Fachhochschule Köln
Cologne University of Applied Sciences

Reinforcement Learning mit N-Tupel-Systemen für Vier Gewinnt

Bachelorarbeit

ausgearbeitet von

Markus Thill

zur Erlangung des akademischen Grades
Bachelor of Science (B.Sc.)

vorgelegt an der
Fachhochschule Köln
Campus Gummersbach
Fakultät für Informatik und
Ingenieurwissenschaften

im Studiengang
Technische Informatik

Erster Prüfer:	Prof. Dr. Wolfgang Konen Fachhochschule Köln
Zweiter Prüfer:	Prof. Dr. Heinrich Klocke Fachhochschule Köln

Gummersbach, im Juni 2012

Kurzfassung

Die Untersuchung maschineller Lernverfahren für Brettspiele stellt auch heute noch ein sehr interessantes Forschungsgebiet dar. Dies liegt vor allem daran, dass das Erlernen komplexer Spiele wie dem Schach- oder Go-Spiel nach wie vor als sehr anspruchsvoll gilt.

Während Menschen in der Lage sind, gewisse Zusammenhänge bzw. Gesetzmäßigkeiten in Spielen zu erkennen und daraus die richtigen Rückschlüsse zu ziehen, ist dies für ein Computerprogramm deutlich schwieriger. Aus diesem Grund müssen die Entwickler häufig viel spieltheoretisches Wissen in das Programm einbringen, damit der Lernprozess überhaupt fähig ist, auf die besonderen spielspezifischen Merkmale zu achten.

In dieser Arbeit wird die Anwendung von sogenannten *N-Tupel-Systemen* – in Kombination mit einer *Reinforcement-Learning*-Trainingsumgebung – auf das Spiel *Vier Gewinnt* untersucht. N-Tupel-Systeme dienen dazu, lineare Nutzenfunktionen von Agenten zu approximieren, sodass Stellungsbewertungen vorgenommen werden können. Um diese Funktionen zu erlernen, werden die N-Tupel-Systeme mithilfe des *Temporal Difference Learnings* (*TDL)*, einem Algorithmus zur Lösung von *RL*-Problemen, trainiert.

Das Training der Agenten erfolgte ausschließlich durch *Self-Play, w*ährend des Trainings kam daher kein Lehrer oder spieltheoretisches Wissen irgendeiner Form zum Einsatz. Dennoch gelang es, Agenten mit hoher Spielstärke zu trainieren, die in vielen Fällen einen perfekten Spieler schlagen konnten. Insbesondere die N-Tupel-Systeme, die eine sehr große Zahl an Features generieren und die passenden selektieren, tragen zu den außerordentlich guten Ergebnissen bei.

Inhaltsverzeichnis

Abkürzungsverzeichnis

AI	Artificial Intelligence
Abb.	Abbildung
AB-Suche	Alpha-Beta-Suche
CPU	Central Processing Unit
CSV	Comma-Separated Values
ETC	Enhanced Transposition Cutoff
GUI	Graphical User Interface
KI	Künstliche Intelligenz
Komb.	Kombination
LUT	Look-up-table
MB	Megabyte
rel.	Relativ
RL	Reinforcement Learning
RP	Random Points
RW	Random Walk
Sig.	Sigmoid
Tanh.	Tangens-Hyperbolicus
TD	Temporal Difference
TDL	Temporal Difference Learning
TTT	Tic Tac Toe
WP	Wendepunkt

Abbildungsverzeichnis

1 Einleitung und Motivation

"Das Spiel ist die höchste Form der Forschung."
(Albert Einstein)

Schon seit vielen Jahren beschäftigen sich Forscher mit der Frage, ob und inwieweit Maschinen jemals in der Lage sein werden, einen gewissen Grad an Intelligenz zu erreichen, die der menschlichen ähnlich ist. Viele Forschungszweige befassen sich mit der Entwicklung intelligenter Maschinen bzw. Computerprogrammen, die die Lösung gewisser Problemstellungen erlernen sollen. In diesem Zusammenhang wird häufig der Begriff der *künstlichen Intelligenz* (*KI*, engl.: *artificial intelligence, AI*) verwendet. Eine exakte bzw. allgemeine Definition des Begriffes ist jedoch schwierig, da bereits bei der Beschreibung von Intelligenz im Allgemeinen die verschiedensten Auffassungen vertreten werden. Der bekannte US-amerikanische Informatiker *John McCarthy* beschrieb die "*Künstliche Intelligenz*" folgendermaßen:

> *"It is the science and engineering of making intelligent machines, especially intelligent computer programs. It is related to the similar task of using computers to understand human intelligence, but AI does not have to confine itself to methods that are biologically observable."* [1] .

Insbesondere die Untersuchung und das Erlernen von strategischen Brettspielen mithilfe von Computerprogrammen hat aus Sicht der Forschung schon immer eine herausragende Rolle gespielt. Eine Vielzahl von Arbeiten – mit den verschiedensten Zielen und Lösungsansätzen – der letzten 70 Jahre beschäftigten sich ausführlich mit der Entwicklung maschineller Spieler. Vor allem das Schach-Spiel fand hierbei besondere Beachtung.

Bereits im Jahre 1769 erlangte *Wolfgang von Kempelen,* ein österreichisch-ungarischer Staatsbeamter, mit seinem *Schachtürken* Berühmtheit. Es handelte sich hierbei um eine Maschine, die angeblich mithilfe einer komplexen Mechanik in der Lage war, Schach gegen menschliche Gegner zu spielen. Erst später stellte sich heraus, dass im Inneren der Maschine ein kleinwüchsiger Mensch versteckt werden konnte, der mit einer speziellen Vorrichtung die Spielzüge ausführte [2] .

