Reinforcement Learning zum maschinellen Erlernen von Brettspielen am Beispiel des Strategiespiels „4-Gewinnt“

Diplomarbeit

vorgelegt an der Fachhochschule Köln
Campus Gummersbach

im Studiengang Allgemeine Informatik

ausgearbeitet von:
Jan Philipp Schwenck

Erster Prüfer: Prof. Dr. Wolfgang Konen
Zweiter Prüfer: Prof. Dr. Hartmut Westenberger

Gummersbach, im Oktober 2008

Inhaltsverzeichnis

1 Einleitung und Aufgabenstellung....4

1.1 Motivation....4

1.2 Zielsetzung....4

1.3 Gliederung....5

1.4 Praxisrelevanz....5

1.5 Stand der Technik und aktuelle Vorarbeiten....6

2 Reinforcement Learning....8

2.1 Die Grundidee....8

2.2 Begriffe und Funktionsweise....9

2.2.1 Policies....10

2.2.2 Situationsbewertende Valuefunktion V....11

2.2.3 Aktionsbewertende Q-Funktion....11

2.3 Ziel der Reinforcement Learning Strategie....12

2.4 Pseudocode des RL-Algorithmus....13

3 Erlernen des Spiels „Nimm-3“ mittels RL....14

3.1 Grundlagen zum Spiel „Nimm-3“....14

3.2 Pseudocode des Self-Play-TD(λ)-Algorithmus 16

3.3 Tabellarische Spielfunktion....17

3.3.1 Aufbau der tabellarischen Struktur....17

3.3.2 Nicht bewertbare Zustände....18

3.3.3 Generierung der tabellarischen Spielfunktion....19

3.4 Lineare Spielfunktion über lineares Netz....21

3.4.1 Arten alternativer Spielfunktionen....21

3.4.2 Erzeugung eines linearen Netzes....23

3.4.3 Anwendung des linearen Netzes....25

3.5 Spielfunktion über neuronales Netz....31

3.5.1 Unterschiede zwischen linearem und neuronalem Netz....31

3.5.2 Anwendung des neuronalen Netzes....33

3.6 Analyse und Ergebnisse zur Lernfähigkeit von Nimm-3....36

4 Erlernen des Spiels „4-Gewinnt“ mittels RL....40

4.1 Grundlagen zum Spiel „4-Gewinnt“....40

4.2 Softwareseitige Entwicklung eines Computergegners....42
4.2.1 Grundlagen zur Klassenaufteilung der Spielumgebung....42
4.2.2 RL_NNAgent – Erzeugung der Spielfunktion über neuronales Netz....43
4.3 Konzeptionelle Gestaltungsvarianten des Input-Vektors....48
4.3.1 Codierung über Spielfeldbelegung (3 Neuronen pro Spielfeld)....49
4.3.2 Ergebnisse der ersten Codierungsvariante....50
4.3.3 Codierung über verkürzte Spielfeldbelegung (-1/1/0-Strategie)....52
4.3.4 Ergebnisse der zweiten Coderierungsvariante....53
4.3.5 Codierung über Muster (konzeptioneller Vorschlag)....55
4.3.6 Gegenüberstellung der Alternativen....58

5 Fazit und Ausblick....60
5.1 Fazit zu Arbeitsergebnissen....60
5.2 Ausblick und Erweiterungsmöglichkeiten....61
5.2.1 Alternativen der Spielcodierung, Entwicklung neuer Features....61
5.2.2 Entwicklung aussagekräftiger Spielsituationen zu Testzwecken....61
5.2.3 Methodenentwicklung zur Lernen aus Spielbegegnungen....62
5.2.4 Alternative Testverfahren zur Evaluation der Theorie....62
5.3 Erfahrungswerte....63
5.3.1 „Lessons Learned“ und Fehlerkorrekturen....63
5.3.2 Persönliches Fazit....64

A Anhang....65
A.1 Entwicklung der tabellarischen Spielfunktion für „Nimm-3“....65
A.2 Kantenanpassungen des linearen Netz bei Nimm-3....70
A.3 Auswertungen der Spielqualität des „4-Gewinnt“-Agenten....71
A.3.A Ergebnisse Codierung 1 bei Endspielvariante 1....71
A.3.B Ergebnisse Codierung 1 bei Endspielvariante 1b....71
A.3.C Ergebnisse Codierung 2 bei Endspielvariante 1b....72
A.3.D Ergebnisübersicht aller durchgeführten Tests....73

B Appendix....74
B.1 Abbildungsverzeichnis....74
B.2 Literatur- und Quellenverzeichnis....75
B.3 Erklärung....76

1 Einleitung und Aufgabenstellung

1.1 Motivation

Seitdem Computer existieren, stellt sich der Mensch die Frage, ob es eines Tages dazu kommen wird, dass künstliche Intelligenz der unseren überlegen ist. Kommt der Tag an dem wir mit unserem begrenzten Wissen einer höheren Intelligenz unterstehen? Werden Maschinen in der Lage sein, die komplexen Anforderungen eines menschlichen Gehirns zu simulieren und nachbilden zu können?

Auch wenn diese Überlegungen wohl aus dem fernen Hollywood oder den Gedanken Science-Fiction produzierender Filmemacher kommen, ist die Grundidee, maschinelle beziehungsweise künstliche Intelligenz erzeugen zu können, gar nicht so abwegig, wie sie bei oberflächlicher Betrachtung zu sein scheint. Bereits heute ist es möglich, maschinelle Intelligenz zu erzeugen und zu fördern, wenn Rechnern genügend Vorwissen in Form von Daten zur Verfügung steht.

Maschinelle Lernverfahren sind daher ein spannendes Forschungsfeld innerhalb der Informatik, weil sie die Möglichkeit implizieren, dass Computer als künstliche Wesen tatsächlich in der Lage sind, fremde Sachverhalte zu erlernen. Die Forschung befindet sich heute zunächst am Anfang einer langwierigen Entwicklung, die sich aber mit genau dieser Fragestellung nach der Reichweite künstlicher Intelligenz auseinandersetzt.

1.2 Zielsetzung

Diese Diplomarbeit setzt sich mit dem Thema des Erlernens strategischer Brettspiele auseinander. Die Frage ist, ob ein Rechner in der Lage ist, komplexe Spielsituationen so zu bewerten, dass er letztendlich das erfolgreiche Spielen von Brettspielen erlernt. Hierzu versucht man, dem künstlichen Spielgegner, der durch den Rechner repräsentiert wird, die Spielfunktion eines Brettspiels zu trainieren. Da diese Spielfunktion aber häufig viel zu komplex ist, muss der Rechner eine möglichst genaue Approximation dieser Spielfunktion erzeugen, die ihm genügend qualitative Aussagen über eine möglichst erfolgreiche Spielweise eines Brettspiels bietet und damit einer Erfolg versprechenden Spielfunktion nahe kommt.

Ziel dieser Arbeit soll es sein, am Beispiel des strategischen Brettspiels „4-Gewinnt“ einen Computergegner zu erschaffen, der dieses Brettspiel erfolgreich spielen kann und sich gegen den durchschnittlichen menschlichen Spielgegner zu behaupten weiß. Der Erfolg des Erzeugens eines solchen Computergegners ist zum Beginn des Ausarbeitungszeitraums der Diplomarbeit noch nicht garantiert, wird aber durch existierende Quellen verschiedener

wissenschaftlicher Ausarbeitungen als grundsätzlich möglich bestätigt, daher ist die praktische Umsetzung einer solchen Realisierung begründet und die Machbarkeit – zumindest theoretisch – gefestigt.

1.3 Gliederung

Die vorliegende Diplomarbeit ist in folgende Struktur untergegliedert: Neben einem ausführlichen Theorieteil, in dem Grundlagen zur theoretischen Betrachtung der angewandten, wissenschaftlichen Methode genannt und erläutert werden, wird im zweiten Teil dieser Arbeit die Erzeugung eines künstlichen Computerspielers anhand eines einfacheren Brettspiels mit Namen „Nimm-3“ beschrieben. Um die Realisierbarkeit der angewandten Methodik am leichteren Beispiel vorab zu testen, ist diese Einteilung sinnvoll. Im abschließenden dritten Teil wird schließlich die Entwicklung und Analyse eines Computerspielers für das Brettspiels „4-Gewinnt“ dargestellt. Evaluation, Analysen und Ergebnisse werden an entsprechenden Stellen betrachtet und erläutert. Im dritten Teil wird eine zusätzliche Unterteilung in zwei größere Abschnitte vorgenommen. Im ersten Abschnitt dieser Unterteilung wird die softwareseitige Entwicklung eines Computergegners dargestellt, der zweite Abschnitt beinhaltet die Auseinandersetzung mit einigen konzeptionellen Gestaltungsvarianten, um diesen Computergegner in seiner Leistung effektiver zu stärken. Die exakte Gliederung dieser Arbeit ist im Inhaltsverzeichnis bis auf eine Abschnittstiefe von drei dargestellt.

1.4 Praxisrelevanz

Intelligente Systeme und lernende Agenten werden inzwischen verstärkt in der Praxis eingesetzt. In der Automobilbranche werden Robotersteuerungen entwickelt, deren Bewegungen zuvor erlernt und trainiert worden sind; das gesamte Feld der Robotik nutzt solche Lernmethoden wie Reinforcement Learning. Lernende Strukturen sind im Laufe der technischen Entwicklungen auch im Spielekontext immer wichtiger geworden, sei es im Bereich strategischer Rollen- oder Kriegsspiele oder auch bei Brettspielen. Reinforcement Learning (im Folgenden mit „RL“ abgekürzt) ist eine Möglichkeit das Lernen von Systemen zu unterstützen. Es ist eine, in der Praxis anerkannte und genutzte Methode, da sie sich besonders dann verwenden lässt, wenn sich Belohnungen oder Bestrafungen – die das Lernverfahren beeinflussen – nicht direkt nach einer Aktion, sondern erst nach einer Reihe von Aktionen feststellen lassen. Lernende Strukturen oder Agenten haben im Vergleich zu formulierten Algorithmen den Vorteil, sich dynamisch auf Änderungen in der Umwelt anzupassen und darauf zu reagieren. Da sich lernende Systeme idealerweise an der menschlichen Lernfähigkeit orientieren sollen, ist deren Entwicklung für unterstützende Arbeitsvorgänge sehr geeignet.

1.5 Stand der Technik und aktuelle Vorarbeiten

Diese Arbeit ist natürlich kein Novum in diesem Bereich. Es existieren schon seit vielen Jahren eine Reihe an Arbeiten, die sich mit RL und dem Nutzen der Strategie in Anwendung auf verschiedene Bereiche auseinandersetzen. Sutton und Bonde sind dabei diejenigen, die 1992 als erste unabhängig voneinander die RL-Strategie als pseudocodeartigen Algorithmus aufgeschrieben haben. Sutton und Barto veröffentlichten daraufhin 1998 ihr Werk „Reinforcement Learning: An Introduction“, in dem sie die Methodik des RL detailliert und fachgerecht erklären und jede Bezeichnung sowie verschiedene Lerntechniken erläutern. Dieses Werk dient bis heute als Grundlage für jeden, der RL als Lernstrategie zur Anwendung bringen will. Dennoch waren sie nicht die Ersten, die sich mit dem Thema RL auseinandersetzten. Vor ihnen schrieben unter anderem schon Kaelbling, Littman und Moore an einer „Reise durch die RL-Technik“.

Wie bereits erwähnt, produzierten Sutton und Bonde aber auch einen Pseudocode, der die komplette RL-Strategie algorithmisch beschreibt. Bis auf Eingabe- und Ausgabe sowie einer Methode zur Zufallswahl ist dieser Pseudocode vollständig und unterstützt Entwickler daher bei der Implementierung der RL Methoden. Beide Werke zusammen beschreiben die RL-Strategie zur Genüge und dienen als gute, grundlegende Literatur zur Einarbeitung in RL.

Die Idee, RL beim maschinellen Erlernen strategischer Brettspiele zu nutzen, wurde zunächst und bisher auch am erfolgreichsten von Gerald Tesauro genutzt. Er schuf ein Backgammon-Spiel, bei dem der Computergegner mittels RL eine perfekte Spielstrategie erlernt und danach auch anwenden kann. Diese Entwicklung diente als Ursprung und Grundlage vieler Weiterentwicklungen und Implementierungen, die heute teilweise als freie, teilweise auch als kommerzielle Spiele-Software auf dem Markt sind. Wer hierzu mehr erfahren möchte, sei auf das Werk Tesauros „Mach. Learning 8“[1] von 1992 verwiesen.

Stenmark schreibt seine Master-Thesis über die Erlernbarkeit des Spiels „4-Gewinnt“ mittels RL, er nutzt dabei allerdings den Vergleich zu einer zweiten Methode ADATE[2], die sein Mentor und Professor entwickelte, um die Lernergebnisse des RLAgenten besser evaluieren zu können und eine Aussage darüber zu tätigen, welche Technik Erfolg versprechender ist. ADATE bezeichnet dabei eine Methode der dynamischen Programmierung; die RL-Strategie kann über dynamische Programmierung gelöst werden, günstiger ist allerdings der „temporal difference“-Ansatz. Was Stenmarks Arbeit ebenso von dieser Arbeit abgrenzt, ist die Tatsache, dass er sich mit der Betrachtung der unterschiedlichen Umsetzungsmöglichkeiten des RL nahe am Buch von

1 Tesauro (1992)
2 Vgl. Stenmark (2005), ab Seite 26

Sutton und Barto orientiert. Er stellt das RL-Konzept mittels dynamischer Programmierung, der sogenannten MonteCarlo-Methode und der „temporal difference“-Lösung vor.

Diese Arbeit widmet sich im Gegensatz dazu im Ganzen der Erklärung und Detailerläuterung der RL-Strategie. Neben einem ausführlichen Theorieteil kommt RL am einfacheren Strategiebrettspiel Nimm-3 zur Anwendung, um erste Lernergebnisse zu erläutern und die Benutzung und Funktion der Technik zu verdeutlichen.

Später, beim Strategiespiel „4-Gewinnt“, wird zusätzlich über verschiedene Möglichkeiten der Eingabecodierung für das neuronale Netz, welches die Spielfunktion erzeugen soll, nachgedacht. Dies ist ebenfalls eine Abgrenzung zu Stenmark, der sich in seiner Arbeit nicht mehr mit Optimierungsmöglichkeiten des RLAgenten beschäftigt. Diese Überlegungen zur Optimierbarkeit der sogenannten Eingabefeatures wird ein weiterer Abgrenzungspunkt zu Stenmark sein. Features sind die Parameter, über die eine Spielfunktion möglichst gut repräsentiert werden kann. Es kann passieren, dass RL als Technik alleine nicht ausreicht, um eine genügend starke Spielfunktion zu erzeugen. Aufgrund eines großen Zustandsraum, der beim Spiel „4-Gewinnt“ vorliegt (siehe Kapitel 4.3.1), kann es notwendig werden, dass Eingabeparameter angepasst, neu überdacht oder sogar komplett andere erzeugt werden müssen. So beschäftigt sich Kapitel 5.3 mit Überlegungen zu Codierungs- und Gestaltungsalternativen.

2 Reinforcement Learning

2.1 Die Grundidee

Grundsätzlich benötigen Systeme, die etwas erlernen sollen, Beispieldaten. Wenn das Umfeld, die Regeln, Arbeitsabläufe oder aufeinanderfolgende Schritte nicht bekannt sind, kann ein System nur dann erfolgreich etwas erlernen, wenn es durch diese Beispieldaten etwaige Muster, Strukturen oder generelle Folgen entdeckt.

Beispieldaten können unterschiedlich vorliegen: Sie können von externen Trainern oder Experten als Input/Output-Kombination bereitgestellt werden, das heißt, es ist bekannt, was als Zustand oder Aktion einem System eingegeben werden kann, und was als Zustand oder Aktion nach Veränderung desselben herauskommt. Die zweite Variante beschreibt das Szenario, dass nur Daten vorliegen, die als Input dem System eingegeben werden können. Beispielsweise existieren einige aufeinanderfolgende Bewegungsabläufe eines Roboters, die aber nicht durch entsprechenden Output bewertet werden können hier müsste das System selbstständig Strukturen erkennen, ohne zu wissen, ob bestimmte Bewegungsabläufe hilfreich sind oder nicht. Die dritte Art des Vorkommens von Lerndaten wird durch Bespieldaten repräsentiert, die als Folge mehrerer Einzelschritte vorliegen und letztlich erst abschließend bewertet werden können. Dies entspricht im Wesentlichen der „Zuckerbrot und Peitsche“-Strategie, auch bekannt als „Belohnungs-/Bestrafungsverfahren“, wo erst nach einer Reihe von Aktionen eine Belohnung oder Bestrafung ausgesprochen werden kann.

RL beschreibt die Technik des bestärkenden Lernens, welche durch die zuletzt beschriebene Art des Datenvorkommens realisiert wird; neben ihr existieren zwei weitere Möglichkeiten, Systeme bestimmte Zusammenhänge oder Aktionen lernen zu lassen. Das überwachte Lernen setzt voraus, dass ein Experte Lernbeispiele an das System weitergibt, anhand deren dieses dann Zusammenhänge erlernt. Beim unüberwachten Lernen wird bereits in der Lernphase das System sich selbst überlassen, es lernt schließlich nur aus eigenen, generierten Lernbeispielen ohne zu wissen, ob diese Erfolg versprechend sind oder nicht.

Bestärkendes Lernen geht nun davon aus, dass nicht bei jeder durchgeführten Aktion der Erfolg bekannt ist. So kann man oft erst nach einer gewissen Zahl von Aktionen feststellen, ob ein Erfolg oder ein Misserfolg vorliegt. Zur Anwendung der RL-Technik sind also Datenreihen notwendig, die letztendlich mit einem Belohnungswert (englisch: Reward) abgeschlossen werden – dieser Wert kann natürlich auch negativ sein, was somit einer Bestrafung gleicht.

2.2 Begriffe und Funktionsweise

„(..) the basic idea is simply to capture the most important aspects of the real problem facing a learning agent interacting with its environment to achieve a goal".[3]

RL setzt also einen Lernagenten voraus, der mit seinem Umfeld interagiert und ein bestimmtes Ziel erreichen will. Dabei soll dieser Agent stets unter Berücksichtigung des Umfeldes, mit Blick auf die wichtigsten Aspekte des ursprünglichen Problems handeln. Ein Agent bezeichnet dabei ein agierendes System, was meistens durch ein entsprechendes Computerprogramm realisiert wird.

Wichtig ist das Wissen über die Existenz von Zuständen und Aktionen. Jedes System hat das Ziel, mit bestimmten Aktionen zu einem bestimmten Zustand zu gelangen. Sei es die Robotik, die versucht Robotersteuerungen zu realisieren, sei es die Lernfähigkeit bestimmter Zusammenhänge. Immer geht es um die Analyse von Aktionen zum Erreichen bestimmter Zustände. Es existiert auch der Fall, dass Systeme über bestimmte Zustände erlernen sollen, welche Aktionen positiv beziehungsweise negativ sind. Häufig analysieren Systeme aber vorhandene Zustände, um mit eigenen Aktionen – die meistens begrenzt sind – zu einem Zielzustand zu gelangen.

Die Interaktion mit dem Umfeld ist für einen Agenten daher so wichtig, weil dieser selbst die Bewertung zu seiner Aktion nicht kennt, das bedeutet, dass nach einer Ausführung einer Aktion, das Umfeld diese Aktion bewertet und an den Lernagenten zurückgibt. Dieser lässt eine eventuell vorhandene Bewertung in die Gewichtung der durchgeführten Aktion einfließen. So lernt der Agent mit der Zeit seine Aktionen so zu bewerten, dass er möglichst zu einem optimalen Weg zu seinem Ziel gelangt.

Das folgende Beispiel (Abbildung 1) soll die Funktionsweise der RL-Technik genauer erläutern:[4]

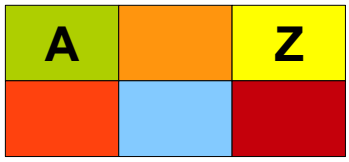


Abbildung 1: Gitternetz für Roboter

Es existiert ein Roboter, der das Ziel hat, in ein bestimmtes Feld Z innerhalb eines Gitternetzes zu gelangen. Dabei kann er pro Schritt nur ein Feld voranschreiten, die Richtung ist nicht begrenzt, er kann sich jedoch nur innerhalb des abgegrenzten Bereichs bewegen und auch nur waagerechte oder senkrechte Bewegungen vollziehen, diagonale Schritte sind nicht erlaubt. Das Startfeld des Roboters ist A.

Der Roboter hat nur eine begrenzte Auswahl an Zugmöglichkeiten: Er kann im ersten Zug nach rechts in das orange-farbige Feld ziehen oder nach unten in das Rote. Die Zustände, die er so nach dem Zug erreichen kann, sind „Roboter ist in Feld orange" oder „Roboter ist in Feld rot."

3 Sutton/Barto (1998), Kapitel 1.1
4 Beispiel entnommen aus Markelic (2004), Seite 12 ff

Egal, welches der beiden Felder er erreicht, er erhält noch keinen Belohnungswert, da dieser erst in Feld Z – das Ziel wird erst hier erreicht – vergeben werden kann.

Der Roboter soll durch seine Zugreihenfolge nun erlernen, wie er das rechte obere Feld erreichen kann. Bisher verfolgt er keine Strategie, da er jede Aktionswahl nur zufällig entscheidet. Es geht also um die Generierung einer Zugfunktion, die angibt, wie wertvoll ein Zug für das Erreichen des Endzustands ist. Diese Zugfunktion wird mit Hilfe der Abschlussbelohnung in Feld Z im Laufe mehrerer Lerndurchgänge erstellt. Denn erst bei Übergang von Feld orange oder Feld rot in das Zielfeld kann eine Bewertung erteilt werden, da erst bei diesen Zügen eine Abschlussbelohnung erreicht wird. Hier würde dem roten beziehungsweise dem orange-farbigen Feld eine Bewertung gegeben, die sich unter anderem aus der Abschlussbewertung des Zielfeldes ergibt.

An diesem Beispiel soll daher verdeutlicht werden, dass es immer um die Analyse des aktuellen Zustands und der möglichen weiteren Zustände geht, die durch Aktionen erreicht werden können. Im zweiten Durchlauf des Roboters könnten die Felder, die nicht direkt am Zielfeld anliegen bewertet werden, indem sie die neuen Bewertungen der Felder orange und rot berücksichtigen.

Die genaue Realisierung der Bewertungen und damit die Formulierung der Zugfunktion hängt aber letztlich neben dem Reward von weiteren Faktoren und Variablen ab, die im Laufe der nachfolgenden Kapiteln erläutert werden.

2.2.1 Policies

Als Policy bezeichnet man beim RL das Regelwerk, nach dem der Agent agieren kann. Nicht immer sind alle möglichen Aktionen auch gestattet, in manchen Fällen werden die ausführbaren Schritte auch begrenzt, um eine Richtung zum Ziel vorzugeben. Als Beispiel sei hier das Schachspiel genannt. Möchte man mit einer Figur ziehen, darf man nur bestimmte, erlaubte Bewegungen vollziehen. Die Sammlung beispielsweise aller erlaubten Zugmöglichkeiten würde im RL-Kontext mit Policy bezeichnet.

Es kann sinnvoll sein – vergleichbar dem Beispiel in Kapitel 2.2 – einem Roboter bei der Zugwahl nur Vorwärtszüge hin zum Ziel zu erlauben, damit er keine Kreise beziehungsweise Schleifen dreht, die unnötig wären und nie zum Ziel führen. Somit würde eine stärker eingeschränkte Policy vorliegen als die zuvor durch die Sammlung aller erlaubten Zugmöglichkeiten beschriebene Policy.

Je nach Policy werden Zustandsbewertungen auch genauer oder eindeutiger, da die Policy bei der Bewertung der Zustände Berücksichtigung findet. Beispielsweise würde eine weitere

Einschränkung auf waagerechte Züge im genannten Beispiel zur Folge haben, dass die untere Hälfte des Gitternetzes gar nicht besucht und daher auch nicht bewertet werden würde. Somit spielt schon die Wahl der richtigen Policy bei der Nutzung der RL-Technik eine entscheidende Rolle. Weitere ausführliche Informationen findet man bei Stenmark[5].

2.2.2 Situationsbewertende Valuefunktion V

Die Value-Funktion beschreibt mathematisch die Qualität eines Zustands beziehungsweise einer vorliegenden Situation. Jeder Zustand wird bewertet, indem die zur Verfügung stehende Policy beachtet und verfolgt wird. Der Wert eines Zustands bezeichnet letztlich seine Güte zum Ziel zu führen.[6]

Die Value-Funktion lässt sich wie folgt definieren: $V^{\pi}(s_t) = r_t + y \cdot r_{t+1} + y^2 \cdot r_{t+2} + \ldots$ [7]

„s_t" bezeichnet den Zustand zum Zeitpunkt t, „r" steht für den Reward zum Zeitpunkt t. Bei Spielen ist beispielsweise erst nach dem Gewinnen oder Verlieren eines Spiels der Reward ungleich null, da er erst dann angegeben werden kann. „y" steht für den sogenannten „Discount Factor", der einen Wert kleiner 1 belegt und den Folgezustand als einen von mehreren möglichen Folgezuständen „abzinst". Es besteht eben auch die Möglichkeit in einen anderen Folgezustand und damit einen anderen Reward r_{t+x} zu gelangen, daher werden weiter in der Zukunft liegende Züge oder Rewards entsprechend weniger stark berücksichtigt.

Ziel ist es, nun eine geeignete Policy zu finden, die es erlaubt, möglichst gute und realisierbare Werte innerhalb der Value-Funktion zu erzeugen. Wie man sich vorstellen kann, würde eine Policy, die jede Zugmöglichkeit im Gitternetz erlaubt, möglicherweise weniger gute Ergebnisse über die Value-Funktion generieren, als eine Policy, die die Schritte in Vorwärts- (also zielorientierte) Richtung einschränkt. Die erstgenannte Policy würde die Möglichkeit implizieren, dass der Agent Schritte zurück zum Startfeld und damit weg vom Zielfeld vornimmt, was ein Erreichen des Ziel in weitere Ferne rücken ließe und damit die Bewertung des jeweiligen Feldes negativ beeinflussen würde. Die zweite Policy würde sichergehen, dass jedes Feld auf dem Weg zum Zielfeld maximal einmal besucht würde, da kein Rückwärtsschritt erlaubt wäre.

2.2.3 Aktionsbewertende Q-Funktion

Im Gegensatz zur situationsbewertenden Value-Funktion kann man mit der sogenannten Q-Funktion die Aktionen bewerten. In manchen Fällen ist es sinnvoller, Aktionen anstatt Zustände zu bewerten, weil es z.B. darum geht, Aktionen auf Qualität zu untersuchen.

