|0|X|■|0|0|0|  |0|0|■|X|0|0|0|  |0|0|■|■|X|0|0|  |0|0|■|X|■|0|0|  |0|0|X|■|X|X|0|  |0|0|■|X|0|0|0|  |0|0|X|■|0|0|0|  |0|0|■|X|■|0|0|  |0|0|■|■|X|0|0|  |0|0|■|X|■|0|0|  |0|0|X|■|X|X|0|  |0|0|■|X|0|0|0|  |0|0|X|■|0|0|0|  |0|0|■|X|■|0|0|  |0|0|■|■|X|0|0|  |0|0|■|X|■|X|0|  |0|0|X|■|X|X|0| | |0|0|X|X|■|0|0|  |0|0|■|■|■|0|0|  |0|0|X|X|X|0|0|  |0|0|■|■|■|0|0|  |0|0|■|X|X|0|X|  |0|X|■|■|X|X|■|  |0|0|X|X|■|0|0|  |0|0|■|■|■|0|0|  |0|0|X|X|X|0|0|  |0|0|■|■|■|0|■|  |0|0|■|X|X|0|X|  |0|X|■|■|X|X|■|  |0|0|X|X|■|0|0|  |0|0|■|■|■|0|0|  |0|0|X|X|X|0|0|  |0|0|■|■|■|0|■|  |0|0|■|X|X|X|X|  |0|X|■|■|X|X|■| |

*Abbildung 18*

Die nächsten 3 Spiele stellen häufig wiederzufindende Spielsituationen aus den Turnierspielen dar. Wenn die *NormalKI* gewinnt, dann immer über eine einfache Kombination, die der NN-Player übersieht. Zwickmühlen werden von der *NormalKI* nicht erzeugt. Weiterhin lässt sich beobachten, dass die Spiele häufig sehr lange dauern. Die durchschnittliche Anzahl an Zügen pro Spiel liegt bei ca. 29. D.h. Spielzustände wie der aus Spiel 1 oder ähnliche, die unentschieden ausgehen, sind häufig in den Spielen zu finden.

Insgesamt kann man das Potenzial der Neuronalen Netze auch bei einem Spielfeld der Größe 6x7 erkennen. Es hätte mehr Zeit gebraucht, um noch mehr Einstellungen zu testen. Insbesondere bei den Daten-Sets würde es durchaus Verbesserungsmöglichkeiten geben. Da die Daten-Sets automatisch generiert werden, schwankt die Gewinnquote des NN-Players erheblich. Mit einem manuell erstellen Daten-Set oder einer anderen Art Daten-Sets zu erstellen, könnten weitere Verbesserungen möglich sein. Eine Möglichkeit wäre, bei der Erstellung darauf zu achten, wie gut die einzelnen Einträge in dem Daten-Set sind.

## Ausblick

Die Ergebnisse dieses Business Analytics Projekts zeigen, dass es mit der Methode des Q-Learnings und auch mit der Methode von Neuronalen Netzen möglich ist, eine selbst lernende Künstliche Intelligenz für das Spiel „Vier-Gewinnt“ zu implementieren, doch gibt es noch weitere Möglichkeiten dieses Projekt in Zukunft fortzuführen bzw. zu erweitern, diese werden im Folgenden dargestellt.

**Q-Player**

Der Q-Player erzielt auf einem 4x5 Feld hervorragende Ergebnisse. Die einzige Frage, die bleibt ist, warum er ab einem gewissen Grad wieder schlechter wird. Hierzu sollte die Berechnung der Werte für die Zustände neu betrachtet werden. Evtl. genüg es schon auf die Durchschnittsberechnung zu verzichten und nur den maximalen Wert der nächsten Zustände zu betrachten, nachdem für den Gegner vom für ihn bestmöglichen Zug ausgegangen wird. Ein anderer Ansatz ist es nicht mehr davon auszugehen, dass der Gegner den bestmöglichen Zug macht, sondern an dieser Stelle einen Mittelwert bildet. Beide Alternativen müssten experimentell erforscht werden.