Als Autor des ersten Schachprogrammes gilt *Konrad Zuse*, der sein Programm in den Jahren 1942 bis 1945 – mit der selbst entworfenen Programmiersprache *Plankalkül* – entwickelte [3] .

Seitdem hat sich sehr vieles im Bereich der künstlichen Intelligenz getan. Einen neuen Höhepunkt erreichte die Entwicklung mit dem Sieg von *Deep Blue* [4] – einem Schachcomputer von IBM – gegen den amtierenden Schach-Weltmeister Garri Kasparow. Allerdings wurde die hohe Spielstärke mithilfe eines massiv parallelen Systems erzielt, mit dem mehr als 2 Mio. Spielstellungen pro Sekunde untersucht werden konnten.

Brettspiele sind nicht ohne Grund eines der ältesten und meist erforschten Gebiete der *KI*, einige Eigenschaften machen sie für die Forschung besonders interessant:

Die meisten strategischen Brettspiele besitzen ein überschaubares Regelwerk und ein exakt definiertes Ziel (terminierende Bedingung; unendlich lang andauernde Spiele sind nicht möglich), das sich während des Spiels in der Regel nicht verändert. Das Regelwerk kann daher meistens sehr leicht erlernt bzw. in einem Programm umgesetzt werden.

Dennoch lässt sich das angestrebte Ziel (z.B. das Schachmatt-Setzen des Gegners) häufig nicht ohne Weiteres erreichen, da hierfür zunächst eine ganze Reihe von Aktionen nötig sind; es kann daher erst nach einer Sequenz von Aktionen eine Aussage über Sieg, Niederlage und ggfs. Unentschieden getroffen werden. Zwar beschränkt das Regelwerk die Anzahl der möglichen Aktionen pro Zug, dennoch ist für nicht-triviale Spiele eine astronomisch große Zahl an möglichen Zug-Sequenzen denkbar, sodass selbst ein Rechner mit massiver Leistung nicht ohne Weiteres alle Kombinationen untersuchen kann. Aufgrund der enormen Komplexität, die viele Spiele besitzen, ist daher eine gewisse Form von Intelligenz nötig, um eine akzeptable Spielstärke zu erreichen.

Ein weiterer Punkt, in dem sich die klassischen Brettspiele (mit vereinzelten Ausnahmen) von vielen anderen Problemen unterscheiden, besteht darin, dass die komplette Umgebung (das Spielfeld) zu jedem Zeitpunkt des Spiels bekannt ist und ferner, wie einzelne Aktionen sich auf den Zustand des Spielfeldes auswirken.

Weiterhin vereinfachen die diskreten Spielzustände der meisten Brettspiele die Implementierung der verschiedenen Algorithmen (vgl. [5]).

Eine herausragende Fähigkeit des Menschen besteht darin, dass er in vielen Situationen in der Lage ist, gewisse Merkmale oder Muster zu erkennen und Gemeinsamkeiten mit anderen Situationen herzustellen. So hat ein Mensch beispielsweise keine Probleme damit, bekannte Personen auf einem Foto zu identifizieren, da er sich wesentliche Gesichtsmerkmale einprägen und bei Bedarf wieder abrufen und vergleichen kann.

Auch bei Brettspielen kann der Mensch schnell bestimmte Zusammenhänge erkennen und erfolgreich auf neue Spielsituationen anwenden. Im Bereich der *KI* ist es deutlich schwieriger, solche Muster zu erkennen und geeignet zu verknüpfen, um ein Spiel zu erlernen. Daher wurden in der Vergangenheit sehr oft Baumsuchverfahren verwendet und versucht, diese an zentralen Stellen zu optimieren (Beschneidung von Teilbäumen etc.). Allerdings können solche Programme das betrachtete Spiel nicht im

klassischen Sinne erlernen, sondern lediglich aufgrund der großen Rechengeschwindigkeit eine Vielzahl an Spielsituationen durchsuchen, um eine vermeintlich intelligente Aussage zu treffen.

Auch viele maschinelle Lernverfahren sind auf externe "Features" angewiesen, die dem Programm vorgeben, welche Merkmale für den Lernprozess von besonderer Bedeutung sind. Die Selektion und Kombination geeigneter Features ist allerdings nicht trivial und verlangt häufig ein hohes Maß an Kreativität.

Im Jahre 2008 stellte *Simon M. Lucas* [6] erstmals ein Verfahren für klassische Brettspiele vor, das eine sehr große Zahl an Features *selbstständig* erzeugt und von denen sich das Lernverfahren im Anschluss die geeignetsten auswählt. Hierbei handelt es sich um sogenannte *N-Tupel-Systeme*, die in Kombination mit einem recht bekannten Lernverfahren, dem *Temporal Difference Learning* (*TDL*), in dieser Arbeit erstmalig auf das nicht-triviale Spiel *Vier Gewinnt* angewendet werden sollen.

Das grundsätzliche *Ziel* dieser Arbeit besteht darin, zu untersuchen, inwieweit das Brettspiel *Vier Gewinnt* mithilfe maschinellen Lernverfahren erlernbar ist, wenn kein externer Lehrer zur Verfügung steht, das Spiel daher ausschließlich durch "*Self-Play*" (Spiele gegen sich selbst) trainiert wird.

Hierzu ist diese Arbeit in folgende Abschnitte gegliedert worden: *Kapitel 2* beschreibt die nötigen Grundlagen, die zur Entwicklung einer Trainingsumgebung nötig sind. In diesem Kapitel wird insbesondere auf die N-Tupel-Systeme und auf geeignete Lernverfahren näher eingegangen.