5 Stenmark (2005), Seite 41 ff
6 Vgl. Markelic (2004), Seite 12
7 Vgl. Sutton/Barto (1998), Kapitel 3.7

Die Q-Funktion lässt sich wie folgt definieren: $Q^{\pi}(s_t, a_t) = r_t + y \cdot r_{t+1} + y^2 \cdot r_{t+2} + ...$

Ähnlich der Value-Funktion bestimmt sich der Q-Wert zum Zeitpunkt x aus der Summe der (möglicherweise) aufeinanderfolgenden Rewards unter Berücksichtigung des Discount Faktors.

Hier bestimmt aber neben der Situation s_t auch die gewählte Aktion a_t den Rahmen, innerhalb dessen diese Funktion ausgeführt wird. Die Formeln und weitere Erklärungen dazu können in Sutton/Barto (1998) ab Kapitel 3.7 nachgelesen werden.

2.3 Ziel der Reinforcement Learning Strategie

Ziel der RL-Idee ist es, die Value-Funktion möglichst so zu approximieren, dass sie aussagekräftige Ergebnisse über die zu bewertenden Zustände liefert. Konkret funktioniert RL so, dass die Policy sich den Folgezustand beziehungsweise die Aktion sucht, bei der s_{t+1} durch die Value-Funktion den größten Wert zugeordnet bekommt.

Um beim Beispiel des Gitternetzes zu bleiben, würde eine geeignete Policy das Feld als nächsten Zustand wählen, welches durch die Value-Funktion den größten Wert erhalten würde. Äquivalent hierzu würde eine aktionsbewertende Policy die Aktion wählen, die den größtmöglichen Wert durch die Q-Funktion erhalten würde.

Das Ziel ist letztlich das Gleiche; ob aktions- oder zustandsbewertende Funktion, es geht um eine gute Approximation einer Funktion, die möglichst optimale Aussagen oder Ergebnisse über einen Sachverhalt liefert.

Der Ablauf der RL-Strategie ist dabei immer Gleich, es wird rückwärts eine Funktion approximiert, die nach Lernabschluss eine Bewertung über bestehende Zustände oder Aktionen zulässt.

Man kann nun fragen, warum rückwärts gelernt wird. Die Antwort ist eigentlich einfach: RL kann erst dann einen Reward-Wert berücksichtigen, wenn eine Aktion zum Zielzustand führt beziehungsweise ein Zielzustand erreicht wird. Dann wird der Value- oder auch der Q-Funktion ein Belohnungs- oder Bestrafungswert zugeordnet. Aus diesem Reward ergibt sich die Bewertung des Zustands oder der Aktion vor Erreichen des Zielzustand, da durch die Funktionen immer der Zeitpunkt t betrachtet wird (vgl. Formel in Kapitel 2.2.2 und 2.2.3). Da nun ein Wert für den dem Zielzustand vorausgehenden Zustand (oder der Aktion) bekannt ist, lässt sich im nächsten Durchlauf auch der Zustand bewerten, der diese gerade bewerteten vorausgeht und so weiter. Somit verfolgt RL eine rückwärts lernende Strategie, die bei genügender Anzahl an Beispieldaten und geeigneter Zufallswahl mancher Zustände oder Aktionen alle Zustände oder Aktionen entsprechend bewertet.

2.4 Pseudocode des RL-Algorithmus

In diesem Kapitel soll der Algorithmus für Reinforcement Learning in Pseudocode-Schreibweise dargestellt werden, um einen besseren Einblick in die Schritte des Lernprozesses zu gewährleisten. Dabei sind textuelle Anweisungen natürlich als programmierte Methoden oder Variablenzuweisungen zu verstehen. Die Variable „s“ steht mit ihren Indizes für eine spezielle Spielsituation zum Zeitpunkt t_x, wobei das „x“ mit dem entsprechenden Index übereinstimmt. Diese Spielsituationen müssen ebenso bekannt (oder erzeugbar) sein wie die Rewards, die den entsprechenden Spielsituationen zugeordnet sind.

Da der Algorithmus dem Prinzip der „temporal difference“[8] folgt, welches sogenannte „eligibility traces“ verwendet, die zusätzlich durch die Variable λ gesteuert werden, nennt er sich auch TD(λ)-Algorithmus. λ steuert dabei sozusagen das Gedächtnis des Algorithmus, indem die eligibility traces des vorherigen Spielzuges optional in den Wert der aktuellen eligibility traces einwirken können.

Der TD(λ)-Algorithmus[9]

Input: Spielverlauf / Zustände $\mathbf{s_0}, \mathbf{s_1}, \ldots \mathbf{s_N}$ und die zugehörigen Rewards $r(\mathbf{s}_t)$.

{

Setze $V_{old} = V(\mathbf{s_0})$ und $\mathbf{e}_0 = \mathbf{0}$.

Setze $\mathbf{e}_1 = \gamma\lambda \mathbf{e}_0 + \nabla_{\mathbf{w}} f(\mathbf{w}; \mathbf{s}_0)$

Für t=0,…,N-1 {

Response $y = f(\mathbf{w}; \mathbf{s}_{t+1})$ aus neuronalem Netz und

Reward $r_{t+1} = r(\mathbf{s}_{t+1})$ aus Spielumgebung abholen

$\delta_t = r_{t+1} + \gamma V(\mathbf{s}_{t+1}) - V_{old}$

Lernschritt: $\mathbf{w} = \mathbf{w} + \alpha \delta_t \mathbf{e}_t$

Response $y = f(\mathbf{w}; \mathbf{s}_{t+1})$ erneut berechnen (geändertes **w**!);

$V_{old} = y$;

$\mathbf{e}_{t+1} = \gamma\lambda \mathbf{e}_t + \nabla_{\mathbf{w}} f(\mathbf{w}; \mathbf{s}_{t+1})$

}

} // Ende TD(λ)-Algorithmus

Der Ablauf der Methodik ist immer derselbe. Es werden Variablen initialisiert und Startwerte gesetzt. Das verdeutlichen die ersten zwei Zeilen. Innerhalb der Schleife, die solange durchlaufen wird, bis das Spiel beendet ist – also alle Spielsituationen s_0 bis s_x abgearbeitet wurden -, wird der beste Spielzug gewählt (über die response-Zeile), der Asuwertungsfehler berechnet (Zeile mit Formel für) und der Lernschritt durchgeführt. Weiter werden die Variablen V_{old} und e_{t+1} für den nachfolgenden Schleifendurchlauf mit neuen, aktuellen Werten belegt.

8 Weil für die meisten Spielzustände das Fehlersignal im Wesentlichen die zeitliche Differenz in der Spielfunktion ist, nennt man den hierauf beruhenden Algorithmus **Temporal Difference (TD)** Algorithmus. Vgl. Konen (2006), S. 5

9 Pseudocode aus Konen (2006), S. 6

3 Erlernen des Spiels „Nimm-3“ mittels RL

3.1 Grundlagen zum Spiel „Nimm-3“

Das Spiel Nimm-3 ist ein einfaches Brettspiel mit geringem, aber durchaus vorhandenem strategischen Anteil. Nimm-3 beschreibt ein Spiel für zwei Personen, bei dem man von einem existierenden Spielbrett ein, zwei oder drei Steine entfernt.

Die Abbildung 2 zeigt die Wahl des möglichen Zuges. Ein Spieler kann maximal drei Steine, muss aber mindestens einen Stein vom Spielbrett entfernen. Damit verändert sich natürlich nach jedem Zug die Anzahl der restlichen, auf dem Spielbrett verbliebenen Spielsteine, was letztlich den Sinn, aber auch den Reiz des Spiels ausmacht. Denn nur über die Änderung der restlichen Spielsteine ändern sich auch die zu wählende Strategie. Es hängt von der Anzahl restlicher Steine ab, wie viele Steine ein Spieler sinnvollerweise entfernen sollte, damit er seine Siegchance behält.

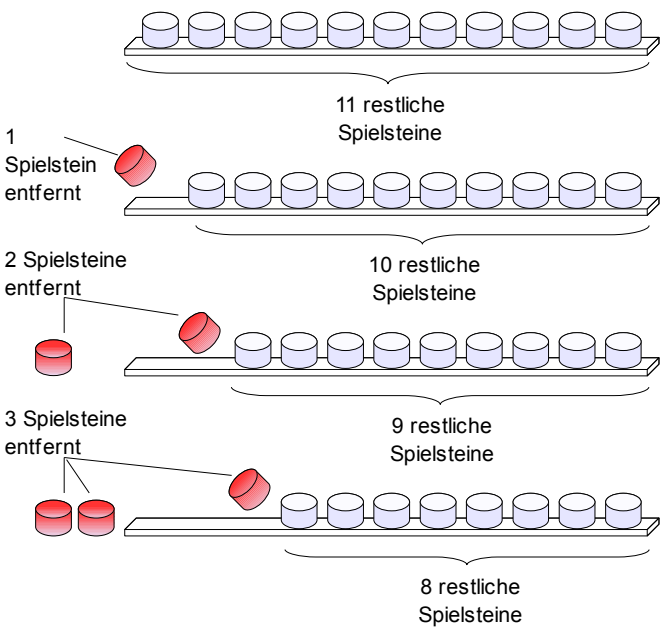


Abbildung 2: Zugmöglichkeit beim Spiel Nimm-3

Ziel des Spiels ist es, das Spielbrett möglichst zu leeren. Das heißt, dass möglichst kein Spielstein mehr auf dem Spielbrett liegen bleiben sollte (natürlich unter Berücksichtigung der Regeln). Wer dies zuerst erreicht, gewinnt das Spiel. Bei bis zu drei liegen gebliebenen Spielsteinen auf dem Spielbrett, hat der Spieler, der am Zug ist, schon gewonnen, weil er ja bis zu drei Steine entfernen kann.

Der strategische Spielverlauf, um ein Spiel für sich zu entscheiden, sollte daher so aussehen, dass ein Spieler versucht, mit seinem Zug möglichst vier Spielsteine auf dem Spielbrett zu hinterlassen. Der Grund hierfür ist im Prinzip recht eindeutig: Durch das Hinterlassen von vier Spielsteinen auf dem Spielbrett hat der Gegner keine Chance, dass Spiel zu beenden. Unabhängig wie viele Steine dieser nun entfernt, mit dem folgenden Zug gewinnt der Spieler, der zuvor vier Spielsteine hinterlassen hat.

In Abbildung 3 wird dieses gerade angesprochene Konzept verdeutlicht. Da sich diese Spielstrategie auch auf die des Gegners übertragen lässt, ist es von besonderer Wichtigkeit auch in der frühen Phase des Spiels eine Anzahl an Spielsteinen zu hinterlassen, die ein Vielfaches von „4“ darstellt. Es findet die gleiche Begründung Anwendung, die bereits beim Beenden des Spiels galt: Hinterlässt man zum Beispiel acht Spielsteine, hat der Gegner mit seinem Zug keine Chance, letztlich vier Steine zu hinterlassen und damit das Spiel vorab für sich zu entscheiden. Diese Überlegung sollte schon beim ersten Zug beachtet werden, so dass man möglichst früh seinem Gegner keine Chance lässt.

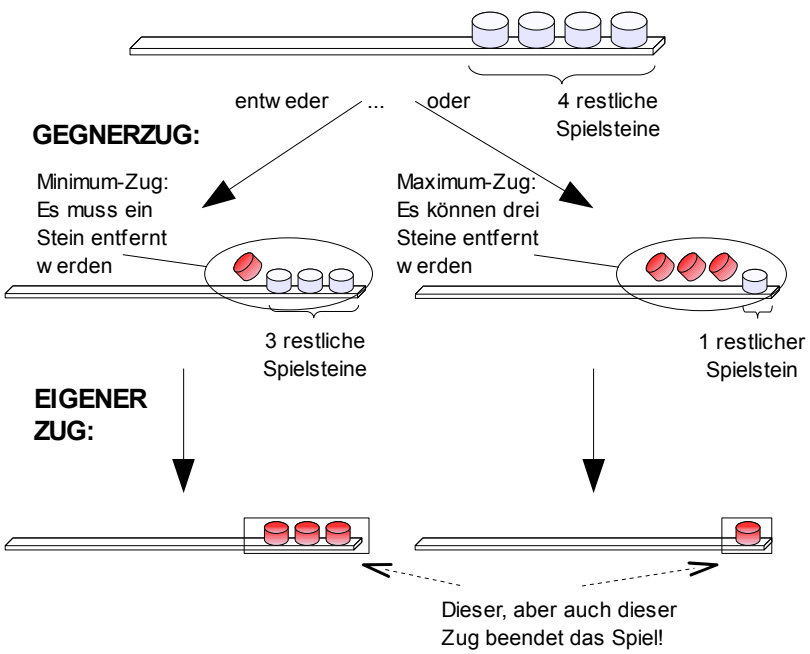


Abbildung 3: perfekte Zugstrategie zum Spielgewinn

Das Spiel ist algorithmisch einfach zu lösen, weil im Grunde nur ein hinterlassener Spielstand von „x * 4“ (ein Vielfaches von 4) Spielsteinen erzeugt werden muss. Es kommt weder auf die Position einzelner Steine an – wie dies zum Beispiel bei Schach der Fall ist –, noch spielt die Zusammensetzung des Spielfeldes eine große Rolle – wie es beispielsweise bei „4-Gewinnt“ vorkommt.

Es steht aber auch fest, dass bei Nimm-3 die anfängliche Spielfeldgröße über die Gewinnmöglichkeit des beginnenden Spielers entscheidet. Der Anziehende (das entspricht dem beginnenden Spieler) gewinnt das Spiel immer dann, wenn er mit perfekter Strategie spielt und eine Spielfeldgröße vorfindet, die ungleich einem Vielfachen von „4“ ist. Entspricht die anfängliche Spielfeldgröße einem Vielfachen von „4“, hat der Anziehende schon vor seinem ersten Zug das Spiel verloren – vorausgesetzt, sein Gegner spielt mit perfekter Strategie.

Aufgrund dieser Tatsache hat das Spiel Schwachstellen und ist leicht zu durchschauen. Aber genau deshalb eignet es sich gut als Test der RL-Strategie, da es in eine einfache Struktur gegliedert werden kann und eine offensichtliche Lösung existiert. Somit sollte man recht schnell eine korrekte Funktionsweise der RL-Strategie erkennen, falls diese richtig implementiert ist.

3.2 Pseudocode des Self-Play-TD(λ)-Algorithmus

In Abwandlung zum in Kapitel 2.4 vorgestellten „Basis-“TD(λ)-Algorithmus, wird in den nachfolgenden Kapiteln bei Lernvorgängen immer der sogenannte Self-Play Algorithmus genutzt. Dieser beeinhaltet die gleichen spielstrategischen Anweisung, wie der TD(λ)-Algorithmus, bietet aber zusätzlich die Möglichkeit ein Spiel vollständig selbst zu spielen, beide Spieleraktionen zu bewerten und daraus zu lernen. Der Vorteil ist das nicht benötigte manuelle Eingreifen, außer bei der Bewertung eines spielbeendenden Zuges durch den Reward.

Der Self-Play-TD(λ)-Algorithmus[10]

Input:Start-Player p [=+1 (Weiss) oder -1 (Schwarz)], Initialposition $\mathbf{s_0}$, sowie eine (teiltrainierte) Funktion $f(\mathbf{w};\mathbf{s}_t)$ zur Berechnung der Spielfunktion $V(\mathbf{s}_t)$ {

Setze $V_{old} = V(\mathbf{s_0})$ und $\mathbf{e}_0=\mathbf{0}$ [Beachte: zu $\mathbf{s_0}$ gehört Player –p]

Setze $\mathbf{e}_1 = \gamma\lambda\,\mathbf{e}_0 + \nabla_{\mathbf{w}} f(\mathbf{w};\mathbf{s}_0)$

Für t=0, …, maxN {

entweder (1) oder (2), je nach Wahrscheinlichkeitswahl

(1) Wahrscheinlichkeit ε: Wähle einen für Player p erlaubten After-State $\mathbf{s}_{t+1}$ **zufällig** aus [explorativer Move = Random Move]

(2) Mit (1-ε): Suche den für Player p erlaubten After-State $\mathbf{s}_{t+1}$, der $p*f(\mathbf{w};\mathbf{s}_{t+1})$ maximiert [greedy Move]

Response $y = f(\mathbf{w};\mathbf{s}_{t+1})$ des NN und Reward $r_{t+1}= r(\mathbf{s}_{t+1})$ aus Spielumgebung abholen

Fehlersignal $\delta_t = r_{t+1} + \gamma V(\mathbf{s}_{t+1}) - V_{old}$

Wenn $\mathbf{s}_{t+1}$ kein Random Move:

Lernschritt: $\mathbf{w} = \mathbf{w} + \alpha\,\delta_t\,\mathbf{e}_t$ [bringt $V(\mathbf{s}_t) = f(\mathbf{w};\mathbf{s}_t)$ näher an $r_{t+1}+\gamma V(\mathbf{s}_{t+1})$ heran]

Wenn $\mathbf{s}_{t+1}$ Spielendstand $\mathbf{s}_N$:

break; [For-Loop verlassen]

Response $y = f(\mathbf{w};\mathbf{s}_{t+1})$ erneut berechnen (geändertes **w**!)

$V_{old}=y$

$\mathbf{e}_{t+1} = \gamma\lambda\,\mathbf{e}_t + \nabla_{\mathbf{w}} f(\mathbf{w};\mathbf{s}_{t+1})$ [dies wird das $\mathbf{e}_t$ für die nächste Iteration]

Setze p = – p [Spielerwechsel]

}

} // Ende „Self-Play“

Der Algorithmus entscheidet selbst über die Folgen, die ein Spielzug eines Spielers auf das Spiel hat. Damit ist – wie bereits erwähnt – kein Eingreifen von außerhalb notwendig. Außerdem unterscheidet er zwischen Zufallszügen zur Verbreiterung des Zustandsraumes und gezielten „greedy“-Zügen, die je nach Response des NN getätigt werden.

10 Pseudocode aus Konen (2006), S. 7

3.3 Tabellarische Spielfunktion

Ziel eines jeden Spiele-Agenten sollte es sein, eine geeignete Spielfunktion zu entwickeln, die es dem Agenten ermöglicht, das Spiel möglichst erfolgreich zu spielen. Nutzt man die RL-Strategie zur Entwicklung einer solchen Spielfunktion, erwartet man, dass die Spielfunktion über viele Beispieldaten rückwärts generiert wird (vergleichbar Kapitel 2.3).

Anhand einer einfachen tabellarischen Struktur wird nachfolgend versucht, für das Spiel Nimm-3 eine gute Spielfunktion zu generieren. Dabei wird der jeweils zu tätigende Spielzug letztlich aus der Tabelle gelesen. Die Tabelle erhält folgende Funktion: In ihr wird die Bewertung der Spielzüge, die ein jeder Spieler hinterlässt, abgespeichert. Der passende Wert ergibt sich aus den gelernten Rewards, die sich mit Hilfe der RL-Strategie errechnen lassen.

3.3.1 Aufbau der tabellarischen Struktur

Im Folgenden wird von einem initiierten Spielbrett mit elf Spielsteinen ausgegangen, sodass der startende Spieler mit der Spielfarbe „weiß“ bei perfekter Strategie gewinnen sollte. Die Entscheidung über die Anzahl der zu ziehenden Spielsteine werden beide Spieler nur anhand der Werte der Tabelle treffen. Die anfängliche tabellarische Struktur wird in Abbildung 4 gezeigt. Hier erkennt man, dass alle Funktionswerte anfänglich auf „0“ gesetzt sind. Das ist eine sinnvolle Belegung, da ohne Lernvorgang über Beispieldaten keine Information über die Spielfunktion vorliegen kann.

WEISS		**SCHWARZ**	
Anzahl hinterlassende Steine	Funktionwert durch RL	Anzahl hinterlassende Steine	Funktionwert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	0	3	0
2	0	2	0
1	0	1	0
0	0	0	0

Abbildung 4: Tabellarische Spielfunktion initiiert

Ziel ist es nun, jeden After-State beider Spieler – der entstehen kann – zu bewerten. RL hilft durch die Vergabe eines Rewards am Spielende, die Spielfunktion entstehen zu lassen. Letztlich sagen die Werte der jeweiligen After-States aus, wie wahrscheinlich ein Sieg des Anziehenden mit dieser Situation als Grundlage ist. Also auch die Werte bei Schwarz tätigen eine Aussage über die Siegwahrscheinlichkeit von Weiß (aus der betreffenden Situation).

3.3.2 Nicht bewertbare Zustände

Einige Zustände, die in der Tabelle existieren, können nicht bewertet werden. Es lässt sich sagen, dass es trotz Lernvorgängen nie eine Bewertung für elf hinterlassende Steine von Weiß geben wird, genauso wenig wird die Situation „10 hinterlassende Steine von Schwarz“ bewertet, so lange Weiß der anziehende Spieler ist. Das liegt daran, dass die jeweils erste Spalte der Spieler den sogenannten After-State darstellt (After-State ist die resultierende Situation nach einer Aktion). Wenn aber Spieler Weiß das Spiel mit seinem Zug beginnt, kann es nie einen After-State von 11 für ihn geben. Bei Schwarz ist das anders, denn wenn Weiß beginnt, findet er sozusagen einen After-State von Schwarz vor, weil er eine Situation bearbeitet, die ihm hinterlassen wurde.

Für 10 hinterlassende Steine von Schwarz wird es auch nie eine Bewertung geben, solange Weiß das Spiel beginnt. Der Grund entspricht dem obigen, denn wenn Weiß startet und er würde auch nur einen Stein entfernen, existieren schließlich nur noch 10 Steine, die Schwarz als Situation vorfindet, er kann aber nie selber einen After-State von 10 erzeugen, da er das Spiel nicht beginnt. Zieht Weiß mehr als einen Stein, gilt natürlich das Gleiche.

Etwas schwieriger – und nicht direkt erkennbar – ist die Tatsache, dass es auch nie eine Bewertung des After-States „0“ geben wird, hier ist sogar egal, welcher Spieler diesen After-State erzeugt. Die Begründung ist im Grunde nur verständlich, wenn man sich die Bewertung einer Situation mittels RL-Strategie anschaut.

Eine Situation (also ein After-State eines Spielers) wird durch die folgende Berechnung bewertet: $V(s_t)=V(s_t)+\alpha\cdot\delta(t)$ mit $\delta(t)=r_{t+1}+\gamma\cdot V(S_{t+1})-V(s_t)$

Der alte Funktionswert wird durch Hinzunahme eines Wertes resultierend aus dem Produkt aus Schrittweite und Fehlerwert verändert. α bezeichnet dabei die sogenannte Lernschrittweite, die bestimmt, wie schnell oder stark ein bestehender Wert verändert werden soll – sie bestimmt schließlich den „Einfluss“ des Fehlers auf den neuen Funktionswert. Der Fehlerwert berechnet sich aus dem Reward – der erst am Ende eines Spiels verteilt werden kann – plus dem mittels Discount-Faktor γ „abgezinsten“ neuen Funktionswert, der dem neu entstandenen After-State zugeordnet ist minus dem bisherigen Funktionswert des aktuellen After-States.

Wird ein Spiel nun durch einen Zug beendet, befindet man sich beispielsweise in der Situation $V(s_T)$ (T steht für terminierend). Man kann aber für den Zustand s_T keinen Fehlerwert berechnen, da innerhalb der Formel der Funktionswert des Zustands s_{t+1} benötigt wird, der am Ende eines Spieles aber nicht existiert. Daher ist die Formel nur bis zum Zustand s_{T-1} definiert, was zur Folge hat, dass ein After-State von „0“ Spielsteinen nie bewertet wird.

3.3.3 Generierung der tabellarischen Spielfunktion

Anhand der im vorangegangenen Kapitel beschriebenen Formeln zur Erzeugung einer situationsbewertenden After-State-Funktion kann nun eine tabellarische Spielfunktion erzeugt werden, die alle möglichen After-States nach Siegwahrscheinlichkeit des Anziehenden bewertet. Dabei wird die initiierte Tabelle aus Abbildung 4 als Grundlage der Spielfunktion angenommen.

Sobald das Spiel beginnt, wird – wie bereits erwähnt – die Entscheidung über die Anzahl der zu entfernenden Spielsteine abhängig von dem Wert des zugehörigen After-States aus der Tabelle gemacht. Das bedeutet, dass jeder Spieler zu Beginn seines Zuges analysiert, welche möglichen After-States er erzeugen kann und welche Bewertung diesen After-States entspricht.

Angenommen, Spieler „Weiß“ beginnt das Spiel, was mit elf Spielsteinen initiiert worden ist. So schaut der besagte Spieler innerhalb der Tabelle zunächst nach den Bewertungen der After-State zehn, neun und acht, die aber alle Spieler „Weiß“ zugeordnet sind. „Weiß“ wählt nun die günstigste Bewertung, – das bedeutet gleichzeitig, die Höchste – um seinen Zug zu tätigen und entfernt die entsprechende Anzahl an Spielsteinen.

Diesen Vorgang wiederholen beide Spieler abwechselnd, bis das Spiel durch Sieg und Niederlage entschieden ist. Bei aussageloser – also mit „0“ initiierter – Funktionstabelle, wie sie im obigen Fall vorliegt, würde der Spielverlauf bei jeden Zug durch die Wegnahme eines Spielsteines beschrieben, so dass vor dem letzten möglichen Zug, Spieler „Weiß“ einen zuletzt liegengebliebenden Stein entfernen kann. Dies tut er und beendet damit das Spiel. Durch die RL-Strategie wird die Tabelle nun mit Information gefüllt, da ein Episodenabschnitt erreicht ist, der bewertet werden kann (vergeichbar einem bewertbaren After-State).

Es werden erneut die Formel aus Kapitel 3.2.2 angewandt:

$$V(s_t) = V(s_t) + \alpha \cdot \delta(t) \quad \text{mit} \quad \delta(t) = r_{t+1} + \gamma \cdot V(S_{t+1}) - V(s_t)$$

Im Unterschied zu den bisherigen Spielzügen wird die Fehlerberrechnung hier ungleich „0“ betragen, da ein Reward ausgeschüttet wird, der ungleich „0“ ist und damit einen Einfluss auf die After-State Bewertung hat. Der Wert von $V(s_{t+1})$ beträgt ebenso wie der Wert von $V(s_t)$ „0“ und spielt damit für die Fehlerberechnung keine Rolle. Der Fehler der Situation, die zum Spielende führte, beträgt damit genau so viel, wie der festgelegte Reward dieser Situation (sollte „Schwarz“ gewonnen haben, wird der Reward sinnvollerweise negativ sein). Je nach Lernschrittweite verändert der Fehler nun die Bewertung der Situation $V(s_t)$.

Ein Beispiel: Angenommen, der Reward bei Spielende beträgt 100 für Sieg des Anziehenden und -100 für Niederlage. Die Lernschrittweite beträgt 1 und der Discount-Faktor beträgt 1.