Auch auf dem 6x7 Spielfeld kann der Q-Player von einer Verbesserung an dieser Stelle profitieren. Zunächst muss jedoch sichergestellt werden, dass er überhaupt alle für ihn wichtigen Spielzustände erforschen kann. Da die gewählte Implementierung der Datenbank hier an ihre Grenzen stößt, liegt hier Verbesserungspotenzial vor, um den Q-Player auch auf einem 6x7 Feld zu trainieren. Die Datenbank wird sehr groß und ab einer gewissen Größe reicht der Arbeitsspeicher nicht mehr aus. Dann wird die Datenbank zu Teilen auf die Festplatte geschrieben und das Abfragen von Spielzuständen dauert sehr lange. Ein Ansatz zur Komprimierung der Datenbank scheint lohnenswert. Der Schlüssel der HashMap, die intern verwendet wird, ist ein 2-dimensionales Int-Array. Dies ist vom Speicherverbrauch nicht sehr effizient, da die Anzahl der Zustände in der Datenbank sehr schnell sehr groß wird. Bei 3.200.00 Trainings gibt es 21.055.352 Einträge in der Datenbank. Einen kleineren Schlüssel zu verwenden könnte den Speicherverbrauch reduzieren.

Allerdings kann es sein, dass auch so nicht alle Zustände in einer händelbaren Datenbank untergebracht werden können. Wie oben erwähnt beschäftigt sich Olszowka in seiner Arbeit mit der Problematik rund um die Anzahl der Zustände bei einem 6x7 Spielfeld. Er geht sogar davon aus, dass es nicht einmal möglich ist, die KI alle Zustände erforschen zu lassen. Diese Aussage basiert jedoch auf der Annahme, dass es etwa Zustände gibt. [vgl. ] Tatsächlich sollte die Anzahl bedeutend kleiner sein, da die Steine nicht beliebig auf dem Feld verteilt werden dürfen. Zu beachten sind u. a. folgende Punkte:

* Die Spieler dürfen nur abwechselnd werfen, d. h. die Anzahl der Steine von Spieler 2 und Spieler 2 unterscheiden sich maximal um eins.
* Steine könne nicht „schweben“. Sie fallen immer so weit nach unten, wie es ihnen möglich ist.
* Wenn vier Steine einer Farbe neben einander liegen, ist das Spiel beendet, d.h. es werden keine weiteren Steine mehr platziert.

Trotzdem wird es sich lohnt, sich mit Optionen zur Zustandsreduzierung zu befassen. Olszowka schlägt die Verwendung Feature-Vektoren vor. Dabei wird nicht mehr jeder Spielzustand einzeln gespeichert. Stattdessen erhält der der Q-Player einen Vektor mit Eigenschaften, die im jeweiligen Zustand enthalten sind. Wichtig ist aber, dass nicht zu viele Informationen verloren gehen. [vgl. ]

**NNPlayer**

Den *NNPlayer2* betreffend, wäre es von Bedeutung, sich die Daten-Sets genauer anzuschauen. Zurzeit werden sie durch zufällige Spielzüge der *NormalKI* generiert und man hat bisher keinen Einfluss darauf, bestimmte Spielzüge auszusortieren, d. h. es ist möglich, dass auch Spielzüge gespeichert werden, die zum Verlieren führen. Daher wäre es sinnvoll, nur die Spielzüge zu speichern, die zum Sieg führen.

Außerdem kann die *turn*-Methode im *NNPlayer2* umgestaltet werden, damit hier mit zwei Neuronalen Netzen gearbeitet werden kann; eines für die Spiele des 1. Spielers und eines für Spiele des 2. Spielers. Je nach Situation im Turnier, sollte dann das passende Netz ausgewählt werden, um so die Gewinnchancen zu erhöhen. Dies müsste allerdings auch experimentell getestet werden.

1. MacKay, David, J.C. (2003). Information Theory, Inference, and Learning Algorithms. Cambridge University Press. ISBN 9780521642989.
2. Neural Networks FAQ. (am 5. September 2015) (<ftp://ftp.sas.com/pub/neural/FAQ2.html#A_act>).
3. Werner Kinnebrock: Neuronale Netze: Grundlagen, Anwendungen, Beispiele. R. Oldenbourg Verlag, München 1994, ISBN 3-486-22947-8