In *Kapitel 3* werden die zentralen Forschungsfragen, die der Arbeit zugrundeliegen, genauer formuliert.

In *Kapitel 4* soll zunächst ein kurzer Überblick über die entwickelte Software gegeben werden. Im Anschluss werden die einzelnen Entwicklungsabschnitte beschrieben und die erzielten Resultate dargestellt.

Zum Schluss wird in *Kapitel 5* eine Zusammenfassung der Ergebnisse vorgenommen und ein Ausblick auf weitere mögliche Arbeiten in diesem Gebiet gegeben.

2 Grundlagen

2.1 Vier Gewinnt

Vier Gewinnt (engl. Connect Four) ist Strategiespiel für zwei Spieler, das völlig ohne Zufallselemente auskommt. Hauptziel beider Spieler ist es, vier eigene Spielsteine in eine zusammenhängende Reihe zu bringen, wodurch das Spiel gewonnen ist.

Gespielt wird *Vier Gewinnt* auf einem Brett mit sieben Spalten und sechs Zeilen. Die Besonderheit des Spiels besteht darin, dass die Spielsteine nur von oben in eine der sieben Spalten geworfen werden können, sodass die Steine in die unterste freie Zelle der entsprechenden Spalte fallen. Der Spieler der das Spiel beginnt (*anziehender Spieler*) verwendet in der Regel gelbe und der zweite Spieler (*nachziehender Spieler*) rote Spielsteine. Die Kontrahenten werfen die Spielsteine abwechselnd in die sieben Spalten, das Aussetzen eines *Halbzuges*[1] ist nicht möglich. Ist eine Spalte vollständig belegt, kann kein weiterer Halbzug in diese Spalte vorgenommen werden. Das Spiel endet daher spätestens nach 42 Spielzügen und ist in diesem Fall als Unentschieden zu werten. Gelingt es jedoch einem der beiden Spieler vorher vier eigene Spielsteine horizontal, vertikal oder diagonal in einer Reihe zu platzieren, gewinnt der Betreffende das Spiel. Bei perfektem Spiel beider Kontrahenten gewinnt immer der anziehende Spieler (Gelb). Dazu muss dieser den ersten Stein in die mittlere Spalte werfen, in allen anderen Fällen endet das Spiel mit einem Sieg des nachziehenden Spielers oder geht unentschieden aus.

[1] Um Missverständnisse zu vermeiden, wird oft (auch in dieser Arbeit) der Begriff Halbzug anstelle von Zug verwendet, vor allem dann, wenn die Bedeutung aus dem Kontext nicht klar ersichtlich ist. Der Begriff hat sich vor allem im Schach durchgesetzt. Dort ergeben zwei hintereinander ausgeführte Aktionen (jeweils eine Aktion von Weiß und Schwarz) einen Zug. Die individuelle Aktion eines Spielers wird daher Halbzug genannt (engl. ply).

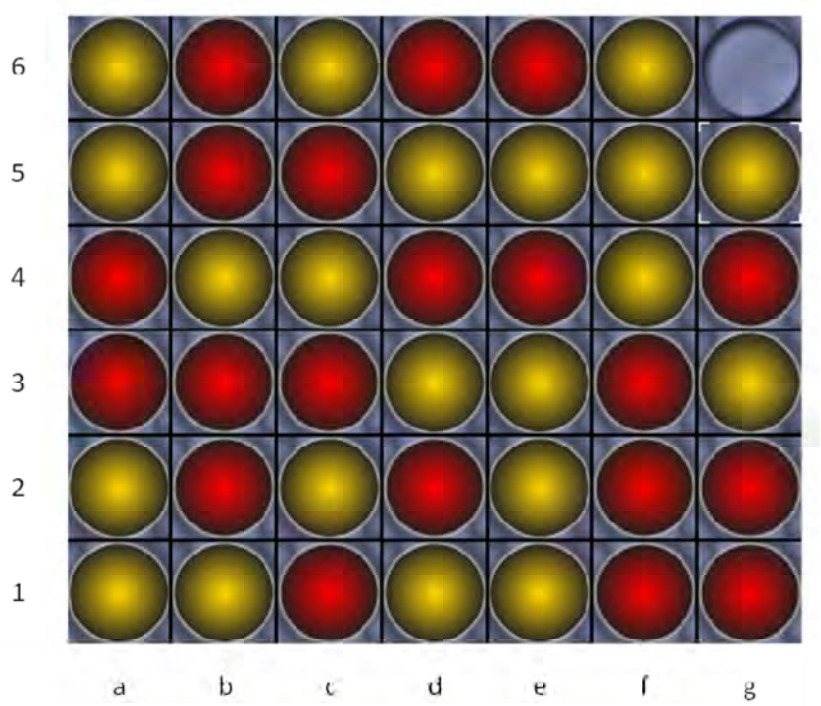


Abbildung 2.1. Typische *Vier-Gewinnt*-Stellung. Das Spiel wurde durch den anziehenden Spieler in seinem letzten Zug gewonnen (bei perfektem Spiel kann der Nachziehende die Niederlage bis zu seinem letzten Spielstein hinauszögern)

Starke Spieler versuchen möglichst früh Drohungen (engl. threats), also Viererketten mit einer freien Zelle (die unbelegte Zelle wird hierbei als *Drohung* bezeichnet), zu erstellen. Zum Spielende hin entscheiden dann vor allem Anzahl und Anordnung der Drohungen über den Spielausgang, da häufig einer der beiden Spieler unter *Zugzwang* gerät. Anfänger hingegen verlieren das Spiel oft frühzeitig, wenn sie *unmittelbare Drohungen* (engl. immediate threats) – also direkt zugängliche Drohungen – des Gegners übersehen.