Nach einem kompletten Spieldurchgang würde sich die Tabelle im Vergleich zur Initialtabelle nur durch einen Wert unterscheiden; dies ist die Situation, die zum Spielende und zur finalen Bewertung geführt hat. Abbildung 5 zeigt dabei, dass sich nur die Bewertung von einem hinterlassenden Stein für Schwarz verändert hat. Der Wert „100“ entsteht aus den obigen Formeln der situationsbewertenden Funktion $V(s_t)$. Diese 100 sagt somit aus, dass eine 100prozentige Wahrscheinlichkeit auf den Gewinn des Spiels durch Spieler „Weiß“ - dem Anziehenden – besteht.

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionwert durch RL	Anzahl hinterlassende Steine	Funktionwert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	0	3	0
2	0	2	0
1	0	1	**100**
0	0	0	0

Abbildung 5: Tabellarische Spielfunktion (ein Spieldurchlauf)

Warum werden keine weiteren Bewertungen vorgenommen? Weil der zu berechnende Fehler immer von der Differenz aus Situation nach dem Zug und Situation vor dem Zug besteht, ein Reward hat bekanntlich nur am Spielende Einfluss auf den Fehler. Da sich bisher die Bewertungen aller Situationen aufgrund der Initialisierung gleichen, existiert auch kein Fehler, der eine Bewertung im Laufe des Spiel verändern könnte. Nur am Ende spielt der vergebene Reward eine Rolle und lässt den Fehler einen Wert ungleich „0“ annehmen, was direkt eine Veränderung der Situationsbewertung zur Folge hat.

Ließe man nun weitere Spiele als Lerndaten durchlaufen, würden sich auch die Bewertungen der anderen Situationen verändern. Man kann sich vorstellen, dass der Fehler bei der Differenz von alter zu neuer Situation auch durch die bereits existierende „100“ beeinflusst wird und somit bestehende „frühere“ Situationen im Spiel anders bewertet als zuvor. Im Anhang A.1 sind alle Tabellenzwischenstände aufgezeigt, die zur chronologischen Entwicklung einer tabellarischen Spielfunktion beigetragen haben.

Nach vielen Spielen als Beispiel-Lerndaten ist die Tabelle soweit mit Bewertungen gefüllt, dass ein Spieler, der dieser Bewertungsregel folgt, das Spiel durchaus auch gegen starke algorithmische oder menschliche Gegner spielen kann und sich dabei bewährt, das heißt, dass er so spielt, dass er als Anziehender gewinnt.

3.4 Lineare Spielfunktion über lineares Netz

3.4.1 Arten alternativer Spielfunktionen

Man kann sich vorstellen, dass eine tabellarische Speicherung aller in Frage kommenden Spielsituationen schnell unmöglich wird.

Dies tritt genau ein, wenn so viele erzeugbare After-States vorliegen, dass diese nicht mehr in realer Zeit abzuspeichern sind. Beim Spiel „Nimm-3“ ist die Zahl der möglichen After-States auf $2 \cdot (x+1)$ beschränkt (wobei „x“ die Spielfeldgröße zu Beginn des Spiels darstellt).

Warum dies so ist, erklärt folgendes Beispiel:

Angenommen, ein Nimm-3 Spiel startet mit elf Spielsteinen. Es ist maximal möglich, dass in mehreren Spielen beide Spieler je 12 After-States erzeugen. Die Zahl 12 kommt zustande, da der Zustand „leeres Spielfeld“, ebenso wie die Zustände „elf hinterlassende Spielsteine“ (zu Beginn des Spiels) bis „ein hinterlassender Spielstein“, auch ein möglicher After-State ist. Da diese After-States aber von zwei Spielern erzeugt werden können, rechnet man 12 * 2, um die Anzahl der möglichen, gesamten After-States zu ermitteln.

Es ist erkennbar, dass bei einem komplexeren Spiel, wie zum Beispiel „Schach“ oder „4-Gewinnt“ die Anzahl möglicher After-States schon alleine dadurch wächst, dass ein Spielbrett nun durch viele, positions- und besetzungsrelevante Spielfelder definiert wird. Anders als beim Spiel „Nimm-3“ kommt es bei den gerade erwähnten Spielen darauf an, wo welcher Spieler seinen Spielstein hinzieht beziehungsweise welchen Spielstein (bei Schach) hinterlässt. Ein After-State setzt sich nun aus wesentlich mehr Information zusammen, als dies beim bisherigen Spiel „Nimm-3“ der Fall ist.

Oft ist durch eine einfache mathematische Berechnung nicht nachweisbar, wie viele After-States möglich sind, aber generell lässt sich die Größenordnung der generierbaren After-States einschätzen.

Das folgende Beispiel zeigt auf, warum dies häufig der Fall ist:

Beim Spiel „4-Gewinnt“ kommt es neben der Spielfeldposition darauf an, welcher Spieler seinen Spielstein auf das entsprechende Spielfeld gelegt hat, oder ob es überhaupt besetzt ist. Das heißt, es existieren drei Möglichkeiten ein Spielfeld zu belegen. Das ergibt bei 42 Spielfeldern, die im „Standard-4Gewinnt-Spielbrett“ existieren (sechs Zeilen mal sieben Spalten), insgesamt 3^{42} mögliche After-States. Das bedeutet, dass es möglich wäre 109.418.989.131.512.359.209 unterschiedliche After-States zu erzeugen – das sind etwa 109 Trillionen unterschiedliche Spielsituationen.

Hier ist natürlich zu berücksichtigen, dass es Regeln gibt, die bestimmte After-States ausschließen. Zur Optimierung dieser Zahl kann man folgende Überlegung anstellen:

Jedes Spielfeld kann drei Zustände haben. Es kann unbelegt, belegt durch Spieler rot oder durch Spieler gelb sein (siehe Abbildung 6). Dabei ist der Zustand „freies Feld“ separiert zu betrachten, da über ein freies Feld keine weiteren Spielsteine gelegt werden können. Daher kann man jedes Feld mit (2+1) Zuständen betrachten.

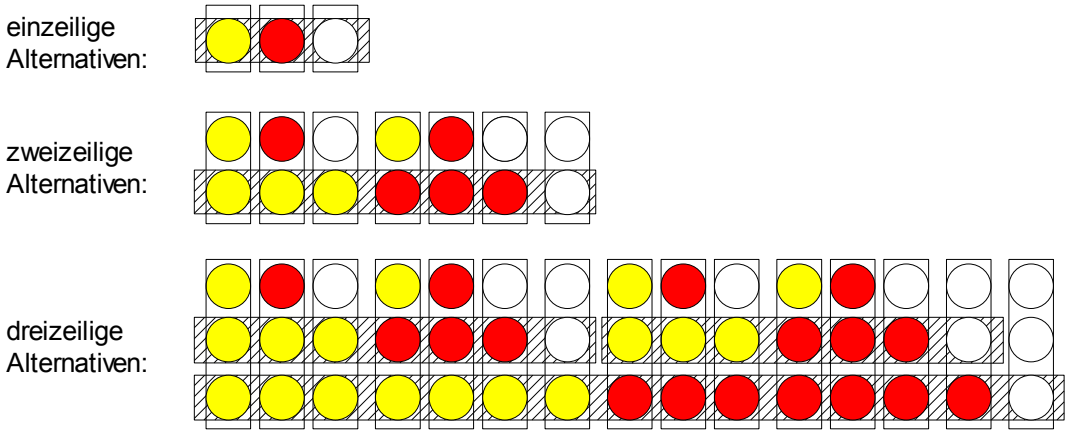


Abbildung 6: Auszug der Belegungsalternativen pro Spielspalte

Für alle Felder einer Spalte gilt daher nachfolgende Rechnung:

$2\cdot(2\cdot(2\cdot(2\cdot(2+1)+1)+1)+1)+1+1=2^7-1$. Bei 7 Spalten ergibt das: $2^{49}\approx 5{,}62\cdot 10^{14}$.

Die oben beschriebene Tatsache begründet vollständig neue Überlegungen zur Generierbarkeit von Spielfunktionen zu komplexeren Spielen. Sie macht eine Änderung und gleichzeitig eine Verallgemeinerung einer Spielfunktion notwendig, da nicht mehr alle After-States eindeutig bewertet und tabellarisch (oder in anderer Form) abgespeichert werden können.

Es ist aber denkbar, dass Spiele lösbar sind, wenn nur wenige Faktoren oder Muster bekannt sind. Beim vorliegenden Spiel „Nimm-3“ ist die Tatsache einen After-State zu erzeugen, der in der Spielsteinanzahl durch vier teilbar ist, die perfekte Codierung der Spielfunktion. Man benötigt nun nicht mehr alle Bewertung der möglichen After-States, sondern rechnet pro Spielzug nur aus, ob die entstehende Anzahl an Spielsteinen durch vier teilbar ist oder nicht. Das heißt, es existiert aus dieser Regel quasi automatisch ein Fakt, der als ein perfektes Feature in eine Spielfunktion integriert werden kann.

Aus diesen Überlegungen lässt sich somit verallgemeinert festhalten, dass es eine Art Feature-Vektor[11] geben muss, der Grundlage einer Spielfunktion wird, um das Spiel möglichst genau zu berechnen beziehungsweise spielen zu können.

11 Der Feature-Vektor beschreibt die Zusammenfassung mehrerer wichtiger Fakten, die zur Lösbarkeit eines Spiels beitragen, indem sie zu Faktoren einer linearen oder mehrdimensionalen Spielfunktion werden.

Nun kann man sich vorstellen, dass Features, die in einem Spiel wichtig sind, alle den gleichen, wichtigen Einfluss auf die Lösbarkeit des Spiels haben. Somit würden sie jeweils zu linearen Parametern innerhalb einer Spielfunktion, die letztlich eine möglichst genaue Approximation zur perfekten Spielcodierung darstellen soll:

$$f(\vec{x}) = \sum \vec{x}_n \; mit \; n \in \mathbb{N} \quad \text{oder} \quad f(\vec{x}) = \vec{x}_1 + \vec{x}_2 + .. + \vec{x}_n$$

Selbst bei unterschiedlich starkem Einfluss der einzelnen Features auf die Spiellösbarkeit, würde eine lineare Funktion entstehen, die jedem Feature eine Gewichtung zuordnen würde.

Es resultiert zum Beispiel die folgende lineare Funktion: $f(\vec{x}) = a_1 \cdot \vec{x}_1 + a_2 \cdot \vec{x}_2 + ... + a_n \cdot \vec{x}_n$

Sobald bestimmte Features in mehrdimensionaler Weise unterschiedlichen Einfluss auf die Lösbarkeit eines Spiels haben, verändert sich die Spielfunktion von einer linearen zu einer mehrdimensionalen Funktion, dies ist dann der Fall, wenn der Feature-Vektor nicht mehr linear abbildbar ist, er also mehrere Vektoren (Parameter) zur Darstellung des Features benötigt:

$$f(\vec{x}) = \sum \vec{a}_n \cdot g(\vec{x}_n, \vec{y}_n) \; mit \; a \in \mathbb{R} \; und \; n \in \mathbb{N}$$

Eine solche, mehrdimensionale Spielfunktion als gute Approximation einer perfekten Spielcodierung zu finden, ist schwer und daher auch nicht mit einfachen Mitteln wie linearen Parametern oder linearer Feature-Vektor-Erzeugung zu lösen.

Im nächsten Kapitel wird die Generierung einer möglichen mehrdimensionale Funktion mit Hilfe von Neuronalen Netzen beschrieben. Hier wird zunächst die Erzeugung einer linearen Spielfunktion mit Hilfe eines linearen Netzes erläutert.

3.4.2 Erzeugung eines linearen Netzes

Ein lineares Netz ist nur eine Reduzierung eines neuronalen Netzes auf eine Eingabe- sowie eine Ausgabeschicht. Das heißt, im folgenden Netz existiert keine verdeckte Schicht, somit lassen sich nicht-lineare Probleme auch nicht lösen. Wählt man eine geeignete Kodierung, so ist das Spiel „Nimm-3“ aber linear lösbar. Das heißt, es macht auch die Verwendung eines linearen Netzes zur Erzeugung einer guten Spielfunktion Sinn. Ziel ist und bleibt schließlich, eine möglich genaue Approximation der perfekten Spielcodierung zu erhalten, die sich durch die Spielfunktion ausdrücken lässt.

Das lineare Netz bietet gegenüber der tabellarischen Spielfunktion den Vorteil, dynamischer und flexibler zu sein. Die Bedeutung der einzelnen Schichten ist im Gegensatz zu den tabellarischen Einträgen nicht von vorne herein fest definiert, sondern lässt sich frei wählen. Während in einer Tabelle für Spalten, Zeilen und auch einzelne Zellen bestimmte Funktionen festgelegt werden, kann innerhalb eines linearen (und neuronalen) Netzes, die Funktion der

Eingabe- und Ausgabeschicht variieren. Das heißt beispielsweise, dass nicht die einzelnen After-States als Eingabewerte dem Netz bekannt gemacht werden müssen, sondern die Eingabe auch über bestimmte Merkmale oder spielrelevante Fakten definiert sein kann.

Letztlich berechnet das lineare Netz nur die Abhängigkeit der Eingabewerte auf die Ausgabewerte. Das heißt, man stellt die Frage, wie sich die einzelne Eingabe in ihrem Wert und ihrer Relevanz auf die Ausgabe auswirkt.

Konkrete Anwendung findet das lineare Netz im vorliegenden Spiel „Nimm-3“, indem man die sogenannten Eingabeneuronen (das sind die einzelnen Eingabefaktoren) mit den jeweils möglichen After-States belegt – jedoch spielerunabhängig. Das bedeutet, dass insgesamt 12 Eingabeneuronen existieren, die jeweils für einen möglichen After-State im Spiel stehen.

Im Gegensatz zur Tabelle wird hier zunächst nicht nach dem Spieler unterschieden, der den After-State erzeugte, obwohl man weiß, dass diese Information ausschlaggebend für den Spielverlauf ist. Der spielerbezogene After-State wird allerdings über die Wertebelegung dieser Eingabeneuronen unterschieden, dabei steht eine angelegte „1“ für den aktivierten After-State des Spielers „Weiß“ und eine „-1“ entsprechend für „Schwarz“. Sollte ein Neuron nicht belegt sein, erhält es die „0“, was gleichzeitig bedeutet, dass dieses Neuron – und damit der entsprechende After-State – nicht aktiviert ist. In der Anwendung auf das Spiel „Nimm-3“ ist immer nur ein Eingabeneuron aktiv, da zu jedem Spielzeitpunkt nur eine After-State-Situation – die zu einem Spieler gehört – besteht. Das bedeutet wiederum, dass 11 Eingabeneuronen nicht aktiviert sind, also jeweils eine „0“ angelegt wird und nur ein Neuron durch eine angelegt „1“ aktiviert wird.

Als Größe der Ausgabeschicht wird nur ein Ausgabeneuron gewählt, dass jeweils die Wertigkeit eines bestimmten After-States angeben soll. Es entspricht den Werten der einzelnen Zellen der tabellarischen Spielfunktion. Die Ausgabe des linearen Netzes sagt somit etwas über die Gewinnwahrscheinlichkeit des anziehenden Spielers aus.

Jedes Eingabeneuron erhält – entsprechend der Theorie neuronaler Netze – eine Gewichtung, die durch eine Kante dargestellt wird, die zusätzlich zum entsprechenden Eingabeneuron auch mit dem Ausgabeneuron verbunden ist. Die resultierende Struktur zeigt eine netzartige

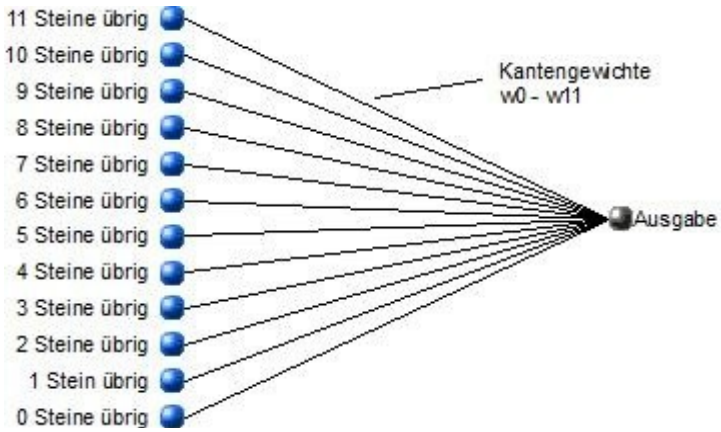


Abbildung 7: Struktur des linearen Netz

Verknüpfung von Eingabe- und Ausgabeschicht, was die Namensgebung begründet. Diese Netzstruktur ist in Abbildung 7 (auf der vorherigen Seite) dargestellt.

Das lineare Netz berechnet letztlich den Einfluss einzelner Eingabeneuronen auf seine Ausgabe, indem die aktivierten Eingabeneuronen mit der jeweiligen Gewichtung multipliziert und nacheinander zum Ausgabewert aufsummiert werden. Diese Berechnung entspricht der bereits bekannten Formel: $f(\vec{x}) = a_1 \cdot \vec{x}_1 + a_2 \cdot \vec{x}_2 + ... + a_n \cdot \vec{x}_n$, wobei $\vec{x}$ für die aktivierten Eingabeneuronen und „a_{Index}“ für die jeweilige Gewichtung steht.

3.4.3 Anwendung des linearen Netzes

Das im vorangegangenen Kapitel beschriebene lineare Netz wird nun zur grundlegenden Berechnung der Nimm-3-Spielfunktion verwendet. Dabei muss das Erlernen des linearen Zusammenhangs aus generiertem After-State und Ausgabewert sichergestellt werden.

3.4.3.1 Der Lernprozess

Man kann sich nun fragen, wie ein lineares Netz überhaupt lernt und wie ein Lernprozess abläuft. Um dies zu beantworten, nutzt man zwei Formeln, die es ermöglichen, rückwirkend Veränderungen an den Gewichten einzelner Eingabeneuronen vorzunehmen. Der eigentliche Lernprozess ähnelt dem Lernen bei neuronalen Netzen, nur dass hier keine Gewichtungen verdeckter Schichten verändert werden müssen, sondern nur die Verbindung von Eingabe- und Ausgabeneuronen überarbeitet wird.

Angenommen, die Gewichte eines linearen Netzes werden mit „w“ bezeichnet und die Spielfunktion – die Berechnung durch das lineare Netz – wird durch $f(w_t, s_t)$ dargestellt, dann lautet die Änderungsformel: $\vec{w_{t+1}} = \vec{w}_t + \alpha \cdot \delta_t \cdot \nabla_w f(\vec{w}_t, \vec{s}_t)$ mit $\delta_t = r_{t+1} + \gamma \cdot V(\vec{s_{t+1}}) - V(\vec{s}_t)$.

Dabei bezeichnet $\nabla_w f(\vec{w}_t, \vec{s}_t)$ das Gradientenabstiegsverfahren (siehe Kapitel 3.2).

Der ursprüngliche Wert der Kanten „w“ wird nach einem Zug durch Addition (beziehungsweise Subtraktion bei negativem Wert) des Lernwertes verändert. Dieser Lernwert besteht aus einem Produkt aus drei Faktoren. Diese sind die Lernschrittweite α, der Fehlerwert δ und der Gradientenabstiegswert $\nabla_w f(\vec{w}_t, \vec{s}_t)$ zum aktuellen Berechnungsergebnis der Spielfunktion. Der Fehlerwert δ berechnet sich aus der bekannten Formel, die Reward und die After-State-Werte nach und vor dem aktuellen Zug berücksichtigt.

Die Lernschrittweite als ein Parameter der Formel hat Einfluss auf die Lerngeschwindigkeit und Genauigkeit des linearen Netztes. Bei manchen Lernprozessen ist es sinnvoll, schnelle und vor

allem große Lernschritte zu machen.Hat man aber bereits eine gute Netzperformance erreicht, sollte die Lernschrittweite verringert werden, um minimale Fehlerkorrekturen durchzuführen und nicht zu ganz neuen – ungenaueren – Netzergebnissen zu gelangen. Typischerweise liegt der Wert der Lernschrittweite anfänglich bei 1 und wechselt zu einem späteren Zeitpunkt im Lernverfahren auf 0,1 – manchmal machen auch kleinere oder größere Werte Sinn.

Die Fehlerberechnung über δ gibt die Größe des Netzauswertungsfehlers an, der noch existiert. Angenommen eine Netzauswertung ergab einen Ausgabewert von 0,2, der Fehlerwert liegt aber bei 0,9, dann existiert eine Differenz von 0,7, die das Netz noch korrigieren muss. Daher gibt der Fehlerwert die Größe der bestehenden Unstimmigkeit zwischen Netzausgabe und Realität wieder.

Das Gradientenverfahren wird benutzt, um allgemeine Optimierungsprobleme zu lösen. Der Gradient gibt die Richtung des steilsten Abstiegs von einem Näherungswert ausgehend an. Das Verfahren sucht über den Gradienten den Optimalwert in einem Fehlergebirge.[12]

3.4.3.2 Netzperformance

Durch das ständige Lernen des Netzes wird eine dauerhafte Aktualisierung der Kanten sichergestellt. Wie leistungsstark ein Netz ist, lässt sich aus dem Lernprozesse alleine aber nicht sagen. Hierzu werden immer Daten benötigt, an denen stichprobenartig die Performance – also die Leistungsstärke des Netzes – gemessen werden kann. Häufig trennt man die vorhandene Datenmenge in mehrere Mengen auf, um Daten zu Trainingszwecken (also für den Lernprozess) und Testzwecken zu erhalten. Testdaten können sogar zusätzlich weiter aufgeteilt werden, sodass eine Testmenge und eine Validierungsmenge entsteht. Anhand letzterer Menge wird häufig im wirtschaftlichen Kontext die Stärke entwickelter Methoden nachgewiesen und – zum Beispiel für den Auftraggeber – verdeutlicht.

Im Spielekontext bedeutet das nun beispielsweise, dass zu Trainingszwecken – in denen das lineare Netz lernt – viele Spiele durchgeführt werden, um dem Netz anhand vieler Datensätze Beispielinstanzen möglicher Zustände aufzuzeigen und seine Reaktionsfähigkeit zu stärken (was lernen bedeutet).

Außerdem werden nach Generierung einer Spielfunktion über ein lineares Netz einzelne Spiele durchgeführt, um die Gewinnquote des linearen Netzes zu ermitteln. Das entspricht dem Testvorgang mittels oben beschriebener Mengen aus Test- und gegebenenfalls Validierungsdaten.

Natürlich spielt die Auswahl des Gegners bei der Messung der Gewinnquote – die

12 Vgl. Wikipedia (Gradientenverfahren)

ausschlaggebend für die Netzperformance ist – eine wichtige Rolle. So liegen beispielsweise zwischen einem Zufallsspieler – der jeden Zug zufällig wählt – und einem spielstarken, algorithmischen Gegner enorme qualitative Unterschiede. Dennoch lässt sich nur so ein Lernfortschritt des linearen Netzes messen, der letztlich ein Indikator der Netzperformance ist.

3.4.3.3 Praktische Umsetzung des RL-Agenten

Aus den theoretischen Überlegungen der vorangegangenen Kapitel und beschriebenen wissenschaftlichen Methoden lässt sich nun eine praktische Versuchsumgebung erstellen, die alle theoretischen Schlüsse und prognostizierten Ergebnisse auch bestätigen oder – im Zweifel – widerlegen sollte. Es geht schließlich unter anderem auch darum, die Verwendbarkeit der RL-Strategie nachzuweisen. Somit wurden mehrere entsprechende Agenten erstellt, die mittels RL in der Lage sind, das Spiel „Nimm-3“ zu erlernen und perfekt spielen zu können. Die Erzeugung der Spielfunktion wird dabei auf unterschiedliche Weise durchgeführt. Neben der tabellarischen Spielfunktion, wird auch eine Spielfunktion über ein lineares Netz erzeugt.

Die Erzeugung einer Spielfunktion durch Verwendung neuronaler Netze wird im nachfolgenden Oberkapitel 3.4 erläutert. Alle Spielfunktionsvarianten werden durch entsprechende Agenten erstellt, die sich ihrer eigenen Strategie bedienen und „nur“ den RL-Kerngedanken der Belohnung/Bestrafung am Spielende berücksichtigen.

So wurde mit Hilfe der Programmiersprache Java (jdk 1.6.0_03[13]) und der Entwicklungsumgebung Eclipse Europe 2007 ein Klassenkonzept entwickelt, in dem besagte Agenten die korrekte Spielfunktion zum Lösen des Spiels „Nimm-3“ erlernen können. Zusätzlich zu den Agenten werden Klassen benötigt, die das Spielumfeld und diverse Hilfsoptionen repräsentieren. Anfänglich wurde erwähnt, dass ein Reward bei Spielende nur durch die Umgebung verteilt werden kann (vergleichbar Kapitel 2.2). Daher ist die Nutzung einer Klasse für das Spielumfeld Voraussetzung. Die Hilfsklassen implementieren die textuelle Ausgabe zur Korrektheitsprüfung der Methoden (Klasse „Logger“) und die Generierung der Spielerfarben (Enumeration „E_Player“), die eventuell für eine spätere GUI[14]-Entwickung notwendig sind.

Der Agent, der die Erzeugung einer Spielfunktion über ein lineares Netz als Ziel hat, erzeugt eine Netzstruktur, die eine perfekte Spielcodierung von „Nimm-3“ darstellt. Dabei werden die Kantengewichtung zwischen Eingabe- und Ausgabeschicht so angelernt, dass das Netz bei Vorlage eines Spielzustands den bestmöglichen Zug errechnen und direkt ausführen kann.

Der Lernvorgang ähnelt dabei sehr der Entwicklung der tabellarischen Spielfunktion, die

13 Jdk 1.6.0_03 = Java Development Kit Version 1.6.0_03 (für diese Diplomarbeit wurde diese Version verwendet)
14 GUI = Graphical User Interface (deutsch: grafische Benutzer-Schnittstelle)

rückwärtig aufgebaut wird. Durch die Nutzung einzelner Zufallszüge werden alle theoretischen Zustände bewertet und daher alle möglichen Werten generiert, um das Spiel perfekt zu spielen.

Die aussagekräftigen Werte, die – ähnlich der Werte aus der Tabelle – die Gewinnwahrscheinlichkeit des anziehenden Spielers repräsentieren, sind hier die Kantengewichtungen des linearen Netzes. Sie geben zwar keinen, auf 100% skalierten Wahrscheinlichkeitswert an, machen über die Höhe der Werte aber dennoch deutlich, welche Züge gut und welche schlecht sind, um das Spiel möglichst zu gewinnen.

Betrachtet man ein erstes Spiel, welches der Agent spielt – der ein lineares Netz als Grundlage zur Erzeugung der Spielfunktion nutzt – fällt auf, dass eine Wertigkeit eines Abschlusszugs erlernt worden ist (siehe Abbildung 8).

Spiel 1		
weight[0]	=	00,07518
weight[1]	=	-24,19192
weight[2]	=	00,55323
weight[3]	=	00,74135
weight[4]	=	00,50214
weight[5]	=	00,83767
weight[6]	=	00,56980
weight[7]	=	00,11385
weight[8]	=	-00,03481
weight[9]	=	00,56485
weight[10]	=	00,21056
weight[11]	=	00,65404

Abbildung 8: Kantengewichtung nach einem Spiel

Das liegt an der schon beschriebenen Technik, die RL nutzt. Es wird rückwärts erlernt, welches Resultat eines Zugs gut beziehungsweise schlecht für den Sieg des Anziehenden ist. Nach dem ersten Trainingsspiel wird deutlich, dass es einen After-State „ein restlicher Spielstein auf dem Spielbrett“ gegeben haben muss, den der anziehende Spieler hinterlassen hat. Es sei bemerkt, dass die Bezeichnungen „weight[0]“ bis „weight[11]“ in den Abbildungen dieses Kapitels für die mögliche Anzahl an restlichen Spielsteinen auf dem Spielbrett stehen.

Wenn man nun eine „1“ für den anziehenden Spieler an das, der zuvor angepassten Kante entsprechende Eingabeneuron anlegt, ergibt sich aus der Berechnung des Output $1 \cdot (-24{,}19192) = -24{,}19192$, was eine hohe Wahrscheinlichkeit angibt, dass das Spiel vom Anziehenden wohl verloren wird.

Spiel 5		
weight[0]	=	00,07518
weight[1]	=	-24,19192
weight[2]	=	-22,66300
weight[3]	=	00,67312
weight[4]	=	00,50214
weight[5]	=	00,68400
weight[6]	=	00,56980
weight[7]	=	00,11385
weight[8]	=	-00,07788
weight[9]	=	00,43959
weight[10]	=	00,21056
weight[11]	=	00,46789

Abbildung 9: Kantengewichtung nach fünf Spielen

Schaut man sich die Entwicklung der Kantengewichtung nach fünf Trainingsspielen an (siehe Abbildung 9), stellt man fest, dass bereits zwei abschließende Spielzüge bewertet wurden. Hier wurde also in einem der letzten Trainingspiele einmal das laufende Spiel ausgehend vom After-State „zwei restliche Spielsteine auf dem Spielbrett“ beendet, und zwar durch den Nachziehenden, was durch den negativen Wert der entsprechenden Kante „weight[2]“ deutlich wird.

Die anderen Kantengewichtungen sind entweder zufällig

initiierte Startwerte, die bei Erzeugung des linearen Netzes erstellt wurden, oder sie resultieren aus einem kleinen Lernschritt, der im Laufe eines Spiels gemacht wurde. Das heißt, es gibt sogar bei Zwischenschritten durchaus einen Lerneffekt.

Nach jedem nicht zufälligen Zug wird bekanntlich ein Fehlerwert berechnet, der sich aus der aktuellen After-State-Bewertung minus der vorangegangenen After-State-Bewertung plus einem möglichen Reward ergibt. Bei der Auswertung dieses Fehlerwertes ergeben sich nun geringe Werte, die im Lernschritt Auswirkungen auf die Kantengewichtung haben und somit diese geringen Kantenanpassungen begründen. Es ergeben sich nur geringe Fehlerwerte, weil ja im Laufe eines Spiels noch kein (hoher) Reward verteilt werden kann; somit ist nur die Differenz der besagten After-State-Bewertungen ausschlaggebend für die Höhe des Fehlers.

Diese „Zwischen-Lernschritte“ sind aber lange nicht so präzise, dass die Kantengewichtungen die Spielfunktion korrekt abbilden, weil sie nicht direkt durch einen hohen (oder niedrigen) Reward beeinflusst worden sind (vergleichbar dem Rückwärts-Lernen durch RL).

Betrachtet man im Weiteren die Entwicklung der Kantenanpassung durch die RL-Strategie, wird deutlich, dass bereits nach 68 Spielen eine ausreichend eindeutige Spielfunktion abgebildet wird, die das Spiel perfekt codiert.

Dies wird in Abbildung 10 anhand der Werte für weight[4] und weight[8] deutlich. Dies sind bekanntlich die After-States, die der anziehende Spieler erzeugen muss, um perfekt zu spielen. Die Bewertung dieser Kanten muss also im Vergleich zu den „Nachbar“-After-State – das sind die After-States, die er bei möglichen Zügen auch erreichen könnte – entsprechend größer sein.

Spiel 68		
weight[0]	=	00,07518
weight[1]	=	-24,19192
weight[2]	=	-22,66300
weight[3]	=	-21,73646
weight[4]	=	01,80643
weight[5]	=	-00,44537
weight[6]	=	-00,46483
weight[7]	=	-00,69928
weight[8]	=	00,03875
weight[9]	=	00,02900
weight[10]	=	00,03326
weight[11]	=	00,00224

Abbildung 10: Kantengewichtung nach 68 Spielen

Liegt beispielsweise ein After-State des Nachziehenden von „elf restlichen Spielsteinen auf dem Spielbrett“ vor (nachfolgenden nur noch durch die Zahl restlicher Spielsteine auf dem Spielbrett bezeichnet), muss der Wert des After-States „8“ so groß sein, dass der Anziehende sich nicht für die After-States „10“ oder „9“ entscheidet. Ebenso muss bei vorliegendem After-State von „9“ – nach dem Zug des Nachziehenden – der After-State „8“ für den Anziehenden höher bewertet werden, als die After-States „7“ oder „6“. Entsprechendes gilt für den Wert des After-States „4“, der ja auch vom anziehenden Spieler erreicht werden sollte.

Aus Abbildung 10 erkennt man, dass dieses Szenario vorliegt, das heißt, dass alle After-States perfekt codiert sind. Die Höhe der einzelnen Bewertungen ist schließlich nicht ausschlaggebend; einzig die Tatsache, dass After-States erreicht werden sollen, die einem Vielfachen der Zahl „4“ entsprechen, ist relevant und wird durch obige Bewertung erreicht.

Vergleicht man die Bewertung der Kanten „weight[0]“ und „weight[12]“ wird auch erkannbar, dass der Bias-Wert ebenso wie der After-State „keine restlichen Spielsteine auf dem Spielbrett“ nie gelernt und damit in seiner Bewertung verändert wird.

Grundsätzlich lässt sich sagen, dass auch bereits vorhandene Bewertungen einer Kante – und damit auch eines After-States – nicht unveränderlich bleiben. Es besteht durchaus die Möglichkeit, dass der Wert als solches nochmals verändert wird. Wenn man einen Schritt zurückgeht und das folgende Beispiel betrachtet, welches auf der Situation, die man aus Abbildung 9 erkennen kann aufbaut, wird deutlich was hiermit gemeint ist.

Entsteht eine Situation, die im Laufe des Trainings schon einmal bewertet wurde, kommt es auf die Werte des Fehlers und des bisherigen Funktionswert dieser Situation an, ob sich die Bewertung des After-States nochmals verändert.

So kann es beispielsweise durchaus sein, dass – innerhalb eines Spieles – der nachziehende Spieler den After-State „drei restliche Spielsteine auf dem Spielbrett“ hinterlässt. Je nach Bewertung der bisherigen Kanten – vergleichbar zu Abbildung 9 – entfernt der Anziehende nun drei Spielsteine vom Spielbrett und gewinnt das Spiel. Die Bewertung des vorgefundenen After-States „drei restliche Spielsteine[..]“ fällt positiv aus, da ein Reward von (beispielsweise) 100 verteilt wird. Man weiß aber, das es generell schlecht ist, wenn dieser besagte After-State positiv bewertet wird, da der Anziehende so im Folgespiel wahrscheinlich versuchen würde, eben diesen hochbewerteten After-State zu erreichen. Dies ist fatal für ihn, da der Nachziehende auch den für sich besten Zug wählen wird, was das Spiel beendet und den Nachziehenden zum Sieger macht. Sobald der anziehende Spieler in einem nachfolgenden Spiel versucht, den gerade hoch bewerteten After-State zu erzeugen, verliert er das Spiel, weil der Nachziehende nun das Spielbrett leert und somit gewinnt. Dieser Schritt hat aber wieder zur Folge, dass der After-State „drei restliche Spielsteine[..]“ negativ bewertet wird, da schließlich ein Reward von (beispielsweise) -100 verteilt wird. Die Kantengewichtung wird also verändert.

Häufig benötigt man mehrere Änderungen einer einzelnen Kantengewichtung, bis der Wert brauchbar für die Spielfunktion wird und diese korrekte Ergebnisse für weitere Spiele liefert.

Der Lernvorgang eines linearen Netzes gleicht schließlich dem Finden eines globalen Maximums innerhalb einer mehrdimensionalen Funktion. Selbst wenn man sich mit den

Kantenanpassungen auf dem richtigen Weg zum globalen Maximum befindet, heißt dies nicht, dass nicht noch bessere Werte für die Kantengewichtungen existieren, die die Netzauswertung noch deutlicher und korrekter machen würden. In der Anwendung auf strategische Spiele bedeutet das, dass eine erzeugte Spielfunktion über ein lineares Netz durchaus eine gute oder sogar perfekte Spielcodierung sein kann; es kann aber ebenso sein, dass die Werte dieser Spielfunktion noch deutlich sein könnten, da man sich mit der Suche nach der besten Netzperformance noch nicht beim globalen Maximum befindet.

Am Beispiel des zuvor erläuterten Lernprozess eines linearen Netzes ist dieser Effekt auch festzustellen. Vergleicht man die Werte der Kantengewichtungen des Netzes zur Erzeugung der Spielfunktion von „Nimm-3“ nach 100 Spielen aus Abbildung 11 mit denen nach 68 Spielen – wo bereits eine perfekte Spielcodierung vorlag –, erkennt man, dass sich die Werte der Kanten noch deutlich verändert haben. Allerdings bleibt die perfekte Codierung erhalten und wird nur in den Abständen der einzelnen Werte größer und eindeutiger.

Spiel 100		
weight[0]	=	00,07518
weight[1]	=	-24,19192
weight[2]	=	-22,66300
weight[3]	=	-21,73646
weight[4]	=	02,05879
weight[5]	=	-00,84478
weight[6]	=	-00,81788
weight[7]	=	-01,08639
weight[8]	=	00,39072
weight[9]	=	-00,01941
weight[10]	=	-00,05771
weight[11]	=	-00,12145

Abbildung 11: Kantengewichtung nach 100 Spielen

Man erkennt, dass außer den Werten von „weight[4]“ und „weight[8]“ alle anderen Werte negativ sind (mit Ausnahme der unberücksichtigten und nicht gelernten Kanten „weight[0]“ und „weight[12]“). Dies verdeutlicht die hohe Wahrscheinlichkeit für den Anziehenden, ein Spiel zu verlieren, wenn er einen entsprechend negativen After-State erzeugen würde. Im Gegensatz dazu ergeben die After-States „4“ und „8“ positive Wahrscheinlichkeiten – wenn auch geringe – das Spiel als Anziehender zu Gewinnen; dies sind schließlich auch die perfekten Spielsituationen, die ein anziehender Spieler erzeugen sollte.

Die Höhe der Wahrscheinlichkeiten ist inzwischen nicht mehr auf 100% skalierbar, das liegt an der Umwandlung der Werte in bestimmte Wertebereiche zur Verwendbarkeit innerhalb eines linearen Netzes. Dennoch sagen die Werte etwas über die unterschiedliche Bedeutung der After-States zum Gewinn des Spiels aus und codieren daher die perfekte Spielfunktion.

3.5 Spielfunktion über neuronales Netz

3.5.1 Unterschiede zwischen linearem und neuronalem Netz

Bei Spielen wie „Nimm-3“ ist die Nutzung eines neuronalen Netzes als Grundlage der zu generierenden Spielfunktion nicht unbedingt notwendig. Dies liegt an der einfachen Codierungsvariante mit deren Hilfe man das Spiel perfekt spielen kann.

Ein weiterer Grund, warum eine lineare Spielfunktion hier ausreicht, ist die Abhängigkeit von wenigen linearen Parametern: Die restliche Spielfeldgröße (After-State) als ganzzahlig positiver Wert und der entsprechende Spieler, der diesen After-State hinterlassen hat, genügen als Information für die Spielfunktion.

Aber man kann sich leicht vorstellen, dass bei komplexeren Spielen – wie „4-Gewinnt“ – zur Erzeugung einer guten Spielfunktion deutlich mehr Informationen relevant sind.

Bei diesem strategischen Brettspiel sind beispielsweise die belegten Positionen innerhalb des Spielbrettes interessant, ebenso spielt natürlich der Spieler, der an der Reihe ist, eine Rolle. Aber im besonderen Maße ist die Kombination mehrerer, eigener Spielsteine, die ein ausbaufähiges Muster bilden, relevant. Da es bei „4-Gewinnt“ bekanntlich um die Erzeugung von vier zusammenhängenden Spielsteinen („Vierer“) geht, spielen zum Beispiel drei aneinanderhängende Spielsteine nur dann eine wichtige Rolle, wenn diese drei noch zu besagtem Vierer ausgebaut werden können (siehe Kapitel 4.1); geht dies nicht, ist das Muster wertlos für den weiteren Spielverlauf.

Man muss also eine klare und eindeutige Unterscheidung der Muster finden sowie vorab klären, welche Muster überhaupt wesentliche, spielentscheidende Informationen tragen. Nur im günstigen Fall lassen sich diese Informationen zu linearen Parametern formen, die letztlich eine lineare Spielfunktion bilden. Häufig spielen aber auch mehrdimensionale Parameter eine Rolle. Der Einfluss bestimmter Größen (oder Fakten) ist relevanter als andere – diese Relevanz ist nicht nur durch einen faktoriellen, sondern durch einen mehrdimensionalen Unterschied ausdrückbar.

Falls solch ein Fall eintritt – und das ist bei Brettspielen nicht selten – benötigt man zur guten Spielcodierung eine mehrdimensionale Funktion. Aufgrund dieser Notwendigkeit bedient man sich der Technik neuronaler Netze, die durch verschiedene Strukturierungen ein Abbild einer mehrdimensionaler Funktionen sein können.

Für das vorliegende Beispiel „Nimm-3“ bedeutet das kaum große Umstellungen. Im Vergleich zur Erzeugung einer linearen Funktion durch ein lineares Netz, wird bei einer mehrdimensionalen Funktion einfach ein neuronales Netz mit verdeckten Schichten angewendet. Das bedeutet, dass die Techniken und Berechnungsgrundlagen dieselben bleiben und einfach eine oder mehrere verdeckte Schichten hinzugefügt werden. Somit lassen sich auch nicht-separable Probleme lösen, die im Grunde mehrdimensionale Abhängigkeiten der Parameter beschreiben. Als Ergebnis sollte eine vergleichbar starke Spielfunktion vorliegen, wie es bei der Verwendung eines linearen Netzes der Fall ist.

3.5.2 Anwendung des neuronalen Netzes

Die Nutzung eines neuronalen Netzes als Grundlage einer komplexen Spielfunktion hat den entscheidenden Vorteil, dass komplexere Zusammenhänge zwischen einzelnen Parametern schneller entdeckt und verarbeitet werden können als bei linearen Netzen. Dort sind Abhängigkeit zwischen den einzelnen Eingabeparametern – so fern diese separierbare Probleme darstellen – schwerer zu erkennen und vor allem zu erlernen.

Die Codierung der Spielfunktion , auf die ein neuronales Netz angewandt wird, variiert etwas und weicht somit von der ursprünglichen Codierung ab. Man codiert hier nun nicht mehr nur die möglichen After-States als Eingabeneuronen, sondern verwendet ein weiteres Neuron, welches den Spieler repräsentieren soll, der den codierten After-State hinterlassen hat. Das heißt, dass als Werte der einzelnen After-State-Eingabeneuronen nur die „0“ und die „1“ zugelassen sind - „0“ für nicht aktives Neuron, „1“ für aktiviertes After-State-Neuron. Das Spielerneuron kann aber die Werte „-1“ und „1“ annehmen, da dieses den Spieler repräsentiert und daher auch die „-1“ für den Nachziehenden erlaubt sein muss. Eine anliegende „1“ sagt also aus, dass der After-State vom Anziehenden hinterlassen wurde. Dieses Neuron muss zu jedem Zeitpunkt aktiviert sein, da schließlich jeder Zug von einem Spieler ausgeführt worden ist.

Die konzeptionelle Unterscheidung tritt nun darin auf, dass ein Zusammenhang zwischen After-State-Neuronen und Spieler-Neuron hergestellt werden muss. Das Netz muss anhand von Trainingsdaten herausfinden, welche Relevanz das Spielerneuron für das Ergebnis besitzt und wie entscheidend der hinterlassende After-State ist.

Um solche Zusammenhänge zu erlernen, ist es hilfreich, das Netz um verdeckte Schichten zu erweitern. Ein Hinzufügen einer verdeckten Schicht ermöglicht das Erkennen und Erlernen nicht-separabler Zusammenhänge. Die Netzleistung steigert sich und es werden schneller Lernerfolge erzielt. Der Nachteil dieser Netzstruktur ist, dass deutlich mehr Trainingsdaten verwendet werden müssen, um gute Lernprozesse zu erzielen und letztlich eine gute Netzperformance zu generieren. Lernen über Kantenanpassung muss jetzt schließlich auch die Kanten zwischen verdeckter und Ausgabeschicht optimieren, das verzögert die Anpassung der Kanten zwischen Eingabe-

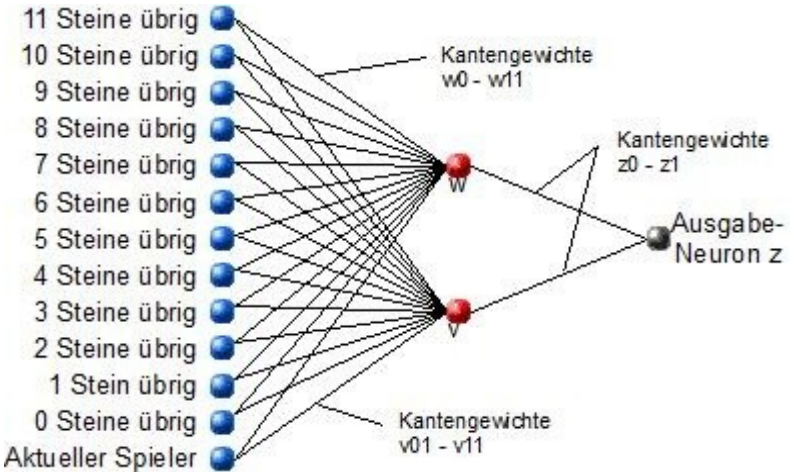


Abbildung 12: Struktur des neuronalen Netz

und verdeckter Schicht, was aber das eigentliche Ziel darstellt.

Das benutzte neuronale Netz ähnelt dem aus Abbildung 12. Durch die mehrfachen Verbindungen zu den Hidden-neuronen (das sind die Neuronen der verdeckten Schicht) können unterschied-liche Zusammenhänge der Eingabeneuronen verschie-dene Werte der Hidden-neuronen erzeugen. Somit hängt die Ausgabe nicht mehr ausschließlich von den Eingabewerten ab, sondern wird durch – möglicherweise – unterschiedliche Werte der Hiddenneuronen beeinflusst.

Die Anwendung dieser Netzstruktur ähnelt nun sehr dem Ablauf zur Erzeugung einer linearen Spielfunktion unter Benutzung eines linearen Netzes. Durch viele Trainingsspiele sollte das neuronale Netz in der Lage sein, Kantenanpassungen so durchzuführen, dass die resultierende Netzstruktur eine möglichst optimal codierte Spielfunktion darstellt.

In Abbildung 13 (auf der nächsten Seite) erkennt man, dass mittels einer neuronalen Netzstruktur die Kantengewichtungen der Eingabeneuronen nach 50000 Spielen ähnlich eindeutig erscheinen, wie bereits in Kapitel 3.3 bei der Verwendung eines linearen Netzes zu erkennen ist. Hier wird auch deutlich, dass es scheinbar bei einer so einfachen Spielcodierung nicht relevant ist, mehrere Hiddenneuronen zu erzeugen, denn die Kantengewichtungen zu beiden Hiddenneuronen sind gleich.

Im Folgenden wird detailliert auf die Erzeugung dieses Ergebnisses eingegangen. Um die Übersicht zu erhalten wird nachfolgend Bezug auf den Anhang A.2 genommen, der die Kantengewichtungen eines neuronalen Netzes nach den ersten drei Trainingsspielen beinhaltet.

Der Grund, warum die Erzeugung einer vergleichbaren Kantenbewertung beim neuronalen Netz so lange dauert, ist die Verzögerung des Lerneffektes. Nach dem ersten der Trainingsspiele erkennt man keinen direkten Lernfortschritt. Vergleicht man die Kantengewichtungen mit denen des linearen Netzes, fällt auf, dass keine Verbindungskante zwischen Eingabe- und verdeckter Schicht verändert worden ist. Lediglich die Verbindungen zwischen verdeckter und Ausgabe-Schicht haben sich in ihrer Bewertung verändert. Jeder Lernschritt wirkt sich zunächst auf die Hiddenneuronen aus und bringt daher zunächst eine Kantenanpassung letzterer Verbindung mit sich. Erst beim zweiten Trainingsspiel wird sich der Fehler auch auf die Eingabekanten auswirken und deren Bewertungen verändern, was man an der grafischen Gegenüberstellung der Kantengewichtungen im Anhang A.2 auch erkennen kann.

Weiter ist der geringe Wertebereich der Transferfunktion ausschlaggebend für den geringen Lernfortschritt nach wenigen Spielen. Dadurch, dass als Transferfunktion die Sigmoidfunkton gewählt wurde, befinden sich die Ausgabewerte immer im Intervall zwischen „0“ und „1“. Durch diesen Transfer wird ein eigentlich hoher Ausgabewert häufig nahe am Funktionsmaximum (von „1“) liegen – und keinen besonders großen Unterschied mehr zu wesentlich höheren Ausgabewerten besitzen. Transferiert man beispielsweise die Zahl 10 durch die Transferfunktion, erhält man einen neuen Wert von 0,9999546021312976. Bei einem zu transferierenden Wert von 20 – der deutlich höher ist – erhält man 0,9999999979388463. Die Differenz dieser beiden neuen Werte beträgt dann nur noch -0,0000453958075487. Zuvor (also ohne Wertetransfer) betrug die Differenz 10.

Spiel 50.000		
weights_in[0][0]	=	0,00000
weights_in[1][0]	=	-0,02959
weights_in[2][0]	=	0,00000
weights_in[3][0]	=	0,00091
weights_in[4][0]	=	2,20461
weights_in[5][0]	=	-0,08104
weights_in[6][0]	=	0,09258
weights_in[7][0]	=	-0,52490
weights_in[8][0]	=	2,20239
weights_in[9][0]	=	-0,52185
weights_in[10][0]	=	1,27157
weights_in[11][0]	=	-0,55790
weights_in[12][0]	=	-2,16054
weights_in[0][1]	=	0,00000
weights_in[1][1]	=	-0,02959
weights_in[2][1]	=	0,00000
weights_in[3][1]	=	0,00091
weights_in[4][1]	=	2,20461
weights_in[5][1]	=	-0,08104
weights_in[6][1]	=	0,09258
weights_in[7][1]	=	-0,52490
weights_in[8][1]	=	2,20239
weights_in[9][1]	=	-0,52185
weights_in[10][1]	=	1,27157
weights_in[11][1]	=	-0,55790
weights_in[12][1]	=	-2,16054
weights_out[0][0]	=	6,38574
weights_out[1][0]	=	6,38574

Abbildung 13: Kantengewichtungen nach 50.000 Spielen

Dies bedeutet für die Gradientenabstiegsrechnung aber, dass ein zu erlernender neuer Kantenwert zusätzlich noch durch einen sehr geringen Faktor (unter „1“) verändert wird. Somit wird eine eigentlich starke Kantenanpassung im Lernschritt durch einen geringen Gradientenabstiegswert minimiert, was eine Verlangsamung des Lernens bewirkt.

Dennoch lässt sich über die Kantenanpassungen des neuronalen Netzes eine geeigneten Spielfunktion erzeugen, die das Spiel so codiert bewertet, dass man in der Lage ist, es perfekt zu spielen. Im Anhang A.2 sieht man die Kantengewichtungen des neuronalen Netzes. Man erkennt, dass bereits nach wenigen Spielen ein kleiner Lerneffekt eingetreten ist.

Wie bereits erwähnt, werden zunächst die Kantengewichtungen zwischen Hidden- und Ausgabeschicht verändert. Nach zwei Trainingsspielen sind aber schon die Eingabekanten bewertet, die die Situation beschreiben, aus der der Sieg erzielt wurde – dieser Lerneffekt lässt sich mit dem ersten Lernschritt beim linearen Netz oder der tabellarischen Spielfunktion vergleichen.

Aus dem dritten Spiel erkennt man keinen großen Lerneffekt. Zwar wurden viele Kanten bewertet, diese Bewertungen sind aber viel zu gering, um langfristig eine korrekte Spielzugentscheidung zu treffen. Zuvor wurde schon darauf hingewiesen, dass viele Trainingsspiele häufig zur Eindeutigkeit der Spielfunktion beitragen. Nach drei Spielen deuten

die bisherigen Kantengewichtungen aber kaum Eindeutigkeit im Ergebnis an. Die vergleichsweise stark bewerteten Kanten sind die des After-States, aus dem der Sieg erzielt wurde (weights_in[1][x]) und die abgehende Kanten des Spielerneurons (weights_in[12][x]). Scheinbar ist die Kante ausgehend vom Spielerneuron von Bedeutung bei der Erzeugung einer Spielfunktion – dies ist ja auch verständlich, schließlich kommt es immer auf den Spieler an, der einen After-State hinterlässt, wenn es um die Bewertung der Situation geht.

Ebenso scheint während der ersten drei Trainingsspiele einmal der Anziehende und einmal der Nachziehende das Spiel gewonnen zu haben. Beide Male scheint der Sieg aus dem After-State „ein restlicher Spielstein auf dem Spielfeld“ erzielt worden zu sein. Man erkennt dies aus der Kantengewichtung der entsprechenden Kante, die einmal positiv und einmal negativ ist. Es müssen also Fehlerwerte verschiedener Vorzeichen diese Kantenanpassung beeinflusst haben.

Aus der exemplarischen Kantenübersicht der ersten wenigen Trainingsspiele wird ebenfalls deutlich, dass der Lerneffekt geringer ausfällt als beim linearen Netz. Die Bewertungen sind viel geringer und tragen damit zur Verlangsamung des Lerneffekts bei.