Fälschlicherweise wird die Komplexität des Spiels zu oft unterschätzt. So ist auch heute noch ein sehr hoher Aufwand nötig, um *Vier Gewinnt* vollständig durchzurechnen. Lange Zeit war dies aufgrund der beschränkten Rechnerkapazitäten nicht möglich. So ist die vollständige Lösung des Spiels erst im Jahre 1988 gelungen [7] .

Eine sehr vage Schätzung für die Größe des Zustandsraums lautet $3^{42} \approx 1{,}09 \cdot 10^{20}$ (bei drei Zuständen pro Zelle), wobei dies auch gleichzeitig eine obere Schranke für den tatsächlichen Wert darstellt. *John Tromp* ist es gelungen, eine exakte Zahl an möglichen Spielzustände zu bestimmen [8] , die deutlich kleiner ist und mittlerweile auch verifiziert [28] wurde: $4531985219092 \approx 4{,}53 \cdot 10^{12}$. Dessen ungeachtet ist die Zustandsraum-Komplexität weiterhin sehr hoch, sie liegt beispielsweise noch über der Komplexität des *Mühle*-Spiels ($\approx 10^{10}$) ([9] , S. 165 f.).

2.2 Reinforcement Learning

2.2.1 Grundgedanke und Begriffe

Definition Der Begriff *Reinforcement Learning* (*RL*, deutsch: *Bestärkendes Lernen*) beschreibt eine Klasse von Lern-Problemen, bei denen ein *Agent* alleine durch *Interaktion* mit einer dynamischen *Umgebung* lernt, wie er sich in gewissen Situation zu verhalten hat, um langfristig ein bestmögliches Resultat zu erzielen; dies geschieht durch das Verteilen von *Belohnungen* und *Bestrafungen,* den sogenannten *Rewards*. Ziel des Agenten ist es daher, *die* Aktionsfolge(n) zu finden, für die die Summe aller Rewards maximiert wird. Im Gegensatz zum "*Supervised Learning*", bei dem ein externes Lehrersignal zum Einsatz kommt, um einen Agenten anzulernen, muss der *RL*-Agent sein Verhalten ausschließlich aufgrund von bereits gemachten Erfahrungen (durch die sich ergebenden Konsequenzen seiner Interaktionen mit der Umgebung) erlernen
([10] , S. 3 ff.) , ([11] , S. 237 ff.).

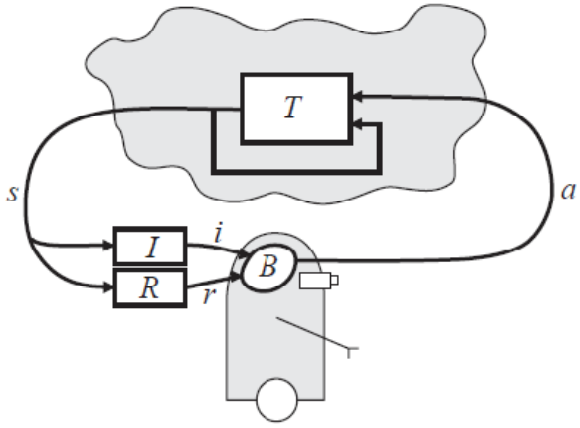


Abbildung 2.2. Modell des Reinforcement Learnings ([11] , S.238).

RL-Modell Die Grafik in Abbildung 2.2 beschreibt, wie die einzelnen Komponenten des *RL* verknüpft werden: Der Agent nimmt den *Zustand* seiner Umgebung $s \in S$ (engl.: *state*) als Eingabe i (engl.: *input*) wahr, wobei S hierbei die Menge aller möglichen Zustände beschreibt. Häufig kann der Agent nur einen Teil seiner Umgebung wahrnehmen, sodass die Eingabefunktion I nur einen Teilausschnitt des Gesamtzustandes s liefert. Für die meisten Brettspiele (Spiele mit vollständiger Information) hingegen, stellt I jedoch eine identische Abbildung dar.

Aufgrund einer Aktion $a \in \mathcal{A}(s)$, die der Agent wählt, wird wiederum der Zustand s der Umgebung in einen neuen Zustand überführt ($\mathcal{A}(s)$ beschreibt die Menge aller – durch den Agenten möglichen – Aktionen für einen Zustand). Die Umgebung ermittelt mithilfe der Funktion R einen Reward r (z.B. $r \in \{-1, 0, 1\}$) für diesen neuen Zustand,

den der Agent erhält. Der Wert des Rewards gibt hierbei an, wie vorteilhaft dieser neue Zustand für den Agenten ist. Ziel des Agenten ist es nun, anhand der erhaltenen Rewards ein Verhalten B (engl.: *behavior*) zu erlernen, mit dem der Agent seinen langfristigen Erfolg (seine zukünftigen Rewards) maximieren kann. Der Agent muss daher erlernen, jedem Zustand s eine geeignete Aktion a zuzuordnen. Eine Abbildung $\pi: S \rightarrow \mathcal{A}$ wird allgemein auch als eine mögliche Policy π des Agenten bezeichnet ([11] , S. 238 ff.).

Maximierung der Rewards Im Regelfall reicht es nicht aus, wenn ein *RL*-Agent den aktuellen Reward zu einer Aktion maximiert, häufig werden Rewards erst / auch für spätere Zustände von der Umgebung vergeben. Bei diskreten Zeitzuständen ergibt sich für eine Aktionsfolge daher eine Gesamt-Belohnung von:

$$R_t = r_{t+1} + r_{t+2} + \cdots r_T = \sum_{k=0}^{T} r_{t+k+1}\,, \tag{2.1}$$

wobei T den Zeitpunkt eines terminalen Zustandes (z.B. das Ende eines Schach-Spiels) beschreibt; Allerdings ist für gewisse Anwendungen auch $T = \infty$ unter bestimmten Umständen (unter anderem ist ein Discount-Faktor nötig, der eine unendlich große Gesamt-Belohnung vermeidet) denkbar ([10] , S. 57 ff.).

Nutzenfunktion Wie zuvor bereits erwähnt, besteht das Hauptziel eines *RL*-Agenten darin, eine Policy π zu finden / erlernen, die den Nutzen (bzw. den zu erwartenden Reward R_t) für den Agenten maximiert. Für die meisten *RL*-Verfahren ist dies gleichbedeutend mit dem Erlernen einer Nutzenfunktion (engl.: *value function*), die abschätzt, wie vorteilhaft ein Zustand für den Agenten tatsächlich ist, wenn dieser die bisherige Policy weiterverfolgt. Allgemein lässt sich der erwartete (geschätzte) Nutzen für eine bestimmte Policy π folgendermaßen beschreiben ([10] , S. 68 ff.):

$$V^{\pi}(s_t) = E_{\pi}\{R_t\} = E_{\pi}\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1}\right\} \tag{2.2}$$

Der Discount-Faktor γ (es gilt: $0 \leq \gamma \leq 1$) schwächt hierbei zukünftige Rewards etwas in ihrer Wirkung ab, wenn $\gamma < 1$ gewählt wird. Dies ist insbesondere für den Fall $T = \infty$ nötig, um eine Konvergenz der Reihe zu garantieren. Bei der Wahl von $\gamma = 0$ wird lediglich der unmittelbar nächste Reward r_{t+1} berücksichtigt. Je größer man γ wählt, desto mehr Gewicht räumt man zukünftigen Rewards ein ([10] , S. 57 ff.).

Optimale Nutzenfunktion Eine Policy π ist dann optimal, wenn keine andere Policy eine bessere Nutzenfunktion besitzt, bzw. wenn der zu erwartende Nutzen aller Zustände $s \in S$ für jede andere Policy geringer oder gleich ausfällt. Daher sind auch mehrere optimale Policies denkbar. Formal lässt sich die optimale Nutzenfunktion folgendermaßen beschreiben ([10] , S. 75 ff.):

$$V^*(s) = \max_{\pi} E_{\pi}\left\{\sum_{k=0}^{T} \gamma^k r_{t+k+1}\right\} \tag{2.3}$$

Um zumindest eine Näherung der optimalen Policy zu finden, muss beim *RL* – im Gegensatz zu vielen anderen Lernverfahren – der Agent seine Umgebung kennenlernen und aus den Interaktionen mit dieser eine Nutzenfunktion erlernen, wodurch sich wiederum die Policy des Agenten ständig ändert; Policy und Nutzenfunktion sind daher eng miteinander verbunden und bedingen sich gegenseitig.

Exploration-Exploitation-Dilemma Damit ein *RL*-Agent seine Umgebung kennenlernen kann, darf er seine Aktionen allerdings nicht vollständig aufgrund seiner aktuellen Policy treffen, da diese nicht zwangsläufig die bestmögliche existierende Lösung darstellt. Einerseits muss der *RL*-Agent bei der Wahl seiner Aktionen auf seine bisherigen Erfahrungen zurückgreifen, andererseits sollte er auch zwischenzeitlich Alternativen erforschen, da diese unter Umständen zu einer noch besseren Policy führen (*Exploration-Exploitation-Dilemma*). Es muss daher eine geeignete Kombination aus *Exploration* und *Exploitation* (Nutzung der bisherigen Erkenntnisse) gefunden werden. Wie später gezeigt wird, kann man den Agenten beispielsweise bei Brettspielen dazu zwingen, von Zeit zu Zeit zufällige Spielzüge auszuführen um alternative Teilbäume des Spiels zu explorieren ([11] , S. 243; Erläuterung des Dilemmas unter anderem am Beispiel des k-armigen Banditen).

Nutzenfunktion und Rewards Während einzelne Rewards sehr leicht zu ermitteln sind (es muss lediglich die Reward-Funktion für den Folgezustand befragt werden), ist das Abschätzen aller zukünftigen Rewards mithilfe der Nutzenfunktion deutlich schwieriger, da die Nutzenfunktion im Gegensatz zur Reward-Funktion nicht bekannt ist. Dessen ungeachtet ist die Nutzenfunktion *die* zentrale Komponente des *RL*, da die aktuelle Policy im Wesentlichen hiervon abhängt. Die Policy wiederum soll den größtmöglichen langfristigen Erfolg erzielen; mithilfe der Reward-Funktion kann lediglich der Erfolg für den unmittelbar nächsten Zustand maximiert werden, dies ist allerdings im Regelfall nicht das Ziel. Dennoch benötigt man zum Erlernen einer Nutzenfunktion die Rewards.

Wichtig ist, dass die Reward-Funktion in keinerlei Weise dazu eingesetzt wird, um eine Lehrer-Funktion im Sinne des "*Supervised Learning*" zu übernehmen, die die Aktionen des *RL*-Agenten bewertet. Stattdessen werden mithilfe der Rewards die langfristigen Ziele des Agenten abgesteckt.

Das Verteilen von Belohnungen für das Erreichen von Teilzielen (z.B. für das Schlagen einer Figur beim Schachspiel) sollte hierbei vermieden werden. Die Umgebung – die ja die Rewards vergibt – kann nicht beurteilen, ob ein Teilziel in jedem Fall auch dem Erreichen der eigentlichen, langfristigen Ziele dient (dies ist die Aufgabe der Nutzenfunktion), da sie nur die unmittelbare Attraktivität des Folgezustandes bewertet.