Insgesamt lässt sich festhalten, dass auch bei der Verwendung eines neuronalen Netzes das Erlernen einer geeigneten Spielfunktion möglich ist. Die Lerndauer ist hier zwar viel länger, es müssen aber schließlich auch mehr Zusammenhänge zwischen Eingabeneuronen erlernt werden als beim linearen Netz, bei dem die Spielercodierung direkt am passenden After-State mittels Wert des Neurons festgehalten worden ist. Da es hier ein eigenes Neuron für die Spielercodierung gibt, muss der Zusammenhang zwischen diesem Neuron und entsprechenden After-States erst noch erlernt werden.

Dies wird auch bei erneuter Betrachtung der Abbildung 13 deutlich, an der man erkennt, dass das Spielerneuron (weights_in[12][x]) eine ähnlich starke Bewertung erhält, wie die spielrelevanten After-States „4“ und „8“ - besagte Vielfache der Zahl „4“, die einem Spieler einen Sieg ermöglichen, wenn er sie als After-States wählt. Das bedeutet, dass es auf drei entscheidende Parameter beziehungsweise Situationen im Spiel „Nimm-3“ ankommt: Möglichst acht und später vier Spielsteine auf dem Spielbrett liegen lassen und möglichst anziehender Spieler sein, da man als dieser die erste Chance auf einen spielrelevanten After-State hat.

3.6 Analyse und Ergebnisse zur Lernfähigkeit von Nimm-3

Das erste Ziel dieses Arbeit wurde als Test formuliert. Es betrachtet die Frage, ob man mittels RL das Spiel „Nimm-3“ erlernen kann. Die zurückliegenden Kapitel setzen sich mit dieser Frage auseinander und analysieren theoretisch und praktisch die Lösbarkeit dieses Problems.

Die tabellarische (und offensichtlichste) Lösung bietet eine gute Möglichkeit die Spielfunktion als

perfekte Spielcodierung zu erzeugen. Aufgrund der geringen Anzahl an Spielsituationen ist es möglich, diese in tabellarischer Form abzuspeichern und mittels RL-Strategie zu bewerten. Wurden ausreichend Trainingsspiele getätigt, liegt eine korrekte Bewertung aller After-States für beide Spieler vor, mit Hilfe derer man das Spiel perfekt spielen kann. Darüber hinaus geben die Bewertungen der einzelnen Spielsituationen auch die prozentuale Wahrscheinlichkeit an, dass der Anziehende das Spiel bei perfekter Spielweise aus der entsprechenden Situation heraus gewinnt. Hier ist das Konzept des RL vollständig gelungen, da eine starke, ja perfekte Spielfunktion erzeugt worden ist. Es ist also grundsätzlich möglich, ohne Wissen über Taktiken und Spielstrategien ein Spiel mit Unterstützung von RL zu erlernen. Die einzelne Belohnung oder Bestrafung am Ende eines einzelnen Spiels erfolgt, reicht aus, um rückwärtig eine gute oder sogar perfekte Bewertung aller möglichen Spielsituationen zu erzeugen.

Der zweite Schritt wird durch die Nutzung eines linearen Netzes beschrieben. Hier gibt es nun keine Möglichkeit mehr, tabellarisch Spielsituationen abzuspeichern. Man generiert vielmehr direkt eine (lineare) Spielfunktion, die über die Auswertung bestimmter Parameter in der Lage ist, die richtigen Schlussfolgerungen zu ziehen und somit das Spiel zu spielen. Hier wurden dem Netz als Eingabeparameter die verschiedenen, möglichen After-States zugewiesen. Derjenige Spieler, der diesen After-State hinterlässt, wurde über die Wertebelegung an den entsprechenden Eingabeparameter gesteuert. Somit verringert sich im Vergleich zur tabellarischen Spielfunktion die Anzahl der Parameter auf die Hälfte, da die After-States nur noch einmal codiert werden und der entsprechende Spieler über die Wertebelegung des Aftes-States codiert wird.

Auch das lineare Netz kann mittels der RL-Strategie eine gute Codierung des Spiels erlernen und realisiert daher eine geeignete Spielfunktion. Die Werte, die über die Spielfunktion berechnet werden können, geben zwar nicht mehr die prozentuale (auf 100% skalierte) Siegwahrscheinlichkeit des Anziehenden an, genügen in ihren Ausprägungen aber, um richtige Züge zu ermöglichen und das Spiel daher perfekt spielen zu können. Letztlich ist auch hier die Nutzung von RL gelungen, da man mittels einer linearen Funktion mit weniger Informationen als bei einer tabellarischen Lösung das Spiel ebenso erfolgreich erlernen und anschließend perfekt spielen kann.

Schließlich wurde im vorausgegangenen Kapitel ein neuronales Netz zur Anwendung gebracht. Hier musste zusätzlich zu den Informationen, die beim linearen Netz benötigt werden, ein sogenanntes Spielerneuron bei der Eingabeschicht ergänzt werden. Man könnte sich natürlich auch beim neuronalen Netz mit den Informationen des linearen Netzes begnügen, diese würden auch ausreichen, aber der Unterschied der Lerngeschwindigkeit und des Nutzens

neuronaler Netze würde letztlich nicht besonders deutlich. Dennoch wäre auch ein neuronales Netz natürlich in der Lage, die Codierung, die beim linearen Netz gewählt wurde, zu erlernen. Um aber eine weitere Codierung zu testen, wurde der Eingabeschicht beim neuronalen Netz noch ein Neuron hinzugefügt. Das Ziel, das hier beabsichtigt war, sollte über die Lerngeschwindigkeit und Performance des Netzes deutlich werden: Durch die Trennung von After-States und Spieler, der diese hinterlässt, muss das Netz größere Zusammenhänge erlernen, als es zuvor beim linearen Netz der Fall war. Mehr noch, beim linearen Netz war nie mehr als ein Eingabeneuron aktiviert, beim neuronalen Netz sind nun immer zwei Neuronen aktiviert, somit ist eine Abhängigkeit der Neuronen untereinander vorhanden, die erlernt werden muss.

Das spiegeln auch die Ergebnisse des Lernprozesses wieder, der beim neuronalen Netz mit Abstand am langwierigsten und aufwendigsten ist – im Folgenden werden die Ergebnisse, die in den Abbildungen 11 und 13 gezeigt werden, angesprochen.

Es mussten um die 50.000 Trainingsspiele erzeugt werden, aus denen das neuronale Netz letztlich die Spielfunktion erlernen konnte. Auch die Wertebelegung der einzelnen abgehenden Kanten aller Eingabeneuronen verdeutlicht das Problem der vorhandenen Abhängigkeiten. Vergleicht man die Kantenbewertungen des neuronalen, mit denen des linearen Netzes, wird deutlich, dass erstere in ihrer Ausprägung und Eindeutigkeit lange nicht so genau wie beim linearen Netz. Das bedeutet, dass beim neuronalen Netz vor allem die Wichtigkeit der After-States „4“ und „8“ über die entsprechenden Kantenbewertungen deutlich wird, während beim linearen Netz vor allem die spielbeendenden After-States hoch bewertet wurden. Ebenso fällt beim neuronalen Netz die recht hohe Kantenbewertung des Spielerneurons auf.

Vor allem an der Entwicklung der Kantenbewertungen beim linearen Netz wird deutlich, welchen Einfluss eine große Zahl an Trainingsspielen auf die Spielfunktion hat. Je mehr Spiele hier gemacht werden, um so eindeutiger werden auch die Ausprägungen der entsprechenden „wichtigen“ Kanten zu den After-States „4“ und „8“. Im Gegensatz zum neuronalen Netz benötigt man beim linearen aber wesentlich weniger Spiele, um eine geeignete Spielfunktion zu generieren.

Aber auch ein neuronales Netz ermöglicht letztlich die Erzeugung einer guten Spielfunktion, mit Hilfe derer man das Spiel „Nimm-3“ perfekt spielen kann. Man benötigt allerdings schon für dieses – verhältnismäßig leichte und kleine – Brettspiel eine riesige Zahl an Trainingsdaten, die nur über komplette Spiele realisiert werden können. Das unterstützt die jetzige Einschätzung, dass die Trainingsdatenmenge bei komplexeren Spielen noch wesentlich größer werden muss, um zu aussagekräftigen Ergebnissen zu gelangen.

Hier könnte man aber – ähnlich wie bei dem Problem der Unbewertbarkeit aller theoretisch möglichen After-States eines Spiels – auf das Problem zu großer Datenbestände stoßen, die letztlich nicht in reeller Zeit zu bewältigen sind.

Somit lässt sich feststellen, dass eine gute Wahl der spielrelevanten Parameter günstiger für die Erlernbarkeit von Spielen ist als die Ausbreitung auf möglichst viele Parameter, bei denen untereinander Abhängigkeiten entstehen, die wieder schwer zu trainieren und zu erlernen sind.

Das muss daher auch das Ziel der folgenden Überlegungen sein, wenn es um die Erlernbarkeit des Brettspiels „4-Gewinnt“ mittels RL-Strategie gehen wird. Dieser Analyse wird ein neuronales Netz zugrunde gelegt, dass über diverse Parameterunterschiede innerhalb der Eingabeschicht verschiedene Performances erzeugen wird und letztlich sicher unterschiedlich gute Lösungen hervorbringt.

Die Wahl guter Parameter ist hierbei ebenso – wenn nicht noch deutlich mehr – ausschlaggebend, wie die Nutzung einer geeigneten Verarbeitungstechnik der Daten und die Wahl eines flexiblen Systems, was durch Lernprozesse verändert werden kann – wie es beim neuronalen Netz der Fall ist.

4 Erlernen des Spiels „4-Gewinnt“ mittels RL

4.1 Grundlagen zum Spiel „4-Gewinnt“

Das strategische Brettspiel „4-Gewinnt“ ist ein Spiel für zwei Spieler, die abwechselnd eine Stelle des senkrecht stehenden Spielbrettes mit ihrem Spielstein belegen. Die Spielsteine haben in der Regel die Farben gelb und rot. Das Spielbrett besteht aus sechs Zeilen und sieben Spalten, deren Schnittstellen die zu belegenden Spielfelder darstellen. Somit ergeben sich 42 Spielfelder, die entweder leer sind oder mit einem Spielstein belegt sein können. Das Spielbrett steht senkrecht, damit die eingeworfenen Spielsteine der Schwerkraft folgend in das unterste freie Spielfeld fallen. Man wirft dazu jeden Spielstein an der Oberkante einer Spalte ein. Somit wird der erste Spielstein definitiv in die unterste Zeile fallen. Danach kommt es auf die Spalte an, in die geworfen wird. Eine Spalte kann maximal sechs Spielsteine halten – das entspricht der Anzahl an Zeilen.

Ziel des Spieles ist es, eine Reihe aus vier eigenen, aufeinanderfolgenden Spielsteinen zu generieren. Diese Reihe muss ununterbrochen (mindestens) vier eigene Spielsteine enthalten und kann waagerecht, senkrecht oder diagonal liegen.

Abbildung 14 zeigt das Spielbrett mit Spielsteinen, die eine beliebige Spielsituation darstellen. Liegt ein sogenannter „Vierer“ vor – das sind eben beschriebene vier aufeinanderfolgende Spielsteine – hat der entsprechende Spieler das Spiel gewonnen. Ebenso kann aber auch ein Unendschieden entstehen, wenn alle Spielfelder mit Spielsteinen belegt sind, aber dennoch kein Spieler einen Vierer erzeugen konnte.

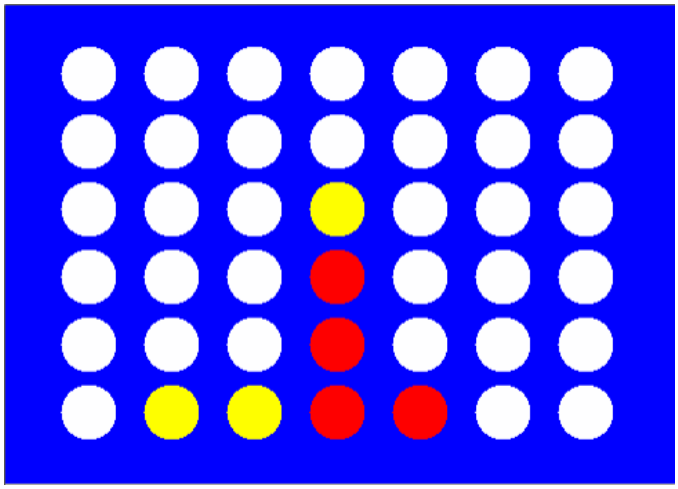


Abbildung 14: Spielbrett von 4-Gewinnt mit Spielsteinen

Die Strategie dieses Spieles kann ganz unterschiedlich aussehen. Manche Spieler spielen mit dem Ziel bestimmte Muster zu erzeugen, die gewisse Gewinnchancen darstellen. Andere nutzen die Erzeugung von Zwickmühlen, die dem strategisch Spielenden zwei Einwurfmöglichkeiten zum Sieg eröffnen, was der Gegner nicht verhindern kann. Wieder andere nutzen den sogenannten Zugzwang, der den Gegner zu einem Zug zwingt, der letztlich gewinnbringend für den strategisch Spielenden ist.

Ein reines Passivspiel, was nur die Chancen des Gegners verhindert, genügt bei diesem Spiel nicht automatisch zum eigenen Sieg, man muss sich immer selbst Gewinnchancen erarbeiten, indem man bestimmte Ziele – seien es Mustererzeugung, Zugzwang- oder Zwickmühlennutzung oder andere günstige Situationen – verfolgt. Wichtige Tatsache dabei ist, dass die unterschiedlichen Positionen des Spielfeldes auch unterschiedlich wertvoll bei der Generierung einer Gewinnchance sind. Die Spielfelder der mittleren Spalte bieten wesentlich mehr Ausbaumöglichkeiten zu Vierern als die Randspalten. Die Abbildungen 15 und 16 verdeutlichen die unterschiedliche Wertigkeit von Spielfeldern.

Wirft man beispielsweise in die äußerste, rechte Spalte in das unterste Spielfeld, trägt dieses Feld selber nur zu drei verschiedenen Möglichkeiten bei, einen Vierer zu erzeugen. Der Spielstein in diesem Feld kann Teil eines waagerechten eines senkrechten und eines diagonalen Vierers sein. Dagegen kann das unterste Feld der mittleren Spalte Anteil an sieben verschiedenen Vierern haben. Neben dem senkrechten Vierer und den beiden rechts und links verlagerten diagonalen Vierern, kann das Feld Anteil an vier verschiedenen waagerechten Vierern haben – sie sind innerhalb der Abbildung 16 farblich eingekreist.

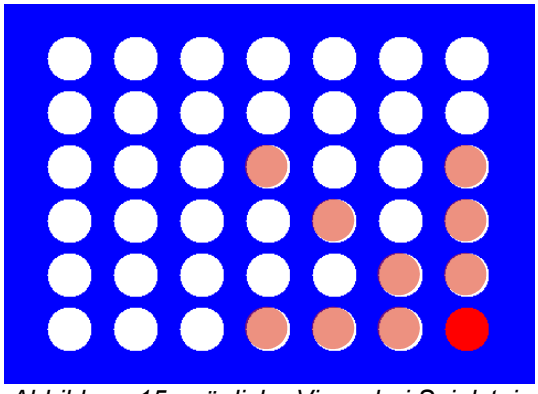


Abbildung 15: mögliche Vierer bei Spielstein unten rechts

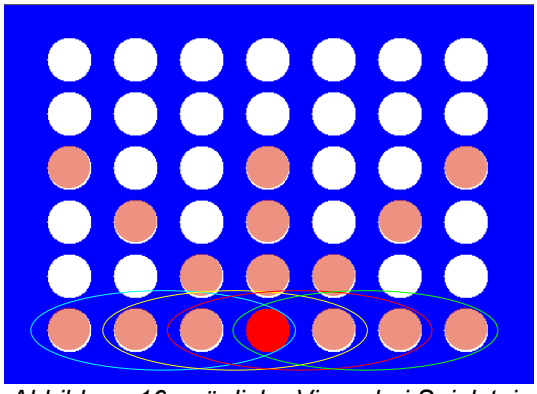


Abbildung 16: mögliche Vierer bei Spielstein unten mittig

Diese einfach festzustellende Tatsache entspricht der ersten Möglichkeit, verschiedene Parameter einer Spielfunktion zu erzeugen. Ohne einen Blick auf Muster, spieltaktische Überlegungen oder sonstige Feststellungen zu werfen, genügen die Informationen über die Belegung der einzelnen Spielfelder und deren Anteile an möglichen Vierern, um Parameter einer Spielfunktion zu erhalten. Ob diese Informationen zur Erzeugung einer ausreichend starken Spielfunktion genügen oder ob man weitere Fakten über strategische Spielelemente sammeln muss, kann erst nach Training und Auswertung einer solchen Spielfunktion festgestellt werden.

4.2 Softwareseitige Entwicklung eines Computergegners

4.2.1 Grundlagen zur Klassenaufteilung der Spielumgebung

Zur Entwicklung eines neuronal lernenden Spielagenten wird selbstverständlich ein Programmierumfeld benötigt, das die Generierung und das Testen notwendiger Methoden, die der Agent können soll, möglich macht. Hier wird – wie schon zuvor bei der Entwicklung von Methoden für Agenten, die Nimm-3 lernen und spielen sollen – Java als Programmiersprache und Eclipse Europe als Entwicklungsumgebung genutzt. Mit der Wahl von objektorientierter Programmierung stellt man sicher, dass die erstellten Klassen Objekte repräsentieren können, die auch mehrfach eingesetzt werden können. Man kann entwickelte Methoden daher flexibler und dynamischer Nutzen.

Da das Spiel 4-Gewinnt genauer analysiert und verwendet wird, als es bei Nimm-3 notwendig war, benötigt man auch eine feinere und strukturiertere Form der Klassenaufteilung, was letztlich eine verbesserte Übersicht ermöglicht. Hierzu wurden Klassen innerhalb verschiedener Pakete erstellt, die thematisch gegliedert werden können. Es gibt daher zurzeit insgesamt vier verwendete Pakete, die unterschiedliche Aufgaben darstellen.

Das Paket „gameagents“ enthält alle Agenten, die zu Erlernen, Spielen und Testen von 4-Gewinnt benötigt werden. Neben dem wichtigsten und zentralen RL_NNAgenten (der das neuronale Lernen des Spiels ermöglicht) gibt es einen RandomAgenten, der nur zufällige Zugentscheidungen trifft und einen MinMaxAgenten, der algorithmisch „4-Gewinnt“ spielt und zur Klasse der MinMax-Algorithmen gehört.

Das Paket „gamefunction“ enthält die Klasse NeuralNetwork, die alle Methoden zur Verarbeitung des neuronalen Netzes bereitstellt, sowie das Interface I_NeuralNetwork, welches zur Nutzung des neuronalen Netzes der entsprechenden Klasse später zur Verfügung gestellt werden kann. Mit der Verwendung von Interfaces erreicht man, dass konkrete Klassen nicht als offener Quellcode bereitstehen müssen, und dennoch die Verwendbarkeit konkreter Objekte (wie hier des neuronalen Netzes) gewährleistet ist. Sollten später weitere Verarbeitungsfunktion erstellt beziehungsweise programmiert werden, könnte man die wichtigsten Methoden in einem gemeinsamen Interface (sicherlich mit anderem Namen) zusammenfassen.

„Gamelogic“ ist die Bezeichnung des dritten Pakets und enthält die Klassen Game und Control. Game verwaltet zentral alle Spieleigenschaften, wie das Spielfeld und die Entwicklung desgleichen. Control führt letztendlich alle spielrelevanten Klassen zusammen und steuert zentral den Lern- oder Spielprozess. Hier werden die zwei Agenten als Spieler und das Spielfeld erzeugt und miteinander „bekannt“ gemacht.

Das Paket „helpers“ enthält schließlich einige Hilfsklassen, die zum Beispiel zur Steuerung der Textausgabe während verschiedener Prozesse benötigt werden oder die Namen der Spieler enthalten (wie es bei der Enumeration „E_Player“ der Fall ist).

Die Pakete „gui“ (stellt die – zurzeit statische und inaktive – grafische Oberfläche bereit) und „test_nimm3“ sind ungenutzte Pakete, die zum Teil zu Testzwecken diverser Methoden genutzt wurden oder aufgrund von Zeitmangel ungenutzt sind.

Diese angesprochene Struktur ermöglicht eine gute Übersicht über die Klassen, die zur Analyse der Erlernbarkeit von „4-Gewinnt“ existieren. Ebenso lassen sich leicht weitere Klassen und Pakete ergänzen und integrieren, falls an dieser Arbeit weitergearbeitet werden sollte. Es existieren durchaus einige Alternativen oder Ergänzungen, die diese Arbeit erweitern könnten. Daher sollte eine gute und übersichtliche Struktur hilfreich zur möglichen Weiterentwicklung sein.

4.2.2 RL_NNAgent – Erzeugung der Spielfunktion über neuronales Netz

Die Klasse RL_NNAgent ermöglicht das Anlegen von Lern- und Spiele-Agenten, die mittels neuronalem Netz das Spiel „4-Gewinnt“ erlernen und letztlich auch eingeschränkt spielen können sollten. Der Agent kann durch seine Methoden viele Trainingsspiele durchführen und somit seine eigene Spielfunktion – die vom neuronalen Netz repräsentiert wird – verbessern. Ebenso ist er in der Lage, „4-Gewinnt“ zu spielen; alleine oder gegen einen weiteren Gegner.

4.2.2.1 Methoden aus RL_NNAgent

Die wichtigsten Methoden sind „simulate_game()“ und „make_best_move()“, aus denen der Lernprozess zur Erzeugung einer guten Spielfunktion im Wesentlichen besteht.

Nachfolgend werden die Hauptmethoden etwas detaillierter erläutert und genannt:

simulate_game

ermöglicht das Simulieren eines „4-Gewinnt“-Spiels zur Verbesserung der Spielfunktion

make_best_move

entscheidet über die Auswahl des besten Zuges innerhalb einer bestimmten Situation (durch Nutzung des neuronalen Netzes)

make_random_move

wird zur möglichen Erweiterung des Zustandsraums benutzt, indem mit einer bestimmten Wahrscheinlichkeit Zufallszüge getätigt werden

prepare_input_vector

bereitet den Input-Vektor für das neuronale Netz vor, indem die codierten Spielfelddaten einzelne Eingabeneuronen aktiveren

Diverse interne Methoden werden zur Verbesserung der Programmierstruktur genutzt, sind aber für das Verständnis der Funktionalität der Klasse nicht so relevant.

4.2.2.2 Pseudocode der Simulationsmethode

Die Grundfunktionalität des Lernvorgangs wird am deutlichsten, wenn man den Vorgang erklärt, der beim Simulieren eines „4-Gewinnt“-Spieles ausgeführt wird.

Der folgende Pseudocode erläutert begrifflich die Aktionen, die während des Simulationsvorgangs an entsprechender Stelle durchgeführt werden:

```
simulate_game(Startspieler, Initial-Spielfeld) {
      aktueller Spieler = Startspieler
      aktuelles Spielfeld = Initial-Spielfeld
      // Reset der Netzwerte
      NN.reset_structure()
      // Vorbereiten des Netzinputs
      prepare_input_vector()
      // aus NN berechnen
      NN.response()
      // Netz-Output sichern
      NN.old_output[] = NN.output[]
      // eligibility traces berechnen
      NN.update()
      // Netz-Input zurücksetzen
      NN.reset_input_vector()
      while (Game is not over) {
            // bestmöglichen oder Zufalls-Zug ausführen
            make_best_move() or make_random_move()
            // aus NN neu berechnen
            NN.response()
            // Netz-Fehler berechnen
            NN.set_error()
            // wenn kein Zufallszug, dann Lernvorgang!
            if (kein Zufallszug) {
                  NN.learn()
            }
            // wenn Spiel zu Ende -> Abbruch
            if (game is over) {
                  break
            }
```

```
                // aus NN neu berechnen
                NN.response()
                // Netz-Output sichern
                NN.old_output[] = NN.output[]
                // eligibility traces berechnen
                NN.update()
                // Spielerwechsel
                change_actual_player()
                // Netz-Input zurücksetzen
                NN.reset_input_vector()
            }
        }
```

Die Grundidee besteht darin, dass während des gesamten Spiels jeder Spielzug erlernt wird. Nach einigen Initialisierungsschritten wird ein Spiel solange fortgeführt, bis es beendet ist.

Erlernen bedeutet dabei, dass der Netzoutput, der über die Methode response() berechnet wird, über einen Fehlerwert mit dem neuen Outputwert nach einem Zug verglichen wird. Wie schon zuvor in dieser Arbeit erwähnt, wird der Fehler aus der Formel $\delta_t = r_{t+1} + \gamma \cdot V(\vec{s_{t+1}}) - V(\vec{s_t})$ berechnet. Hier wird der alte Output-Wert (der im Pseudocode mit old_output[] gekennzeichnet ist) vom neuen Output-Wert nach einem jeden Zug abgezogen, letztlich wird noch der Reward der entstandenen Situation hinzugerechnet. Dieser ist bekanntlich erst bei Spielende ungleich Null und daher relevant. Der Fehler wirkt sich nun auf den Lernvorgang aus, da die Kanten des neuronalen Netz unter anderem durch den Fehlereinfluss verändert werden.

Durch die wahrscheinlichkeitsgesteuerte Wahl zwischen Zufallszug und bestmöglichem Zug erreicht man eine gute Abdeckung des Zustandsraums. Würde man jeden Schritt nur aus der Wahl des bestmöglichen Zuges tätigen, könnte sich der Spielverlauf an ein wiederkehrendes Muster anpassen, was letztlich die erzeugten Zustände des Spiels minimiert. Da nicht sicher ist, ob gerade dieser erlernte Spielverlauf von Anfang an gewählt wird, muss eine Streuung der Zugwahl getätigt werden. Das erreicht man über eine zeitweise zufallsgesteuerte Wahl des Spielzuges der möglicherweise eine neue Spielsituation erzeugt, die wiederum ebenfalls neue Zustände erzeugt.

4.2.2.3 Kombination mehrerer neuronaler Agenten

Der Agent, der die oben beschriebene Simulationsmethode verwendet, benötigt zur Entwicklung einer Spielfunktion – wie bereits erwähnt – ein neuronales Netz als grundlegendes Lernmodell. Somit erklärt sich die Tatsache, dass sich viele Berechnungsmethoden direkt auf das neuronale Netz beziehen, wie zum Beispiel die response()- oder learn()-Methode.

Nun könnte ein weiterer neuronaler Agent, quasi als Gegenspieler, ebenfalls ein neuronales Netz erhalten, was in anderer Weise oder anders stark trainiert werden kann. Es ist denkbar, dass man über das Gegeneinanderspielen dieser Agenten die einzelne Netzperformance messen beziehungsweise sogar nachvollziehen kann.