Vergibt man nun Belohnungen für das Erreichen von Teilzielen, so könnte der *RL*-Agent diese zwar möglicherweise erreichen, aber gleichzeitig die Gesamtziele aus den Augen verlieren ([10] , S. 56).

Verzögerte Rewards und TDL Für Brettspiele wie dem *Vier-Gewinnt-Spiel* würde dies bedeuten, dass die Umgebung erst nach Ablauf eines Spiels die Rewards vergeben kann (bzw. einen Reward von $r = 0$ für alle non-terminale Zustände). Die Schwierigkeit des *RL*-Agenten besteht allerdings dann darin, dass dieser Aktionsfolgen erlernen muss, die erst in ferner Zukunft zu einem Erfolg führen.

Einen sehr guten Lösungsansatz, um solchen *RL*-Problemen zu begegnen, liefert das sogenannte *Temporal Difference Learning* (*TD-Learning,* oft auch einfach *TDL*). Dieses Lernverfahren beruht darauf, den zu erwartenden Wert einer Nutzenfunktion für jeden diskreten Zeitschritt etwas näher in Richtung des Wertes des Folgezustandes zu verändern (daher der Begriff "*Temporale Differenz*"). Im einfachsten Fall lässt sich ein Lernschritt folgendermaßen beschreiben (für den $TD(0)$-Algorithmus; entnommen aus ([10] , S. 134):

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \ , \tag{2.4}$$

wobei γ weder dem zuvor erwähnten Discount-Faktor entspricht, mit dem zukünftige Werte der Nutzenfunktion etwas in ihrer Wirkung abgeschwächt werden. Weiterhin beschreibt α die sogenannte Lernschrittweite. Insgesamt eignet sich *TDL* sehr gut, um – trotz verzögerter Rewards – eine passende Nutzenfunktion zu lernen. In Abschnitt 2.4 wird noch einmal im Detail auf die Thematik eingegangen.

RL und Brettspiele Das Ziel eines jeden Spielagenten sollte es im Regelfall sein, während eines Spiels Züge zu machen, die ihm langfristig den größtmöglichen Vorteil (bestenfalls eine Gewinnposition) verschaffen. Aus diesem Ziel folgt unmittelbar, dass der Agent für jeden Spielzustand die Qualität der möglichen legalen Spielzüge einschätzen muss und anschließend, den für sich besten heraussucht und ausführt.

Zur Beurteilung könnten beispielsweise alle legalen Züge sukzessive ausgeführt[2] und der Nutzenfunktion für alle Folgezustände befragt werden. Im Weiteren wird hierfür der passendere Begriff *Spielfunktion* verwendet.

Die Spielfunktion $V(s_t)$, erhält als Argument einen *Spielzustand* $s_t \in S$ zu einem bestimmten Zeitpunkt t und bildet diesen auf einen reellen Wert $V(s_t) \in \mathbb{R}$ ab, wobei S die Menge aller möglichen legalen Spielzustände beschreibt. Häufig ist der Funktionswert auf ein bestimmtes Intervall beschränkt (z.B. $V(s_t) \in [-1, +1]$). Im Allgemeinen

[2] In der Literatur wird in diesem Zusammenhang häufig die aktions-basierte Nutzenfunktion $Q(\mathrm{s_t}, \mathrm{a_t})$ genannt, die den Nutzen einer bestimmten Aktion $\mathrm{a_t}$ ausgehend von Zustand $\mathrm{s_t}$ abschätzt.

sind niedrige Werte für den einen Spieler vorteilhaft, hohe dagegen für dessen Gegner [12] .

Approximierung der Nutzen- bzw. Spielfunktion Eine perfekte Spielfunktion hätte man dann, wenn jedem Spielzustand s_t der korrekte Wert zugeordnet wird. Solch eine Spielfunktion könnte man etwa mithilfe einer Tabelle realisieren, die für jeden Zustand den entsprechenden Eintrag enthält. Während ein derartiger Ansatz für Spiele mit einem kleinen Zustandsraum $|S|$ (z.B. *Tic Tac Toe*) noch denkbar ist, wäre der Speicherbedarf für komplexere Spiele wie *Vier Gewinnt* beträchtlich. Weiterhin müsste während des Trainings der Tabelleneintrag eines jeden Zustandes einzeln angelernt werden, was häufig einen enormen Trainings- bzw. Zeitaufwand bedeutet; ein weiteres Problem besteht darin, dass Spielzustände bei Verwendung einer tabellarischen Spielfunktion in keinen Zusammenhang gebracht werden können, auch wenn sie sehr ähnlich sind und denselben spieltheoretischen Wert besitzen; es ist also unmöglich, Generalisierungen durchzuführen, obwohl dies theoretisch möglich wäre.

Menschen sind in der Lage Muster zu erkennen und auf andere Situationen zu übertragen, auch wenn diese noch unbekannt sind. Auch ein guter *RL*-Agent sollte hierzu – zumindest teilweise – in der Lage sein.

Aus den oben genannten Gründen geht man daher in vielen Fällen dazu über, die Spielfunktion mithilfe einer speziellen generalisierenden Funktion zu approximieren. Häufig kommen hierzu lineare Funktionen oder künstliche neuronale Netze zum Einsatz, die mithilfe des Reinforcement Learnings trainiert werden. Findet man eine Funktion, die gute Generalisierungen vornehmen kann, ergeben sich eine Reihe von Vorteilen: So kann im Allgemeinen der Speicherbedarf drastisch reduziert werden. Durch das generalisierende Verhalten der Funktion, muss nicht mehr jeder Zustand erreicht werden, um eine Abschätzung mithilfe der Spielfunktion vornehmen zu können. Stattdessen kann anhand von Gemeinsamkeiten – zu bereits besuchten Zuständen – eine angemessene Aussage getroffen werden. Der Agent ist somit in der Lage, viele Situationen gleichzeitig zu erlernen.