Auf der anderen Seite kann man sich auch vorstellen, dass zwei Agenten das gleiche Netz benutzen, um Spielzüge zu berechnen, aber jeder Agent nutzt die Netzausgabe in anderer Weise und interpretiert das Ergebnis getrennt. So könnte zum Beispiel ein Agent das neuronale Netz zur Wahl des besten Spielzuges befragen. Der Gegner könnte sich der Antwort des gleichen Netzes bedienen, aber zusätzlich beispielsweise noch einen MinMax-Algorithmus verwendet, um die Antwort quasi algorithmisch abzusichern.

So ist jede mögliche Kombination mehrerer Strategien und Modelle denkbar, um die Performance eines Spielagenten zu optimieren.

Hier wird ein neuronaler Agent trainiert und später gegen einen RandomAgenten getestet.

4.2.2.4 Problematik fehlender terminierender Zugbewertung

Eine Problematik, die im Zuge der programmtechnischen Entwicklung auftrat, soll hier auch noch erläutert werden, da sie die Lösbarkeit des Lernproblems auch bei größeren Zustandsräumen deutlich beeinflussen wird.

Die Entwicklung der Methoden „simulate_game()“ und „make_best_move()“ sind größtenteils aus der Methode main() des Pseudocodes eines „self-play TD“-Algorithmus von Sutton und Bonde[15] entstanden. „Self-play“ bedeutet hierbei, dass der Algorithmus eigenständig Wissen aus Trainingsspielen erlernen kann, ohne dass zur Ausführungszeit der Methode schon alle auftretenden After-State-Situationen bekannt sind. Der Pseudocode aus Kapitel 2.4 ermöglicht zwar auch ein „self-play“ des Spiels, es müssen aber bereits zur Ausführung der Methode alle After-States des Spiels bekannt sein. Hier werden aber Trainingsspiele sukzessive entwickelt und ausgewertet. Dennoch bleibt der Lerngedanke durch die RL-Strategie erhalten.

Der zu tätigende Zug bei „4-Gewinnt“ wird durch die Methode make_best_move() errechnet. Die Berechnungsgrundlage bietet dabei die Funktionsapproximation durch das neuronale Netz oder eine lineare Funktion mit anderen Features oder Parametern.

$f(w, s_{t+1})$ ist somit die Funktion, die zum Beispiel durch ein neuronales Netz die Spielefunktion möglich genau approximieren soll. „w“ bezeichnet dabei alle Parameter, die die Funktion als Inputgrößen beeinflussen, s_{t+1} ist der After-State zum Zeitpunkt t+1.

15 siehe Sutton/Bonde (1992)

Allerdings spielt der Spieler, der diesen After-State (der mit der Funktion f berechnet werden kann) hinterlässt, ebenso eine Rolle bei der Wahl des bestmöglichen Spielzuges, wie der mögliche Reward. Die Erklärung hierzu ist recht einfach und nachvollziehbar. Man betrachte dazu folgendes Szenario:

Als Beispiel liegt ein Spielzustand vor, bei dem man sich kurz vor dem Ende eines laufenden „4-Gewinnt“-Spiels befindet. Rot ist am Zug und kann mit einem Zug in die letzte Spalte gewinnen.

Nun kann die Spielfunktion alleine aber gar keine Aussage über den korrekten Wurf treffen. Das liegt an der Eigenart des Lernens mit RL. Hierbei wird ja nie der Zustand s_T (also der Endzustand eines Spieles) erlernt. Es werden lediglich die vorangegangenen Spielsituationen auf ihre Gewinnchance für den anziehenden Spieler bewertet. Somit muss es einen zusätzlichen Einfluss auf die Bewertung des bestmöglichen Spielzuges geben.

Hierzu berücksichtigt man den Reward(s_{t+1}), der ja nur bei Beendigung eines Spieles einen relevanten Wert annimmt und somit die Summe aus Spielfunktion und Reward auch nur dann verändert. Ebenso ist die Information, welcher Spieler den neuen (terminierenden) Zustand des Spieles hinterlässt, wichtig. So kann man beim Reward zwischen Belohnung oder Bestrafung unterscheiden. Der Einfachheit halber, rechnet man also für die Wahl des Zuges $f(\vec{w}, s_{t+1}^{\rightarrow}) + r(s_{t+1}^{\rightarrow}) + p$, wobei p entweder den Wert „1“ oder „-1“ annimmt.

Nur diese Berechnung bewertet den spielbeendenden Zug korrekt, da dieser über den Einfluss des Rewards mehr Gewicht erhält als alle anderen möglichen Züge. In jeden anderen Fall spielt der Reward auf die Wahl des korrekten Zuges keine Rolle, da er immer nur mit „0“ bewertet wird.

Die Berücksichtigung des Rewards bei der Wahl des bestmöglichen Zuges war allerdings nicht innerhalb der Pseudocodes erwähnt, sodass diese fehlenden Information schließlich auch ganz anderen Einfluss auf die Spielverläufe von „4-Gewinnt“-Spielen hat. Zuvor war der korrekte spielterminierende Zug eher zufällig gefunden worden oder ist durch die Wahl der Implementierungsvariablen beeinflusst worden. Als Beispiel sei hier eine HashMap genannt, die jeder möglichen Spalte, in die gezogen werden kann, die entsprechende Zeile zuordnete. Dies passiert aber regelmäßig sortiert bei Spalte eins beginnend und Spalte sieben endend. Somit ist bei gleichem Wert aus der Spielfunktion für mehrere Züge stets die Spalte eins gewählt worden, da sie als ersten innerhalb der HashMap aufgerufen und auf ihre Wertigkeit untersucht worden ist.

Das recht einheitliche Bild der Trainingsspiele, die zu Lernzwecken erstellt und gespielt wurden, beinhaltete daher auch häufig den Sieg des Anziehenden über einen Vierer in Spalte eins. Das

liegt aber nicht an der schnell erlernten Wahl durch RL-Strategie, sondern vielmehr daran, dass alle Spielzüge gleich bewertet wurden und daher nur in die erste untersuchte Spalte geworfen wird. Der Einfluss der RL-Strategie wird zwar berücksichtigt, spielt aber so oder so kaum eine Rolle, da die erste Spalte aufgrund der Implementierung sowieso gewählt wird.

Somit ist es notwendig, den Pseudocode und daher auch die Implementierung auf die oben beschriebenen Änderungen anzupassen, was im Zuge der Entwicklung dieser Arbeit auch passierte. Das ermöglicht der Simulationsmethode die direkte Spielbeendigung, wenn eine Gewinnmöglichkeit durch einen Spieler besteht. Weiterhin ist eine deutlich klarere und korrekte Analyse der Lernprozesse und Trainingsspiele möglich, da die Spielzüge dem Verlauf menschlicher und logischer Spielweise mehr ähneln als zuvor. Auch die Terminierung eines Spiels ist nun erklärbar und geschieht (folge)richtig.

4.3 Konzeptionelle Gestaltungsvarianten des Input-Vektors

Man kann sich vorstellen, dass eine unterschiedliche Eingabecodierung am neuronalen Netz zu unterschiedlichen Ergebnissen der Spielfunktion und der Spielstärke des Agenten führen kann. Ebenso klar und erklärbar ist, dass aussagekräftigere Features als Eingabe an ein neuronales Netz bessere und deutlichere Ergebnisse liefern als weniger starke Features. Somit ist es relevant und wichtig, die geeigneten Features zur Codierung der aktuellen Spielsituation zu finden.

Je nach dem, wie das Spielfeld oder der entstandene After-State codiert wird, kann auch die Entscheidung über den besten Zug unterschiedlich gut oder schnell erlernt werden. Je mehr Informationsgehalt die Codierung der Spielsituation trägt, desto stärker ist die Netzperformance und desto besser wird die Spielfunktion. Die Schwierigkeit besteht im Herausfinden und Darstellen des Informationsgehalts einer Spielsituation und in der Umsetzung der Codierung.

Um die verschiedenen Codierungsalternativen zu testen, werden keine vollständigen „4-Gewinnt“-Spiele gespielt und erlernt. Die Problematik beim Erlernen kompletter Spiele wird deutlich, wenn man sich den Zustandsraum, der durch alle möglichen After-States gekennzeichnet ist, ansieht. Wie schon in Kapitel 3.3.1 erwähnt, gibt es circa $2^{49} \approx 5{,}62 * 10^{14}$ Zustände. Die Zahl repräsentiert also die Anzahl möglicher After-States, die theoretisch in Betracht kommen, um das Spiel in seiner Spielbreite vollständig zu erlernen. Da dies als Lernprozess aber auf den normalen Arbeitscomputern nicht in endlicher Zeit möglich ist, muss man sich – was nachfolgend auch angewendet wird – auf eine bestimmte Spielsituation (mit weniger Folgezuständen) einschränken, die als Initialsituation vorliegt, von der aus ein Erlernen des weiteren Spiels möglich ist. Daher werden alle Untersuchungen zunächst von sogenannten Endspielen ausgehen, die bereits eine bestimmte Belegungssituation des Feldes vorgeben.

4.3.1 Codierung über Spielfeldbelegung (3 Neuronen pro Spielfeld)

Schon in Kapitel 5.1 wurde davon gesprochen, dass eine gute und ernstzunehmende Codierungsmöglichkeit dadurch besteht, dass man das aktuelle Spielfeld beachtet. Genauer bedeutet das, dass man den aktuellen Belegungszustand des Spielfeldes nach jedem Zug codieren und als Input dem neuronalen Netz bereitstellen muss. Wie die Spielfeldbelegung codiert wird, hängt nun von den Programmier- und Optimierungsmöglichkeiten ab.

In einer ersten Variante wird jedes einzelne Spielfeld durch drei Eingabeneuronen vertreten. Dabei wird das erste Neuron dann aktiviert (also mit einer „1“ belegt), wenn das Feld durch einen Spielstein von Spieler ROT belegt ist. Das zweite Neuron wird aktiviert, wenn Spieler GELB seinen Stein dorthin wirft und das dritte Neuron ist aktiv, wenn keine Belegung des Feldes vorliegt, dass heißt, wenn das Feld noch frei wäre. Somit ergibt sich bei 42 Spielfeldern ein Input-Vektor der Größe 126 (= 42 * 3), der die Eingabeschicht des neuronalen Netzes darstellt.

Der Nachteil dieser ersten Codierung ist die Größe und Ausprägung des Input-Vektors. Bei so großer Anzahl an Neuronen ist die Anzahl der Kanten sehr hoch und daher die Lerngeschwindigkeit und der -fortschritt möglicherweise gering.

Neutral zu werten ist die Tatsache, dass eine Abhängigkeit zwischen je drei Neuronen besteht. Das Netz müsste erlernen, dass immer nur eins der drei zusammengehörenden Neuronen aktiviert ist. Das kann als Vorteil gewertet werden, wenn man formuliert, dass somit eindeutige Eingaben getätigt werden. Es existiert eine klare Aufteilung der Aufgaben der Neuronen. Nachteilig wäre eher, dass oben beschriebener Zusammenhang der Neuronen erst zu erlernen ist.

Ein Vorteil besteht aber in der gesamten Art der Codierung. Da man das Spielfeld als Ganzes untersucht, werden immer alle Zusammenhänge der Spielsteine erfasst. Das Erlernen dieser Zusammenhänge ist dadurch allerdings noch nicht garantiert. Dennoch liegt bei einem Lerneffekt das gesamte Spielfeld zugrunde und wird als Gesamtes analysiert. Es spielt somit eine direkte Rolle für den Lernprozess und wird auch nur in diesem Erscheinungsbild später erneut wieder erkannt. Das bedeutet aber auch, dass bei weiteren Spielen nur dann der bisherige Lerneffekt genutzt werden kann, wenn die Spielfeldbelegung der bisher erlernten Situation gleicht; man befindet sich somit auf einem bereits erlernten Pfad einer Zustandsfolge und nutzt vorhandene Kenntnisse.

Ein weiterer Vorteil ist das Wissen, dass das Spielfeld definitiv verwendbar für eine Codierung ist. Wer sollte – wenn nicht die aktuelle Spielfeldsituation – eine Aussage darüber tätigen können, wie Erfolg versprechend eine bestimmte Situation (after-state) ist? Bevor man also

Muster oder spielstrategische Überlegungen als Input-Vektor codiert, sollten die einfachsten Codierungsvariationen auf ihren Lernerfolg untersucht werden.

Bei oben beschriebener Codierungsvariante sind also immer 42 der 126 Neuronen aktiviert, da ja jedes Feld entweder belegt oder frei ist. Die wesentliche Frage ist, welcher Zusammenhang zwischen den einzeln aktivierten Neuronen besteht und welche Neuronen dabei die wichtigste Rolle für einen Sieg des Anziehenden spielen (das wird unter anderem letztlich erlernt).

Eine vorab gestellte Hypothese ist, dass der Lerneffekt wohl nur langsam eintritt, sodass man nur wenig erlernte Spielzüge erkennen kann. Eine Einschränkung auf wesentliche Spielsituationen oder Spielzüge, die gegen Ende eines Spiels auf ein halbvolles Spielbrett zu tätigen sind, würden sicher dem Lernerfolg entgegenkommen.

4.3.2 Ergebnisse der ersten Codierungsvariante

Die vorab erläuterte Codierungsvariante wird nun praktisch am Lern-Agenten angewendet und mit Hilfe der programmierten Testumgebung ausgewertet. Man erhält aussagekräftige Ergebnisse, die im Folgenden dargestellt, ausgeführt und zum Teil erklärt werden:

Das Endspiel, welches bei genauer Betrachtung recht trivial zu Ende gespielt zu werden scheint, wird in Abbildung 17 gezeigt. Spieler ROT ist am Zug. Er kann das Spiel direkt beenden.

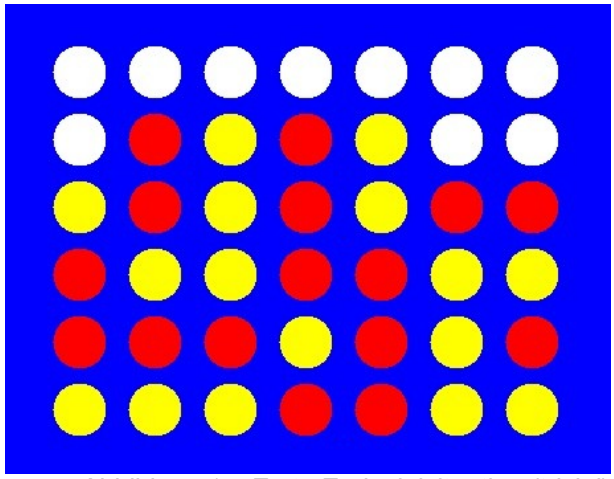


Abbildung 17: Erste Endspielsituation (trivial)

Getestet wird hierbei nur die in Kapitel 5.2.2.4 erwähnte Änderung am Quellcode, die zulässt, dass der Agent eigenständig das Spiel beendet, wenn es möglich ist. Die Testergebnisse zeigen, dass der Agent gegen den randomisierten Agenten mit 100% Quote gewinnt. Selbst bei 10000 Trainingsspielen, die zur Anpassung des neuronalen Netzes gespielt worden sind, ändern sich an der eindeutigen Gewinnquote nichts. Das bedeutet, die Beendigung des Spieles durch den Zug des roten Spielsteins in die mittlere Spalte wurde korrekt implementiert und im Grunde sogar erlernt. Denn die Bewertung des Abschlusszuges könnte ja durch die Bewertung eines Wurfs in eine andere Spalte übertrumpft werden. Da dies nicht passiert, wurde auch kein anderer günstigerer Spielzug fälschlicherweise erlernt. Dies ist ein positives Ergebnis, da es zeigt, dass der Lern- und auch der Spielalgorithmus des Agenten korrekt funktioniert. Die genauen Auswertungsergebnisse findet man im Anhang A.3.

Das zweite Endspiel, welches zu Testzwecken verwendet wird, lässt längere abschließende Spielfolgen zu, da mehrere Züge (mindestens zwei) zum Sieg eines Spielers notwendig sind. Dadurch dass zwei komplette Zeilen des Spielbretts noch unbelegt sind, ergibt sich ein Zustandsraum von $(2^3\text{-}1)^7 = 7^7 = 823543$ möglichen Zuständen, was insgesamt zwar noch groß, aber deutlich geringer als der gesamte Spielzustandsraum ist. Da nicht alle Zustände gewinnrelevant sind und damit nicht unbedingt erlernt werden müssen, minimiert sich das Lernproblem und ist in endlicher Zeit lösbar. Das Spielbrett zu dieser Endspielsituation wird in Abbildung 18 gezeigt. Spieler Rot ist auch in dieser Situation am Zug, kann aber im Gegensatz zu Endspielsituation 1 das Spiel durch den nachfolgenden Spielzug noch nicht beenden.

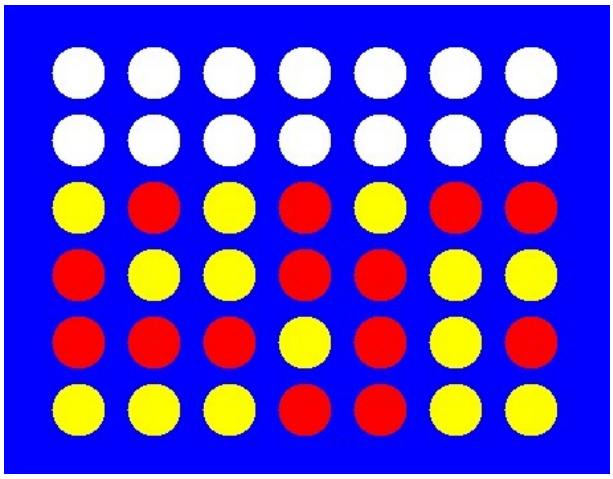


Abbildung 18: Erweiterte Endspielsituation

Spielt nun der trainierte RL_NNAgent (Rot) gegen den zufallsgesteuerten Agenten (Gelb), siegt er im Durchschnitt mit einer Quote von über 80%. Auch hier stehen die kompletten Testspielergebnisse im Anhang A.3 in den entsprechend untergeordneten Kapiteln.

Bei den Tests fällt auf, dass verlorene Spiele, daher verloren gegen, weil der Gegner einen gelben Spielstein in die mittlere Spalte wirft und somit die Siegchance des Anziehenden erheblich minimiert. Gewonnene Spiele werden über die Erzeugung eines senkrechten Vierers in der mittleren Spalte gewonnen. Dieses Muster (beziehungsweise dieser Spielverlauf) wurde also scheinbar auch zuvor trainiert. Das bedeutet aber, dass der RL_NNAgent nur relativ unflexibel auf andere Situationen reagieren kann, wenn sein gewohnter Siegspielzug oder der, diesem Zug Vorausgehende nicht mehr so möglich ist, wie zuvor erlernt.

Dabei bestünde theoretisch die Möglichkeit, den ersten Spielzug (für den Anziehenden) nicht wie erlernt, in die mittlere Spalte zu werfen, sondern in die dritte Spalte. Somit ergäben sich zwei diagonale Dreier, die entweder über die zweite oder die vierte Spalte und oberste Zeile zum Vierer ausgebaut werden könnten. Da diesem Muster aber zuvor einige bestimmte Spielzüge vorausgehen würden, scheint der Zug in die mittlere Spalte mehr Erfolg zu bringen und wird daher eher gewählt.

Ebenso wurde durch die Testspiele erkannt, dass der gelernte Spielverlauf beziehungsweise die Bewertung der After-States nicht ausreicht, um gegen einen Agenten zu bestehen, der die Berechnungstechnik eines MinMax-Algorithmus[16] nutzt.

16 Algorithmen, die die MinMax-Technik nutzen, berechnen über sogenannte Zustandsbäume die möglichen, optimalen Wege innerhalb des Baumes, um den bestmöglichen Zug zu erfassen.

Bei Testspielen gegen einen solchen Agenten, beträgt die Quote der gewonnenen Spiele des RL_NNAgenten nur 1-2%.

Der MinMaxAgent erkennt sofort, dass er die mittlere Spalte mit einem Wurf in die Selbe schließen muss, um seinem Gegner alle Siegchancen zu nehmen. Da der MinMaxAgent danach selber das Spiel für sich entscheiden kann, ergibt sich somit seine deutliche Überlegenheit. Die Technik des MinMaxAgenten ist im Gegensatz zur gering trainierten Spielfunktion des RL_NNAgenten einfach noch zu stark, als dass der trainierte Agent eine Chance auf eine bessere Siegquote hätte. Da der MinMax-Algorithmus immer die (innerhalb der Beschränkung berechenbaren) besten Züge wählt und der RL_NNAgent nur teilweise logisch und trainiert handelt, siegt der MinMaxAgent sehr häufig und es ergibt sich eine solch schwache Quote des trainierten Agenten.

Man müsste neutralere Endspielsituationen wählen oder Situationen generieren, bei denen der RL_NNAgent mehr Chancen und Ausbaumöglichkeiten besitzt. So könnte man die eigentliche Leistungsstärke des RL_NNAgenten messen, indem er gegen den MinMaxAgenten spielt und versucht, Spiele möglichst für sich zu entscheiden.

Durch eine alternative Codierungsvariante wird nachfolgend versucht, die Leistungsfähigkeit des RL_NNAgenten zu verbessern, um ihm mehr Chancen gegenüber dem MinMaxAgenten einzuräumen.

4.3.3 Codierung über verkürzte Spielfeldbelegung (-1/1/0-Strategie)

Wie bereits erwähnt, kann es notwendig sein, dass man alternative Codierungsvarianten implementiert und testen, um die Lernfähigkeit und -geschwindigkeit zu optimieren. Häufig benötigt man bessere Features als Eingabevektor des neuronalen Netzes, da der Zustandsraum einer Problematik zu groß ist, um ihn im Lernprozess vollständig zu erfassen. Je genauer die Features das Problem kategorisieren beziehungsweise codieren, desto geringer wird die Anzahl der möglichen Zustände.

Auch bei „4-Gewinnt“ sind verschiedene Codierungsalternativen denk- und umsetzbar. So kann man zum Beispiel, anders als bei der ersten Codierung, alle 42 Spielfelder als Eingabeneuronen implementieren, wobei die Belegung eines Felders dieses Mal über den anliegenden Wert am Neuron geregelt wird. So kann ein leeres Spielfeld mit „0“ belegt werden. Wenn Spieler ROT seinen Stein in ein Feld gelegt hat, erhält das entsprechende Eingabeneuron den Wert „1“, bei Spieler GELB entsprechend die Zahl „-1“. So werden jedem Neuron drei verschiedene Belegungszustände direkt über den Wert mitgegeben und es verringert sich insgesamt die Anzahl der Eingabeneuronen. Dies könnte eine Optimierung des

neuronalen Netzes erzeugen, da die Zahl der zu erlernenden Zusammenhänge der Eingabeneuronen deutlich geringer ist. Das Netz muss beispielsweise nicht mehr erlernen, dass immer drei Neuronen ein Spielfeld repräsentieren und nur die unterschiedliche Belegung codieren. Dieser Zusammenhang fällt bei der gerade beschriebene Codierung weg.

Die unterschiedliche Bewertung der Wichtigkeit eines Neurons wird hier auch deutlicher. Die Belegung eines Feldes kann entweder einen positiven Einfluss auf das gesamte Netz erzeugen (Spieler ROT = „1“) oder einen negativen (Spieler GELB = „-1“). Die Interpretation dieser Unterschiedlichkeit ist deutlicher codiert als zuvor, wo der Unterschied durch zwei Neuronen codiert werden musste (Spieler ROT -> Neuron x = 1 oder Spieler GELB -> Neuron x+1 = 1).

Man erwartet schnellere und zufriedenstellende Ergebnisse beim Lernprozess des Agenten, wenn diese Codierungsalternative angewendet wird. Die Frage ist, ob diese Erwartung auch bestätigt wird oder ob sich die Leistung des Agenten als noch zu schwach herausstellen wird.

4.3.4 Ergebnisse der zweiten Coderierungsvariante

Das hauptsächliche Resultat aller Tests der zweiten Codierungsalternative ist, dass durch ihre Anwendung schneller Ergebnisse erzielt werden. Dabei wird auch deutlich, dass die Siegquoten sich allerdings kaum merkbar verbessern. Die Frage ist, wie das begründet werden kann?

Der schnellere Testdurchlauf ist relativ einfach zu begründen. Dadurch, dass nicht mehr 1260 (126 * 10) Kanten, sondern wesentlich weniger Kanten zur Hidden-Schicht angepasst werden müssen, minimieren sich die Ausführungszeiten der entsprechenden Methoden. Bei bis zu 100.000 Spielen macht sich dies vor allem durch die geringeren Ausführungszeiten der Testdurchläufe bemerkbar. So verringern sich die Zeiten um bis zu 40% und mehr.

Bessere Ergebnisse, gemessen an höheren Siegquoten, werden dabei aber meistens nicht erzielt. Wenn auch erkennbar ist, dass häufiger hohe Quoten bei Spielen gegen den MinMaxAgenten erzielt werden, bleibt der Fortschritt in der Spielperformance im Durchschnitt jedoch aus. Einzig bei 10.000 Trainingsspielen mit darauf folgenden Testspielen gegen den MinMaxAgenten ist die Siegquote des Lernagenten deutlich besser als zuvor. Dennoch werden durchschnittlich ähnliche Quoten erzielt, wie es bereits zuvor der Fall war. Es scheint, als ob der Zustandsraum entweder nicht stärker erforscht und bewertet wird oder als ob die vorliegende Endspielsituation, zumindest in Spielen gegen den MinMaxAgenten, keine höheren Quoten zulässt. Der Grund der Stärke dieses Agenten, wurde ja bereits in Kapitel 4.3.2 erklärt.

Der größte Nutzen, der aus dieser Codierung gezogen werden kann, ist also eine schnellere Ausführung des Lernvorgangs, was aber auch eine höhere Anzahl an möglichen Trainingsspielen zulässt. Dies bedeutet aber eine Verbesserung der Testverfahren.

Weitere Testresultate mit genauerer Gegenüberstellung der Ergebnisse werden in Abbildung 19 deutlich gemacht. Hier erkennt man recht deutlich, dass der IBEF-Agent bei allen Lernvorgängen als Anziehender Spieler mit hoher Wahrscheinlichkeit gewinnt. Interessant ist allerdings vor allem die Tatsache, dass der RL-Agent bei höherer Trainingsspielanzahl mehr Siegchancen gegen den MinMax-Agenten (der nach der IBEF-Bewertungsfunktion spielt) erhält, als bei kleiner Trainingsspielanzahl. Bei 100000 Trainingsspielen scheint der RL-Agent allerdings wieder Informationen verloren zu haben, da die Spiel-Quoten hier deutlich schlechter sind, als zuvor bei 50000 oder sogar bei nur 10000 Trainingsspielen.

Codierungsalternative	**Spiel**	**Anzahl Trainingsspiele**								
		10000			***50000***			***100000***		
126 Inputneuronen	RL vs. IBEF	27,33	0,67	72,00	64,00	5,33	30,67	14,00	1,00	85,00
	IBEF vs. RL	96,00	2,00	2,00	96,00	3,33	0,67	97,67	1,33	1,00
42 Inputneuronen	RL vs. IBEF	35,00	2,67	62,33	67,00	4,33	28,67	20,67	1,33	78,00
	IBEF vs. RL	97,00	1,00	2,00	93,67	2,00	4,33	96,00	2,33	1,67
		Win	*Draw*	*Loose*	*Win*	*Draw*	*Loose*	*Win*	*Draw*	*Loose*

Abbildung 19: Weitere Tests des RL-Agenten gegen den IBEF-Agenten

Die Angaben der Win-,Draw- und Loose-Werte sind prozentual zu verstehen, was verdeutlicht, dass hier auch mit unterschiedlicher Anzahl an Testspielen gearbeitet wurde. Die Testparameter sind nachfolgend dargestellt:

- Lernschrittweiten α und β: 0,5
- Discount-Faktor γ: 0,9
- Wahrscheinlichkeit für Zufallszug ε: 30%

Die neuronale Netzstruktur, die der RL-Agent nutzt, ist über die Codierungsart im Grunde schon vorgegeben. Hinzu kommt eine Hidden-Schicht, die sich aus 10 Neuronen zusammensetzt und ein Output-Neuron. Als Transferfunktion wird die Sigmoid-Funktion sowohl an der Hidden- als auch an der Output-Schicht genutzt.

Bei der Auswertung einzelner Testspielszenarien wurde deutlich, dass der RL-Agent je nach Lernerfolg recht schnell den Sieg gegen den MinMax-Agenten erzielt. Das heißt, wenn die Kantenanpassungen im Lernvorgang über die Trainingsspiele gut gelingt, ist der RL-Agent in der Lage bei diesem Endspiel sogar gegen den MinMax zu gewinnen. Bei schlechter Kantenanpassung – was im Laufe des Testprozesses leider häufiger vorkommt – wird aber auch deutlich, dass der Lernerfolg ausbleibt und die Siegchance des RL-Agenten kaum noch besteht. Hier ist die zufallsgesteuerte, initiale Belegung der Kanten wohl ausschlaggebend für die Performance des Netzes. Je nach Startbelegung kann eine gute Anpassung der Kanten im Laufe der Trainingspiele passieren oder nicht.

4.3.5 Codierung über Muster (konzeptioneller Vorschlag)

Ein weitere Überlegung, die ein bisher komplett unbeachtetes Feld an Codierungsmöglichkeiten zulässt, lässt sich anstellen, indem man über Mustervorkommen im visualisierten Spielfeld nachdenkt. Muster bestimmen und beeinflussen die täglichen Gedanken. Hört man Musik, versucht man Muster innerhalb eines Liedes oder Instrumentalstückes zu erkennen. Sieht man Farben und Formen, erkennt das Gehirn daraus oft bekannte Bilder oder Grafiken. Das menschliche Gehirn ist darauf angelegt, Muster aus Informationen herauszufinden oder zu erzeugen. Daher wird man auch oft bei Spielen versuchen, Muster für seine Spielweise oder die Abläufe zu entwickeln. Man spricht ja auch nicht umsonst von dem Begriff der „Musterlösung“.

Somit lohnt es sich, über Muster im Spiel „4-Gewinnt“ nachzudenken. Es geht schließlich um die Erzeugung eines bestimmten Musters; der Vierer ist gewinnbringend. Oft schaut der menschliche Spieler nach Mustern, die zu solchen Vierern ausgebaut werden können.

Als Muster wird nachfolgend gesprochen, wenn ein bestimmtes Bild an Spielsteinkombinationen auf dem Spielbrett zu erkennen ist.

Also sollte ein intelligenter Computergegner in der Lage sein, Muster, die für den Sieg des Spiels relevant sind, zu erkennen, zu analysieren und zu interpretieren, um daraus seine Zugbewertung zu entwickeln.

4.3.5.1 Relevante Muster

Die Frage, die sich als erstes stellt, wenn man über Muster im Spiel „4-Gewinnt“ nachdenkt, ist: Welche Muster gibt es und welche sind überhaupt relevant?

Sicher ist, dass der Vierer als Muster unumgänglich ist. Er repräsentiert ja den Siegzustand. Wenn also ein Vierer auf dem Spielbrett gefunden wird, ist das Spiel beendet.

Wenn aber später die Muster an die Eingabeneuronen des neuronalen Netzes angelegt werden, ist der Vierer nicht mehr unbedingt relevant. Aus einigen vorausgegangenen Bemerkungen dieser Arbeit weiß man, dass der terminierende Spielzustand nicht mehr erlernt beziehungsweise trainiert wird. Daher ist der Vierer als Eingabeneuron nicht notwendig, um eine gute Spielfunktion zu erzeugen. Diese Funktion soll später ja nur die Vorzustände zu einem guten Spielausgang bewerten und nicht den terminierenden Spielzustand. Der Vierer kommt also als Muster nur zur Anwendung, wenn in der Methode „game_over()“ getestet wird, ob ein Spiel beendet wurde und sich in einem terminierenden Zustand befindet.

Betrachtet man nun einen möglichen Spielzustand, bevor ein Spiel gewonnen wird, fällt auf, dass der Sieg nur durch das Vorhandensein eines freien Dreiers möglich ist. Ein freier Dreier

repräsentiert drei aufeinanderfolgende, gleichfarbige Spielsteine, die mindestens zu einer Seite (oben, unten, links oder rechts) offen sind, das heißt, noch freie Spielfelder haben. Ein solcher Dreier geht jedem Spielgewinn voraus, da aus ihm erst ein Vierer erzeugt werden kann.

Nun stellt sich die Frage, ob ein Zweier auch schon gewinnbringende Informationen beinhaltet, das bedeutet, ob das Muster „Zweier“ an sich, genügend Relevanz für ein „4-Gewinnt“-Spiel trägt, um es als Eingabeneuronen dem neuronalen Netz bekannt zu machen. Sicherlich ist es nicht falsch, einen Zugang zu dieser Information zu ermöglichen. Dennoch muss man aufpassen, dass die Fülle an Möglichkeiten, wie ein Zweier im Spiel vorkommen kann, nicht überhand nimmt. Man muss sich also explizit überlegen, welche Muster, die „Zweier“ anzeigen, wichtige Informationen tragen, die den Lernagenten das Spiel gewinnen lassen könnten.

So ist ein Zweier, der nur zu einer Seite offen ist und dort nur noch ein freies Spielfeld hat, unnütz, wenn man ihn als Ausbaumöglichkeit für einen Vierer betrachtet, weil hieraus kein Vierer entstehen kann. Viel wahrscheinlicher ist der Gewinneinfluss eines Zweiers, der zu beiden Seiten die Möglichkeit offen hält, ihn zu einem Vierer zu erweitern. So muss also eine Art Rangliste der Gewinnrelevanz von Mustern entstehen, die natürlich wieder auf unterschiedlichste Weise erzeugt werden kann. So ist zum Beispiel denkbar, dass Muster während Testspielen gezählt und gespeichert werden und deren Relevanz und Einfluss auf den Sieg eines Spielers festgehalten wird. So ergäbe sich mit der Zeit eine aussagekräftige und realistische Liste aller Muster samt ihrer Einflüsse auf den Gewinn des Spiels.

Weitere Muster könnten auch durch Betrachtung des gesamten Spielbrettes erzeugt werden. Zum Beispiel darf der Einfluss einer Zugzwang-Situation nicht unterschätzt werden. Einige Kapitel zuvor beschreibt diese Arbeit, was genau die Chance des Zugzwangs ist. Der Einfluss eines Muster, das eine Zwickmühle repräsentiert ist wohl am größten, da im Falle einer solchen Situation der Sieg für einen Spieler oft schon feststeht. Aber auch diese Situation müsste in ein Muster überführt und dessen Einfluss auf den Spielgewinn getestet und analysiert werden.

So ergeben sich viele relevante Muster, die bereits in unterschiedlichen, vorangegangenen Diplom- und Bachelorarbeiten thematisiert wurden. So setzte sich Anna Klassen 2005 bereits mit der Erzeugung einer Java-Klasse zur Generierung von Mustern im Spiel „4-Gewinnt“ auseinander, die einem neuronal lernenden Computergegner als Spielbrett-Analyse zum Trainieren bereitgestellt wurden. Zuvor haben Thorsten Wende und Thorsten Rudolph im Jahr 2003 eine Spielumgebung für „4-Gewinnt“ erstellt, die einem neuronal lernenden Computergegner unterschiedliche Trainingsmöglichkeiten und natürlich Spielmöglichkeiten zur Verfügung stellt. Die Weiterführung dieser Entwicklung und Integration der RL-Strategie in die bestehende Umgebung ist aus unterschiedlichen Gründen nicht zu empfehlen.

Die Kombination aus dieser Diplomarbeit und den Vorarbeiten sollte aber genügend Grundlage zur Verfügung stellen, um die Musterüberlegungen in die RL-Lernstrategie einzubringen und genauere und passendere Spiel-Features zu generieren, damit dem Lernagenten letztlich eine stärke Performance ermöglicht wird und er eine größere Gewinnchance gegen ein MinMax-algorithmisch spielenden Agenten oder menschliche Gegner hat.

4.3.5.2 Interpretation von Mustern als Eingabevektor des neuronalen Netzes

Will man nun gewinnrelevante Muster als neue Features der Spielfunktion deklarieren und somit dem neuronalen Netz als Eingabevektor zur Verfügung stellen, benötigt man eine Methode, die zunächst das Vorkommen beziehungsweise die Häufigkeit der Muster misst und diese codiert als Eingabevektor dem neuronalen Netz zur Verfügung stellt. Man erkennt letztlich nur aus den Lernprozessen und der daraus resultierenden Spielstärke des Agenten, wie das Netz mit den gegebenen Spielinformationen umgegangen ist.

Ein möglicher Ablauf könnte hier sein:

1. Scannen auf Mustervorkommen in einer bestimmten Situation der Spielbrettbelegung
2. Zählen der Muster nach Häufigkeit und skalieren
3. Wert dem entsprechenden Neuron, welches das Muster „vertritt“, anlegen
4. Output des neuronalen Netz berechnen und weiter nach RL-Strategie verfahren

Die bisherige „prepare_input_vektor()“-Methode würde nun das Scannen auf Muster und Zählen deren Häufigkeiten sowie die Erzeugung eines komplett neuen Eingabevektors beinhalten. Dabei gäbe es so viele Eingabeneuronen, wie es Muster gibt, nach denen gescannt würde. Jedes Neuron erhielte als Wert nun die skalierte Häufigkeit des Vorkommens des entsprechenden Musters. Es entstehen allerdings eine Reihe an Fragen, mit denen man sich auseinandersetzen muss:

- Welche Muster sind gewinnrelevant (Frage bereits weiter oben erwähnt)?
- Wie skaliert man die Häufigkeit des Vorkommens der Muster?
- Spielt die Position einzelner Muster vielleicht eine größere Rolle als die Häufigkeit?

Diese und weitere Fragen könnten zum Inhalt weiterer wissenschaftlicher Arbeiten werden, die genau diese theoretisch beschriebenen Untersuchungen thematisieren. Hier bleibt aus Zeitgründen nur der Verweis auf die möglichen Denkansätze, die in Praxistests untersucht und interpretiert werden müssten.

4.3.6 Gegenüberstellung der Alternativen

Allein aus zeitlichen Gründen lohnt sich die Implementierung der zweiten Codierungsalternative. Diese errechnet recht schnell eine Spielfunktion, die sich in der Stärke aber nur unwesentlich von einer Funktion, resultierend aus der ersten Codierung unterscheidet.

Die Überlegung 126 Eingabeneuronen, also drei pro Spielfeld zu codieren, erübrigt sich im Grunde, wenn man die zweite Alternative untersucht. Dadurch, dass die Belegungszustände der Spielfelder nun über die angelegten Werte der Neuronen codiert sind, ist die erste Überlegung durch eine bessere Codierungsmöglichkeit abgelöst worden. Der Lernprozess sollte sogar einfacher sein, da der Zusammenhang dreier Neuronen zu einer logischen Einheit nicht mehr erlernt werden muss. Die Tatsache, dass bei der ersten Codierungsalternative immer drei Neuronen zusammengehören, von denen aber immer ein Neuron aktiviert ist und zwei nicht, wird durch die zweite Überlegung zur Codierung abgelöst. Außerdem stellt die gerade beschriebene Tatsache einfach einen zu großen Lernprozess dar, der im Grunde aber gar nicht notwendig wäre. Bei der zweiten Alternative wird jedes Feld des Spielbrettes durch ein Neuron repräsentiert, die Belegung des Feldes wird durch den angelegten Wert gesteuert. So ändert sich je nach Belegungssituation auch die Bewertung eines Neurons und somit der Einfluss dieses Neurons – und des Feldes – auf die gesamte Spielsituation. Das war zwar zuvor auch der Fall, aber um diesen Effekt zu erkennen beziehungsweise zu erlernen, bedarf es wesentlich mehr (unterschiedlicher) Daten, als nun notwendig sind. Dies wird durch die schnelleren Lernprozesse bei Codierungsalternative zwei und die etwas besseren Siegquoten deutlich.

Eine Codierung, die Muster und spielrelevante Situationen, die das gesamte Spielbrett betreffen, berücksichtigen würde, wäre sicherlich ein guter und entscheidender Schritt zur Entwicklung einer wesentlich stärkeren Spielfunktion, die auch der MinMax-algorithmischen Spielweise entgegentreten könnte. Leider ist es aus Zeitgründen nicht möglich, diese bereits angesprochenen Überlegungen in die Praxis umzusetzen und durch Tests zu validieren.

Dennoch ist durch Verwendung der RL-Strategie deutlich geworden, dass es einem künstlichen (maschinellem) Spieler möglich ist, – zumindest im kleinen Rahmen – strategische Brettspiele zu erlernen.

Das gesamte Kapitel, welches sich mit dem Spiel „Nimm-3“ auseinandersetzt, zeigt auf, dass es über verschiedene Berechnungsmethoden funktioniert, die Bewertungen aller Spielzustände im kleinen Zustandsraum so zu erlernen, dass eine Spielfunktion entsteht, mit deren Hilfe ein künstlicher Agent das Spiel spielen kann und dabei durchaus gegen seine Gegner besteht.

Im Kapitel zu „4-Gewinnt“ wird deutlich, dass die RL-Technik alleine allerdings auch Grenzen

hat. So ist zum Beispiel erkennbar geworden, dass man zwar Endspielsituationen bewerten kann, dass aber ein trainierter Agent nicht unbedingt gegen perfekt spielende algorithmische Gegner konkurrieren kann. Allerdings ist solch ein Agent durchaus in der Lage gegen zufallsgesteuerte Spieler zu bestehen, die von sich aus vollkommen willkürlich spielen. Das wiederum zeigt aber, dass die Verwendung einer Lernstrategie in die richtige Richtung zur Erzeugung eines starken, selbsttrainierten Agenten führt.

Insgesamt lohnt sich die wissenschaftlich, experimentelle Auseinandersetzung mit Reinforcement Learning zum Erlernen strategischer Brettspiele, da sie deutlich macht, dass es innerhalb einer künstlichen Struktur durchaus Lernprozesse und -fortschritte geben kann. Es kommt allerdings – wie sooft – darauf an, wie diese Entwicklungen gesteuert, analysiert und interpretiert werden, damit sie zum Nutzen werden und nicht an Bedeutung verlieren. Im Spielekontext heißt das, dass jede Situationsbewertung analysiert werden muss und oftmals Einschränkungen oder Spezialisierungen notwendig werden, um erkennbaren Erfolg zu erzielen. Das Hauptziel, eine nutzbare und möglichst starke Spielfunktion zu erzeugen, mit deren Hilfe man Spiele spielen kann, ist hier zwar nicht vollständig gelungen, bleibt aber bei Umsetzung der Entwicklungsmöglichkeiten durchaus möglich. So zeigt diese Arbeit Chancen, aber auch Grenzen der RL-Strategie und verdeutlicht die größten Probleme, auf die man bei der Entwicklung einer Spielfunktion stoßen kann. Es liegt an weiteren Untersuchungen, ob es praktisch möglich ist, einen guten – in endlicher Zeit lernenden – Spieleagenten zu entwickeln.

5 Fazit und Ausblick

5.1 Fazit zu Arbeitsergebnissen

Diese Arbeit sollte die Lernfähigkeit des Strategiespiels „4-Gewinnt“ mit Hilfe der RL-Technik analysieren. Daher wird in den ersten Kapiteln die Theorie von Reinforcement Learning Prozessen beschrieben und auf die Grundlagen verschiedener Strategiespiele eingegangen.

Das eigentliche Ziel konnte nicht eindeutig gelöst werden. Es sollte gezeigt werden, dass ein künstlicher Computerspieler durchaus in der Lage ist, ein Strategiespiel zu erlernen. Hier ist das Spiel „4-Gewinnt“ in den Mittelpunkt der Betrachtung gerückt. Allerdings ist es aufgrund des großen Zustandsraum aller Spielsituationen nicht möglich, eine geeignete Spielfunktion über einfache Codierungen zu erstellen. Am Spiel „Nimm-3“ wurde jedoch deutlich, dass eine Nutzung der RL-Strategie durchaus lohnenswert ist. Über verschiedene Modelle konnte eine perfekte Spielfunktion erzeugt werden, die das Spiel erfolgreich codiert beziehungsweise in der Lage ist, gut genug gegen andere starke Spielgegner zu spielen.

Die verschiedenen Tests der Lernprozesse und Auswertungen zur Erlernbarkeit von „4-Gewinnt“ verdeutlichen eine Schwäche beim Erfassen aller möglichen und relevanten Spielzustände. Selbst Tests an Endspielsituationen brachten keine genügende Einschränkung des Zustandsraumes, um abschließende Spielzüge in „4-Gewinnt“ zu erlernen und deren Nutzen geeignet zu bewerten. Man erkennt zwar Lernfortschritte des Agenten; diese werden aber nur deutlich, wenn der Agent gegen einen zufallsgesteuerten Gegner spielt, der selber keine starke Spielweise besitzt. Spielt der Lernagent gegen einen perfekt spielenden MinMaxAgenten, wird die noch vorhandene Schwäche des Lernagenten deutlich. Er verliert circa 90% oder mehr der Testspiele. Die verschiedenen Gründe sind in den vorangegangenen Kapiteln erläutert worden. Sicher gibt es noch weitere Analysemöglichkeiten und Tests, die weiterführende Ergebnisse liefern würden. Die Durchführung dieser Tests ist im Rahmen dieser Arbeit, vor allem aus zeitlichen Gründen, nicht möglich gewesen.

Die kleinen, aber vorhandenen Lernfortschritte ermutigen dennoch zur Weiterführung dieser Arbeit. Sie begründen auch den lohnenswerten Aufwand, sich dieser Thematik zu stellen und sich mit RL auseinanderzusetzen. Der Umfang dieser schriftlichen Arbeit verdeutlicht nicht unbedingt den Arbeitsaufwand, da er zwar die Ergebnisse diverser Tests und Untersuchungen, aber nur wenig Entwicklungsschritte und Denkprozesse beschreibt, die aber dennoch durchgeführt worden sind. Im Zusammenhang mit Vorarbeiten beziehungsweise ähnlichen, schriftlichen Arbeiten wie Stenmarks Master These oder den Diplomarbeiten von Wende und Rudolph bietet diese Arbeit sicher eine gute Grundlage für weitere Forschungen.

5.2 Ausblick und Erweiterungsmöglichkeiten

Wie schon erwähnt, ist es aus zeitlichen Gründen häufig nicht möglich gewesen, tiefere Analysen bestimmter Zusammenhänge durchzuführen. Dies soll im Weiteren nicht mehr erwähnt werden, dennoch geben die folgenden Abschnitte einen Überblick über mögliche Weiterentwicklungen.

5.2.1 Alternativen der Spielcodierung, Entwicklung neuer Features

Diese Arbeit setzt sich praktisch mit zwei Codierungsalternativen auseinander. Sie werden getestet und analysiert, erbringen aber nicht den gewünschten Lerneffekt. Das Kapitel 5.3.5 beschreibt aus theoretischer Sicht die Möglichkeit Muster als Features zu entwickeln und diese als Eingabevektor dem neuronalen Netz anzubieten. Dies könnte zentraler Inhalt einer weiteren wissenschaftlichen Arbeit sein, da die Entwicklung spielrelevanter Muster eine wichtige und interessante Möglichkeit der Codierung darstellt, die – zumindest aus theoretischen Überlegungen heraus – Erfolg versprechend ist. Der Aufwand, die wichtigen und richtigen Muster zu finden, ist aber sehr hoch. Der Prozess bedarf einer zeitintensiven Vorarbeit und einer guten Analyse und Evaluation. Die Codierung der Muster in einen geeigneten Eingabevektor ist wiederum relativ schnell zu bewältigen.

Auch hier helfen Vorarbeiten wie Rudolphs und Wendes Diplomarbeiten zur Entwicklung einer Spielumgebung oder die Bachelorarbeit von Anna Klassen, die die gerade beschriebenen Überlegungen zur Generierung verschiedener Muster thematisiert.

5.2.2 Entwicklung aussagekräftiger Spielsituationen zu Testzwecken

In dieser Arbeit wird anhand einer Endspielsituation die Lernfähigkeit und die daraus resultierende Spielstärke eines Agenten getestet. Das Endspiel entsteht aus einem teilweise belegten Spielbrett, was noch 14 freie Spielfelder bietet und beiden Spielern Gewinnchancen offen lässt. Es stellt sich aber heraus, dass es zum objektiven Testen des Lernagenten eher wenig geeignet ist, da es dem MinMaxAgenten zu große Chancen einräumt, das Spiel zu beenden. Somit kann die eigentliche Stärke des Lernagenten nicht wirklich bestimmt werden.

Aus dieser Problematik ergibt sich aber die Möglichkeit, weitere Endspielsituationen zu entwickeln, die als Datengrundlage zum Lernen und auch zum Evaluieren des Lernagenten dienen. Durch eine Variation an Endspielsituationen kann auch die allgemeine Stärke des Agenten gefördert werden, da er auf mehr unterschiedliche Spielsituation (und -verläufe) trainiert wird und so möglicherweise stärkere Ergebnisse hervorbringt. Mehrere Endspielsituationen stellen aber auch eine wesentlich bessere Testumgebung dar.

5.2.3 Methodenentwicklung zur Lernen aus Spielbegegnungen

Es besteht zurzeit noch nicht die Möglichkeit, dass der Lernagent aus den Spielbegegnungen mit anderen Agenten oder menschlichen Spielern lernt. In dieser Arbeit lernt der Agent nur durch das Durchführen vieler Spiele gegen sich selbst, indem er beide Seiten simuliert und beide Spieler aufgrund gleicher Entscheidungsfindung reagieren. Es ist aber durchaus denkbar, dass durch Entwicklung weiterer Methoden die Alternative entsteht, dass auch aus Spielen gegen andere Agenten gelernt werden kann. Es sollte einen „human-computer-play“- sowie einen „minmax-computer-play“-Algorithmus geben, bei dem der zweite Spieler – der Gegenspieler des Lernagenten – entweder der menschliche Spieler oder der MinMaxAgent ist.

So variieren auch die Lernprozesse, da aufgrund anderer Spielzüge auch ganz andere Spielverläufe entstehen, auf die sich der Lernagent erst einmal wieder einstellen muss – er muss also neue Zusammenhänge und Situationsbewertungen erlernen.

5.2.4 Alternative Testverfahren zur Evaluation der Theorie

Abschließend scheint eine alternative Testumgebung eine gute Entwicklung weiterer Arbeiten zu sein. Es vergeht viel wertvolle Zeit durch Testen diverser Einstellungen und Parameterkonstellationen. Eine automatisierte Testumgebung, die ja nach Eingabe des Benutzers mehrere Tests mit gleichen Einstellungen durchführt oder Tests mit unterschiedlichen Parametern wiederholt, ist eine wünschenswerte und zeitsparende Entwicklungsmöglichkeit. Eine geeignete Darstellung der Testergebnisse in der gleichen Entwicklungsumgebung, in der auch die Tests durchgeführt werden, vereinfacht die Auswertung und Erklärung der Ergebnisse. Andernfalls ist immer ein manuelles Kopieren und Einstellen der Tests sowie der Ergebnisse notwendig, um Zusammenhänge (Diagramme, Tabellen, Grafiken) geeignet darzustellen.

Zum Beispiel wäre die Entwicklung einer grafischen Darstellung des Trainingsnetzfehlers sinnvoll, da sie direkt in der Entwicklungsumgebung die Auswertung der Netzperformance deutlich macht, und ein Eingreifen durch Änderung der Parameter schnell möglich ist. Zeitmessungen von Testdurchläufen und Testspielergebnisse in Form von Siegquoten oder Ähnlichem sollten nicht mehr durch externe Programme erstellt werden müssen, sondern können direkt in der gleichen Entwicklungsumgebung gemessen und dargestellt werden, in der auch das Programm zur Durchführung der Lernprozesse und Tests verwaltet wird.

5.3 Erfahrungswerte

5.3.1 „Lessons Learned“ und Fehlerkorrekturen

Es gibt zu viele Lerneffekte, die sich auf die persönliche Arbeit und den Umgang mit einer wissenschaftlichen Thematik, wie dieser beziehen, um sie alle hier zu nennen. Dennoch sollen die wichtigsten Richtungsänderungen, Fehlerkorrekturen und auch gegangene Sackgassen genannt werden, um sie Nachfolgern in diesem Thema zu ersparen.

So wurden zum Beispiel wesentliche Fehler, aber vor allem fehlende Betrachtungen oder Berechnung am Pseudocode von Sutton/Barto in der letztlichen Implementierung, die zu dieser Arbeit gehört, korrigiert. Kapitel 4.2.2.4 beschreibt recht ausführlich die Problematik fehlender Bewertung der Spielterminierung. Dabei wurde der Fehler korrigiert, dass der RL-Agent ein Spiel nicht unbedingt eigenständig beenden konnte. Es fehlte die Untersuchung des Rewards zum Zeitpunkt T (terminierender Spielzug zum Spielende). Fehlt dieser Reward-Wert aber, kann allein durch die trainierte Spielfunktion unter Umständen nicht der terminierende Spielzug gefunden werden, da die Spielterminierung nicht trainiert wird. Solche und weitere kleine Korrekturen, die sich hauptsächlich auf die Programmierlogig beziehen, wurden behoben. Orientiert man sich an den implementierten Methoden der RL-Strategie bei „4-Gewinnt“, erkennt man im Vergleich zum entsprechenden Pseudocode von Sutton/Barto kleine Unterscheide.