Weiterhin können viele Situationen, die für starke Spieler bedeutungslos sind, wieder – zugunsten relevanter Stellungen – "verlernt" werden, wodurch die Spielfunktion nur auf eine Teilmenge aller Zustände – die für ein starkes Spiel nötig sind – spezialisiert wird.

In der Regel besteht die Herausforderung darin, eine ausgewogene Balance zwischen dem Grad der Generalisierung und der Approximationsgenauigkeit zu finden. Bei zu starker Generalisierung können keine geeigneten Stellungsbewertungen mehr vorgenommen werden, da die Spielfunktion kaum noch Unterschiede zwischen verschiedenen Zuständen ermitteln kann. Eine lineare Funktion für das *Vier-Gewinnt*-Spiel, die lediglich eine Linearkombination anhand der 42 Spielfeldzellen berechnet (Feature-Vektor mit 42 Elementen), ist daher vermutlich unbrauchbar, da jede Spielfeldzelle völlig unabhängig von den anderen Zellen bewertet wird, ohne dass größere Zusammenhänge erkannt werden können ([11] , S.259).

2.2.2 Stand der Technik zu *RL* und strategischen Brettspielen

Bereits im Jahr 1959 beschrieb Arthur Samuel [13] ein Verfahren, mit dem das Brettspiel *Dame* alleine durch *Self-Play* erlernt wurde. Das Programm erreichte zwar keine sehr hohe Spielstärke, konnte jedoch schon die Autoren des Programms schlagen. Obwohl Samuel den Begriff *TDL* noch nicht verwendete, entsprachen viele seiner vorgeschlagenen Methoden bereits den grundsätzlichen Ideen des *TDL* ([10] , S. 267).

Besondere Bekanntheit erreichte *RL* – insbesondere das *TDL* – in den späten 1980er Jahren mit dem Programm *TD-Gammon* von Gerald Tesauro [14] .

TD-Gammon konnte das Spiel Backgammon alleine durch Self-Play erlernen und war in der Lage, die stärksten Programme seiner Zeit zu schlagen und behauptete sich auch gegen einige der weltweit stärksten menschlichen Spielern.

Das Programm verwendet künstliche neuronale Netze, um eine (nicht-lineare) Spielfunktion zu erlernen. Interessanterweise benötigt das Training keine explorativen Elemente ([11] , S.271), um diese hohe Spielstärke zu erreichen. Dies liegt vor allem daran, dass durch die Zufallselemente (Würfel) im Spiel bereits genügend Abwechslung während des Trainings gegeben ist und so ausreichend viele Spielzustände besucht werden.

Es wurden in der Vergangenheit zwar bereits einige Versuche unternommen, das Spiel *Vier Gewinnt* zu erlernen, insgesamt ist die Zahl der Arbeiten hierzu jedoch recht überschaubar. So ist zum Zeitpunkt der Erstellung dieser Arbeit kein Programm bekannt, das eine (nachvollziehbar) hohe Spielstärke erreicht.

Martin Stenmark [15] vergleicht in seiner Arbeit unter anderem einen *TDL*-Agenten – basierend auf neuronalen Netzen – mit anderen Typen von Agenten. So gewinnt der *TDL*-Agent beispielsweise 75% der Spiele gegen einen Minimax-Agenten mit trivialer Evaluierung (Position der Steine dient als Bewertungsgrundlage; Spielsteine in zentralen Feldern werden am höchsten bewertet), wenn beide eine Suchtiefe von vier Halbzügen verwenden. Agenten mit deutlich höheren Suchtiefen werden in seiner Arbeit allerdings nicht als Vergleichsmöglichkeit herangezogen.

Sommerlund [16] trainierte mithilfe von *TDL* verschiedene neuronale Netze – zum Teil mit komplexeren Features – und vergleicht diese mit einer einfachen *Vier-Gewinnt*-Evaluierungsfunktion (Suchtiefe von einem Halbzug). Die Resultate fielen aus der Sicht des Autors allerdings ernüchternd aus.

Curran und O'Riordan [17] verwendeten "*Cultural Learning*" um *Vier Gewinnt* zu erlernen. Hierbei werden ganze Generationen von Agenten (basierend auf neuronalen Netzen) erzeugt, wobei die stärksten Agenten einer Generation als Lehrer der nächsten Generation dienen. Um die Spielstärke der Agenten zu bestimmen, spielten alle Agenten einer Generation gegeneinander und im Anschluss gegen einen Minimax-Spieler.

Ein Problem, das jedoch für fast alle oben genannten Arbeiten (zum Spiel *Vier Gewinnt*) gilt, liegt darin, dass eine objektive Bewertung der Spielstärke von trainierten Agenten nicht möglich ist, da in den meisten Fällen einfache Heuristiken oder Minimax-Agenten mit einer niedrigen Suchtiefe als Vergleichsmöglichkeit herangezogen werden. Letztendlich kann daher keine Aussage zu der tatsächlichen Spielstärke getroffen werden.