Eine weitere Überlegung, die so sicher nicht zum Ziel einer guten Spielfunktion führt, ist die Tatsache, ganze „4-Gewinnt“-Spiele zu trainieren beziehungsweise als Lerngrundlage für den RL-Agenten zu nehmen. Auch aus Kommentaren der einzelnen Kapitel wird deutlich, dass der Zustandsraum, der mögliche relevante Spielsituationen beinhaltet, bei einem kompletten „4-Gewinnt“-Spiel viel zu groß ist, als dass er erlernt und damit bewertet werden könnte. Beim Spiel „Nimm-3“, was an sich noch einen kleinen Zustandsraum hat, ist dies noch kein Problem. Bei „4-Gewinnt“ nimmt diese Problematik aber schnell nicht erfassbare Dimensionen an. Daher muss man sich auf Teilprobleme des Spiels beschränken oder die Features der Spielfunktion so optimieren, dass sie in der Lage sind, nur die wichtigsten und relevanten Informationen zu codieren.

Abschließend lässt sich noch festhalten, dass eine gute Struktur der Trainings- und Testumgebung viel Zeit einsparen kann. Dabei nimmt die Aufbereitung der einzelnen Ergebnisse oft die meiste Zeit in Anspruch, was durch automatisierte Methoden erleichtert werden kann. So sollte man sich – hier ist das nur teilweise geschehen – schon vorab Gedanken zur vollständigen Programmstruktur und modularen Bauweise des Programms machen, damit nachträgliche Änderungen oder Ergänzungen auch möglich bleiben.

5.3.2 Persönliches Fazit

Die Auseinandersetzung mit dem Thema klingt zunächst spannend und interessant. Das ändert sich auch nicht, wenn man sich näher damit beschäftigt. Dennoch wird gerade der praktische Test entwickelter Methoden und Möglichkeiten zur Geduldsprobe und oftmals auch zum Nervenspiel. Geht es doch darum, theoretisch formulierte Ergebnisse zu validieren und nachzuvollziehen. Erst der praktische Test einer Methodik sagt schließlich etwas über ihre Verwendbarkeit und den Nutzen aus.

Die Tatsache, dass RL als Methodik im Zusammenhang des Erlernens strategischer Spiele schon untersucht worden ist, stellte sich als weniger starkes Hindernis heraus als gedacht. Stenmark, der in seiner Master-These zwei Methoden miteinander vergleicht (unter anderem RL) setzt seinen Schwerpunkt anders, als es diese Arbeit tut. Tesauro, der mit seinem Weltklasse-Backgammonspiel, welches mittels RL trainiert wurde, einen tollen Anreiz bietet, sich mit der Thematik auseinanderzusetzen, untersuchte seinerzeit ein anderes Strategiespiel, als diese Arbeit. Daher stellt auch diese schon vorhandene Arbeit keinen Grund dar, RL nicht als Untersuchung in den Mittelpunkt von Lernstrategien für strategische Brettspiele zu stellen.

Die Bearbeitung des Themas brachte nun zwei verschiedene Schwerpunkte zum Vorschein: Die Analyse strategischer Brettspiele im Allgemeinen und „4-Gewinnt“ im Speziellen und das Verständnis einer durchaus mächtigen Lernstrategie „Reinforcement Learning“, die auf dieses Feld der Strategiespiele angewandt werden sollte. Letzteres wurde zum zentralen Thema dieser Arbeit, da schnell deutlich wurde, dass eine detaillierte Analyse des Spiels „4-Gewinnt“ mit samt seiner Möglichkeiten, Strategien und Taktiken nicht nur zeitlich schwierig wird, sondern einfach schon häufig untersucht wurde und daher kein neues, unerforschtes Gebiet darstellt. So widmet sich diese Arbeit der Untersuchung von RL auf die Verwendbarkeit für Strategiespiele.

Persönlich hätte ich gerne mehr Zeit gehabt, um diese Arbeit auch im Detail abzurunden und vermehrt Querverweise und Bezüge zu anderen, dennoch verwandten Themengebieten darzustellen. Wie so oft, ist aber auch hier die Zeit ein Grund dafür, dass sich die Arbeit doch spezieller und mit weniger großer Wirkung mit einzelnen Problemstellungen auseinandersetzt und versucht diese, im Detail zu verstehen, zu analysieren und letztlich auch zu erklären.

Ist die Aufgabe, „4-Gewinnt“ erlernbar zu machen, mit dieser Arbeit noch nicht abgeschlossen, wird sie aber durch diese Ausarbeitung zumindest in grundlegenden Gedanken begonnen und die Dimension dieser Problematik dargestellt.

So kann ich schließlich auch selber weitgehend zufrieden mit der Bearbeitung und dem Ergebnis dieser Arbeit sein und mich daran freuen, dass sie hoffentlich zur Grundlage späterer Forschungen oder Ausarbeitungen wird.

A Anhang

A.1 Entwicklung der tabellarischen Spielfunktion für „Nimm-3“

Im Folgenden wird der Aufbau der tabellarischen Spielfunktion mittels Veränderung der Tabelleneinträge gezeigt. Dieser Aufbau entspricht den allgemeingültigen Werten einer perfekten Spielkodierung, kann aber im Erzeugungsweg unterschiedlich sein. Das liegt an den zufälligen Spielzügen, die zu Bewertungen solcher Spielsituationen führen, die mittels fest kodiertem Regelwerk nicht erreicht und damit auch nie bewertet würden. Einige Erläuterungen folgen im Anschluss an die tabellarischen Abbildungen.

SPIELFUNKTION (in Tabellenform)

Nach 1 Spiel:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	0	3	0
2	0	2	0
1	0	1	**100**
0	0	0	0

Nach 2 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	0	3	0
2	0	2	0
1	**-100**	1	**100**
0	0	0	0

Nach 3 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	0	3	0
2	0	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 4 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	0	3	0
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 5 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	0
3	**-100**	3	0
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 6 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	0	6	0
5	0	5	0
4	0	4	**-100**
3	**-100**	3	0
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 7 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	**-100**	6	0
5	0	5	0
4	0	4	**-100**
3	**-100**	3	0
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 8 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	**-100**	6	0
5	**-100**	5	0
4	0	4	**-100**
3	**-100**	3	0
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 9 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	**-100**	6	0
5	**-100**	5	0
4	0	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 10 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	0
6	**-100**	6	0
5	**-100**	5	0
4	**100**	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 11 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	**100**
6	**-100**	6	0
5	**-100**	5	0
4	**100**	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 12 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	**100**
6	**-100**	6	**100**
5	**-100**	5	0
4	**100**	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 13 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	0	8	0
7	0	7	**100**
6	**-100**	6	**100**
5	**-100**	5	**100**
4	**100**	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 14 Spielen:

WEISS		SCHWARZ	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	0
10	0	10	0
9	0	9	0
8	**100**	8	0
7	0	7	**100**
6	**-100**	6	**100**
5	**-100**	5	**100**
4	**100**	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

Nach 15 Spielen:

WEISS		**SCHWARZ**	
Anzahl hinterlassende Steine	Funktionswert durch RL	Anzahl hinterlassende Steine	Funktionswert durch RL
11	0	11	**100**
10	0	10	0
9	0	9	0
8	**100**	8	0
7	0	7	**100**
6	**-100**	6	**100**
5	**-100**	5	**100**
4	**100**	4	**-100**
3	**-100**	3	**100**
2	**-100**	2	**100**
1	**-100**	1	**100**
0	0	0	0

. . .

Erläuterungen:

Die Spielfunktion ist daher letztendlich nicht vollständig erzeugt worden, da die anfänglichen Spielzüge nur sehr selten per Zufallszug ausgewählt worden sind, was aber für eine Bewertung dieser Situation notwendig wäre. Da aber bei den anfänglichen Zügen kaum zufällige Zugentscheidungen vorkommen, ist eine Bewertung dieser Situationen unmöglich geworden. Da gerade zu Beginn des Spiels von beiden Spielern immer gleiche Züge gemacht worden sind, wurde nur die Spielfunktion der späteren Spielsituationen erstellt (was aber hier ausreicht, um eine eindeutige und perfekte Spielkodierung zu erreichen).

Die Nummerierung der durchgeführten Spiele entspricht nicht den realen Versuchswerten, wurde hier aber aus Platzgründen vorgenommen. Durch Zufallszüge werden manche Spiele „umsonst“ – das heißt ohne neue Bewertungen – gespielt. Diese Zufallszüge sind aber notwendig, um Spielzüge zu generieren, die durch die Wahl der bestbewerteten Zugmöglichkeiten nie ausgewählt würden. Dies wiederum lässt abwechselnde Spielverläufe entstehen, damit nicht nur bekannte Situationen bewertet werden, sondern auch solche, die nicht häufig und regelmäßig gespielt werden. Wird aber ein Zug zufällig ausgewählt, erhält die entstehende Situation keine neue Bewertung, somit wird auch nichts verändert.

A.2 Kantenanpassungen des linearen Netz bei Nimm-3

Initialisierte Kanten		
weights_in[0][0]	=	0,0000000000
weights_in[1][0]	=	0,0000000000
weights_in[2][0]	=	0,0000000000
weights_in[3][0]	=	0,0000000000
weights_in[4][0]	=	0,0000000000
weights_in[5][0]	=	0,0000000000
weights_in[6][0]	=	0,0000000000
weights_in[7][0]	=	0,0000000000
weights_in[8][0]	=	0,0000000000
weights_in[9][0]	=	0,0000000000
weights_in[10][0]	=	0,0000000000
weights_in[11][0]	=	0,0000000000
weights_in[12][0]	=	0,0000000000
weights_in[0][1]	=	0,0000000000
weights_in[1][1]	=	0,0000000000
weights_in[2][1]	=	0,0000000000
weights_in[3][1]	=	0,0000000000
weights_in[4][1]	=	0,0000000000
weights_in[5][1]	=	0,0000000000
weights_in[6][1]	=	0,0000000000
weights_in[7][1]	=	0,0000000000
weights_in[8][1]	=	0,0000000000
weights_in[9][1]	=	0,0000000000
weights_in[10][1]	=	0,0000000000
weights_in[11][1]	=	0,0000000000
weights_in[12][1]	=	0,0000000000
weights_out[0][0]	=	0,0000000000
weights_out[1][0]	=	0,0000000000

Nach Spiel 1		
weights_in[0][0]	=	0,0000000000
weights_in[1][0]	=	0,0000000000
weights_in[2][0]	=	0,0000000000
weights_in[3][0]	=	0,0000000000
weights_in[4][0]	=	0,0000000000
weights_in[5][0]	=	0,0000000000
weights_in[6][0]	=	0,0000000000
weights_in[7][0]	=	0,0000000000
weights_in[8][0]	=	0,0000000000
weights_in[9][0]	=	0,0000000000
weights_in[10][0]	=	0,0000000000
weights_in[11][0]	=	0,0000000000
weights_in[12][0]	=	0,0000000000
weights_in[0][1]	=	0,0000000000
weights_in[1][1]	=	0,0000000000
weights_in[2][1]	=	0,0000000000
weights_in[3][1]	=	0,0000000000
weights_in[4][1]	=	0,0000000000
weights_in[5][1]	=	0,0000000000
weights_in[6][1]	=	0,0000000000
weights_in[7][1]	=	0,0000000000
weights_in[8][1]	=	0,0000000000
weights_in[9][1]	=	0,0000000000
weights_in[10][1]	=	0,0000000000
weights_in[11][1]	=	0,0000000000
weights_in[12][1]	=	0,0000000000
weights_out[0][0]	=	0,1250000000
weights_out[1][0]	=	0,1250000000

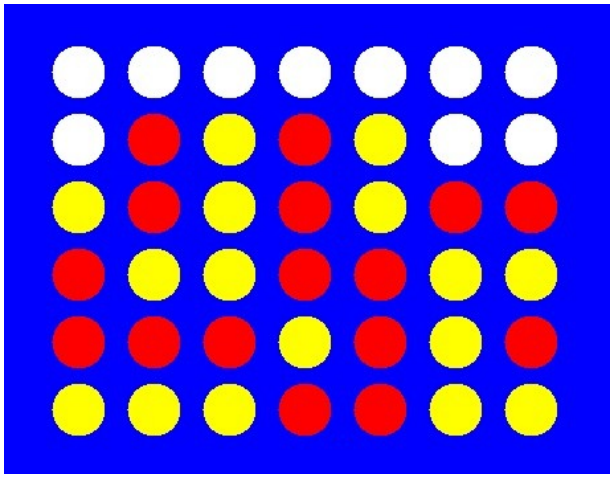
Nach Spiel 2		
weights_in[0][0]	=	0,0000000000
weights_in[1][0]	=	0,0077820617
weights_in[2][0]	=	0,0000000000
weights_in[3][0]	=	0,0000000000
weights_in[4][0]	=	0,0000000000
weights_in[5][0]	=	0,0000000000
weights_in[6][0]	=	0,0000000000
weights_in[7][0]	=	0,0000000000
weights_in[8][0]	=	0,0000000000
weights_in[9][0]	=	0,0000000000
weights_in[10][0]	=	0,0000000000
weights_in[11][0]	=	0,0000000000
weights_in[12][0]	=	-0,0077820617
weights_in[0][1]	=	0,0000000000
weights_in[1][1]	=	0,0077820617
weights_in[2][1]	=	0,0000000000
weights_in[3][1]	=	0,0000000000
weights_in[4][1]	=	0,0000000000
weights_in[5][1]	=	0,0000000000
weights_in[6][1]	=	0,0000000000
weights_in[7][1]	=	0,0000000000
weights_in[8][1]	=	0,0000000000
weights_in[9][1]	=	0,0000000000
weights_in[10][1]	=	0,0000000000
weights_in[11][1]	=	0,0000000000
weights_in[12][1]	=	-0,0077820617
weights_out[0][0]	=	0,2495129875
weights_out[1][0]	=	0,2495129875

Nach Spiel 3		
weights_in[0][0]	=	0,0000000000
weights_in[1][0]	=	-0,0075668001
weights_in[2][0]	=	0,0000000000
weights_in[3][0]	=	-0,0000036134
weights_in[4][0]	=	0,0000072877
weights_in[5][0]	=	-0,0000072961
weights_in[6][0]	=	0,0000073015
weights_in[7][0]	=	-0,0000073099
weights_in[8][0]	=	0,0000073153
weights_in[9][0]	=	-0,0000073238
weights_in[10][0]	=	0,0000073291
weights_in[11][0]	=	-0,0000073376
weights_in[12][0]	=	-0,0230688092
weights_in[0][1]	=	0,0000000000
weights_in[1][1]	=	-0,0075668001
weights_in[2][1]	=	0,0000000000
weights_in[3][1]	=	-0,0000036134
weights_in[4][1]	=	0,0000072877
weights_in[5][1]	=	-0,0000072961
weights_in[6][1]	=	0,0000073015
weights_in[7][1]	=	-0,0000073099
weights_in[8][1]	=	0,0000073153
weights_in[9][1]	=	-0,0000073238
weights_in[10][1]	=	0,0000073291
weights_in[11][1]	=	-0,0000073376
weights_in[12][1]	=	-0,0230688092
weights_out[0][0]	=	0,1264321471
weights_out[1][0]	=	0,1264321471

A.3 Auswertungen der Spielqualität des „4-Gewinnt“-Agenten

A.3.A Ergebnisse Codierung 1 bei Endspielvariante 1

Bei Tests der ersten Codierungsalternative (126 Eingabeneuronen) an Endspiel 1 (siehe Bild) wurde folgendes Ergebnis erzielt:

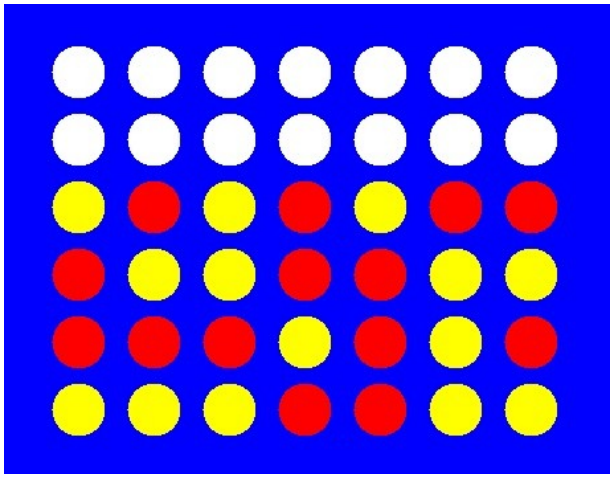


```
Trainingsspiele: 10000
gegen den Random-Agenten

Anzahl Eingabeneuronen: 126
Anzahl verdeckte Neuronen: 10
Anzahl Ausgabeneuronen: 1

Anzahl Testspiele: 1000, davon:
     Gewonnen: 1000
     Verloren: 0
     Unendschieden: 0

     Gewinnquote: 100%
```

Die Gewinnquote liegt bei 100%, da stets der rote senkrechte Vierer in der mittleren Spalte zum Erfolg und damit zum Spielgewinn führt. Da direkt der erste Spielzug das Spiel beendet, kann auch kein Lernschritt durchgeführt werden. Diese Tests dienen der Überprüfung auf korrekte Implementierung der Lern-, Simulations- und Spielalgorithmen des Agenten.

A.3.B Ergebnisse Codierung 1 bei Endspielvariante 1b

Bei Tests der ersten Codierungsvarianten am ausführlicheren Eindspiel 1b wurden die folgenden Ergebnisse erzielt:



```
a.)
Trainingsspiele: 10000
gegen den Random-Agenten

Anzahl Eingabeneuronen: 126
Anzahl verdeckte Neuronen: 10
Anzahl Ausgabeneuronen: 1

Anzahl Testspiele: 1000, davon:
     Gewonnen: 849
     Verloren: 151
     Unendschieden: 0

     Gewinnquote: 84.9%
```

```
b.)                                     c.)
Trainingsspiele: 50000                  Trainingsspiele: 50000
gegen den Random-Agenten                gegen den MinMax-Agenten

Anzahl Eingabeneuronen: 126             Anzahl Eingabeneuronen: 126
Anzahl verdeckte Neuronen: 10           Anzahl verdeckte Neuronen: 10
Anzahl Ausgabeneuronen: 1               Anzahl Ausgabeneuronen: 1

           Anzahl Testspiele: 1000, davon (je Test):
Gewonnen: 917                           Gewonnen: 116
Verloren: 83                            Verloren: 883
Unendschieden: 0                        Unendschieden: 1

Gewinnquote: 91.7%                      Gewinnquote: 11%
```

Bei diesen Tests – die nur eine (im Grunde durchschnittliche) Übersicht aller durchgeführten Tests darstellen – wird deutlich, dass Siege gegen den zufallsgesteuerten Agenten wahrscheinlicher werden, je länger der RL_NNAgent trainiert wird. Das liegt mit großer Wahrscheinlichkeit daran, dass bei mehr Trainingsspielen, vermehrt siegrelevante Zustände trainiert werden. Dennoch (auch nach längerer Trainingsphase) hat der lernende Agent keine große Siegchance gegen den MinMaxAgenten, der durch seine Berechnungen einfach stärker ist und daher schneller beziehungsweise häufiger gewinnt.

A.3.C Ergebnisse Codierung 2 bei Endspielvariante 1b

Hier die Ergebnisse der Testspiele bei Verwendung der zweiten Codierungsalternative:

```
a.)                                     b.)
Trainingsspiele: 10000                  Trainingsspiele: 50000
gegen den Random-Agenten                gegen den Random-Agenten

Anzahl Eingabeneuronen: 42              Anzahl Eingabeneuronen: 42
Anzahl verdeckte Neuronen: 10           Anzahl verdeckte Neuronen: 10
Anzahl Ausgabeneuronen: 1               Anzahl Ausgabeneuronen: 1

           Anzahl Testspiele: 1000, davon (je Test):
Gewonnen: 831                           Gewonnen: 942
Verloren: 169                           Verloren: 58
Unendschieden: 0                        Unendschieden: 0

Gewinnquote: 83.1%                      Gewinnquote: 94.2%


c.)
Trainingsspiele: 50000        Testspiele: 1000, davon:
gegen den MinMax-Agenten
                                    Gewonnen: 31
Anzahl Eingabeneuronen: 42          Verloren: 960
Anzahl verdeckte Neuronen: 10       Unentschieden: 9
Anzahl Ausgabeneuronen: 1           Gewinnquote: 3.1%
```

A.3.D Ergebnisübersicht aller durchgeführten Tests

Die nachfolgende Tabellen zeigen eine organisierte Übersicht aller durchgeführten Tests.

Dabei wird zwischen vier verschiedenen Testeinstellungen unterschieden:

1. Spielbrett über 126 Eingabeneuronen codiert gegen den RandomAgenten
2. Spielbrett über 126 Eingabeneuronen codiert gegen den MinMaxAgenten
3. Spielbrett über 42 Eingabeneuronen codiert gegen den RandomAgenten
4. Spielbrett über 42 Eingabeneuronen codiert gegen den MinMaxAgenten

Alle Einstellungsvarianten werden mit dreifacher Durchführung getestet. Außerdem wird zwischen drei verschiedenen Anzahlen an Trainingsspielen unterschieden. Aus den Tests ergeben sich Durchschnittsquoten für die Siegchance des Lernagenten.

Parameter			**Ø-Quote bei Codierung I (126N)**		**Ø-Quote bei Codierung II (42N)**	
Testlauf	**Anzahl Trainingsspiele**	**Anzahl Testspiele**	**(1) gegen Random**	**(2) gegen MinMax**	**(3) gegen Random**	**(4) gegen MinMax**
1	10000	1000	86,93	0,83	86,6	55,6
2	50000	1000	81,77	1,23	86,7	0,9
3	100000	1000	88,93	12,3	86,9	33,97

Diese Tabelle zeigt detailliert die einzelnen Testergebnisse und die Zeitintervalle aller Tests:

Einzelne Testdurchläufe zum Ermitteln des Durchschnittes												
	Testspiele bei 126N (Cod.I) gegen Random			Testspiele bei 126N (Cod.I) gegen MinMax			Testspiele bei 42N (Cod. II) gegen Random			Testspiele bei 42N (Cod. II) gegen MinMax		
Zu 1	86,1	89,6	85,1	0,7	0,7	1,1	85,8	91,6	82,4	33,3	77,7	55,8
Zu 2	76	83,6	85,7	3,1	0,4	0,2	94,6	80,5	85	1,8	0,4	0,5
Zu 3	91,5	85	90,3	1,1	1,9	33,9	83,4	91,4	85,9	5,9	88,9	7,1
Zeitintervalle der einzelnen Durchläufe und Durchschnittswerte												
Zeit zu 1:	70,62	73,27	79,49	78,53	75,56	78,09	31,98	48,6	64,72	36,8	34,44	31,73
Ø-Zeit		**74,46**			**77,39**			**48,43**			**34,32**	
Zeit zu 2:	298,37	312,75	301,72	273,51	297,64	258,04	172,88	189,45	180,92	162,7	157,62	171,79
Ø-Zeit		**304,28**			**276,4**			**181,08**			**164,04**	
Zeit zu 3:	560,82	554,67	542,87	609,84	542,95	531,29	363,92	327,97	344,21	329,07	353,68	306,74
Ø-Zeit		**552,79**			**561,36**			**345,37**			**329,83**	

Aus den einzelnen Quoten der Testspiele ergeben sich die Durchschnittsquoten der ersten Tabelle. Die Ergebnisse werden in den untergeordneten Kapiteln von Kapitel 5.3 analysiert.

B Appendix

B.1 Abbildungsverzeichnis

Abbildung 1: Gitternetz für Roboter....9
Abbildung 2: Zugmöglichkeit beim Spiel Nimm-3....14
Abbildung 3: perfekte Zugstrategie zum Spielgewinn....15
Abbildung 4: Tabellarische Spielfunktion initiiert....17
Abbildung 5: Tabellarische Spielfunktion (ein Spieldurchlauf)....20
Abbildung 6: Auszug der Belegungsalternativen pro Spielspalte....22
Abbildung 7: Struktur des linearen Netz....24
Abbildung 8: Kantengewichtung nach einem Spiel....28
Abbildung 9: Kantengewichtung nach fünf Spielen....28
Abbildung 10: Kantengewichtung nach 68 Spielen....29
Abbildung 11: Kantengewichtung nach 100 Spielen....31
Abbildung 12: Struktur des neuronalen Netz....33
Abbildung 13: Kantengewichtungen nach 50.000 Spielen....35
Abbildung 14: Spielbrett von 4-Gewinnt mit Spielsteinen....40
Abbildung 15: mögliche Vierer bei Spielstein unten rechts....41
Abbildung 16: mögliche Vierer bei Spielstein unten mittig....41
Abbildung 17: Erste Endspielsituation (trivial)....50
Abbildung 18: Erweiterte Endspielsituation....51
Abbildung 19: Weitere Tests des RL-Agenten gegen den IBEF-Agenten....54

B.2 Literatur- und Quellenverzeichnis

Literaturquellen:

Konen (2006):
Konen, W.: *Reinforcement Learning für Brettspiele: Der Temporal Difference Algorithmus*, Technischer Bericht, Fachhochschule Köln Campus Gummersbach, 2006
Internetlink: http://www.gm.fh-koeln.de/~konen/Publikationen/TR_TDlambda.pdf, (01.10.2008)

Markelic (2004):
Markelic, I.: *Reinforcement Learning als Methode zur Entscheidungsfindung beim simulierten Roboterfußball*, Studienarbeit, Universität Koblenz-Landau, 2004
Internetlink: http://www.uni-koblenz.de/~fruit/ftp/teaching/mar04-studienarbeit.pdf, (01.10.2008)

Stenmark (2005):
Stenmark, M.: *Synthesizing Board Evaluation Functions for Connect4 using Machine Learning Techniques*, Master-These, Østfold University College, 2005
Internetlink:http://www.hiof.no/neted/upload/attachment/site/group12/Martin_Stenmark_Synthesizing_Board_Evaluation_Functions_for_Connect4_using_Machine_Learning_Techniques.pdf, (01.10.2008)

Sutton/Barto (1998):
Sutton, R. S., Barto, A. G.: *Reinforcement Learning: An Introduction*, MIT Press, Cambridge, MA, 1998
Internetlink: http://www.cs.ualberta.ca/~sutton/book/the-book.html, (29.08.2008)

Sutton/Bonde (1992):
Sutton, R. S., Bonde, A. Jr.: *Nonlinear TD/Backprop pseudo C-code*, GTE Laboratories, 1992
Internetlink: http://www.cs.ualberta.ca/~sutton/td-backprop-pseudo-code.text, (29.08.2008)

Tesauro (1992):
Tesauro, G. J.: *Practical issues in temporal difference learning.* Machine Learning - Volume 8, Issue 3-4 (May 1992), Pages: 257-277, 1992

Zusätzliche Internet-Quellen:

Wikipedia (Gradientenverfahren): http://de.wikipedia.org/wiki/Gradientenverfahren, (29.08.2008)

B.3 Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbstständig verfasst zu haben.

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht.

Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben.

Die Arbeit hat mit gleichem Inhalt beziehungsweise in wesentlichen Teilen noch keiner Prüfungsbehörde vorgelegen.

Gummersbach, den 08. Oktober 2008

_____

Jan Philipp Schwenck