Auch an der *Fachhochschule Köln* sind bereits einige Arbeiten zu diesem Thema entstanden. So entwickelte Jan Schwenk [18] eine *TDL*-Umgebung zu dem Spiel *Vier Gewinnt*, jedoch konnten nur einfache Endspiele erlernt werden. Ein Grund für die vergleichsweise enttäuschenden Resultate sieht Schwenk in der fehlenden Berücksichtigung von Mustern zur Erzeugung von Eingabecodierungen für die neuronalen Netze.

Im Jahre 2008 stellte *Simon M. Lucas* [6] erstmals ein neues Verfahren zur Approximation der Spielfunktion vor, das mithilfe von zufällig ausgewählten Spielfeldausschnitten eine Vielzahl an Mustern bzw. Features erzeugt. Diese Features verwendete er um – mithilfe des *TDL* – eine lineare Funktion für das Spiel *Othello* zu erlernen. Laut eigenen Angaben waren lediglich 1250 Trainingsspiele nötig, um einen Agenten zu erzeugen, der allen bisherigen linearen und neuronalen Netzen deutlich überlegen war. Bei dem vorgestellten Verfahren handelt es sich um sogenannte N-Tupel-Systeme, die in dieser Arbeit erstmalig auf das Spiel *Vier Gewinnt* übertragen werden sollen.

2.3 N-Tupel-Systeme zur Funktionsapproximierung

Da der Zustandsraum $|S|$ für viele Brettspiele (auch für *Vier Gewinnt*) sehr groß ist, kann eine Spielfunktion, die jedem Spielzustand den exakten spieltheoretischen Wert zuordnet, nur sehr schwer gefunden werden, insbesondere dann, wenn diese Funktion noch erlernt werden muss. So wäre eine tabellarische Spielfunktion für *Vier Gewinnt* aus zwei Gründen undenkbar: Zum einen wäre sehr viel Speicher nötig und zum anderen müsste eine unvorstellbar große Zahl an Trainingsspielen durchgeführt werden, um diese Funktion vollständig zu erlernen.

Wie zuvor bereits erwähnt, geht man in solchen Fällen dazu über, die Spielfunktion zu approximieren. Durch den Informationsverlust, den man hierdurch im Regelfall in Kauf nehmen muss, können oft keine exakten Stellungsbewertungen mehr vorgenommen werden. Bei geeignetem Entwurf der Funktion lässt sich der Fehler dennoch häufig zumindest so klein halten, dass der Agent in vielen Fällen die Wahl der richtigen Aktionen erlernt.

Simon M. Lucas konnte zeigen, dass sich insbesondere N-Tupel-Systeme zur Funktionsapproximation einer linearen Funktion für das Spiel *Othello* eigneten. Auf Basis seiner Veröffentlichung [6] , soll in den folgenden Abschnitten beschrieben werden, wie eine lineare Spielfunktion für strategische Brettspiele mithilfe von N-Tupel-Systemen entwickelt werden kann.

2.3.1 Einführendes Beispiel

Obwohl die Verwendung von N-Tupel-Systemen in Zusammenhang mit lernenden Spielfunktionen erstmals von *Lucas* vorgeschlagen wurde, werden N-Tupel-Systeme an sich schon länger in anderen Bereichen erfolgreich eingesetzt.

Bereits im Jahr 1959 veröffentlichten *Bledsoe* und *Browning* eine Arbeit [19] zur Erkennung von Mustern, insbesondere zur Erkennung von alphanumerischen Zeichen mithilfe von N-Tupeln.

Ein weiteres mögliches Einsatzgebiet, nämlich die Gesichtserkennung, soll an dieser Stelle als einführendes Beispiel dienen (basierend auf einem Artikel von Simon M. *Lucas* [20]), um die grundsätzliche Funktionsweise von N-Tupel-Systemen zu verdeutlichen.

Ziel dieser beschriebenen (konventionellen) Gesichtserkennung ist es, eine Funktion zu erlernen, die einem Bild die korrekte Person zuordnet; die Funktion muss daher zunächst anhand von Beispieldaten trainiert werden. Hierzu nimmt man eine gewisse Anzahl an Trainingsbildern jeder Person (allgemein auch Klasse), aus denen das N-Tupel-System eine ganze Reihe an Merkmalen heraussucht und in sogenannten *Look-up-Tabellen* festhält. Jede Person (Klasse) bekommt hierbei einen eigenen Satz an *Look-up-Tabellen* (*LUTs*) zugeordnet, in denen alle seine individuellen Merkmale abge-

legt werden. Wichtig ist, dass während des Trainings bekannt ist, zu welcher Person ein Trainingsbild gehört, damit für jede Person die richtigen *LUTs* trainiert werden.

Im Anschluss an das Training kann eine Eingabe (Bild) dann der Person (Klasse) zugeordnet werden, für die das System die meisten Übereinstimmungen innerhalb der *LUT*-Gruppe findet.

Wie werden jedoch die einzelnen Gesichtsmerkmale ermittelt und innerhalb der *LUTs* festgehalten, sodass eine spätere Vergleichbarkeit mit bestimmten Eingaben möglich ist?

Im Wesentlichen entspricht jedes ermittelte Merkmal einem kleinen Ausschnitt des betrachteten Bildes (eine Teilmenge aller Pixel). Die sogenannten *N-Tupel* mit ihren *Abtastpunkten* legen fest, welche Pixel (Koordinaten) eines Bildes jeweils ein Merkmal bilden. Die Länge eines N-Tupels ergibt sich aus der Anzahl seiner Abtastpunkte.

In Abbildung 2.3 ist beispielhaft aufgeführt, wie sich für einige N-Tupel gewisse Gesichtsmerkmale ergeben. Insgesamt sind drei Tupel der Länge $N = 3$ dargestellt. Die Abtastpunkte selbst sind rein zufällig ausgewählt worden (im Bild durch die Linien markiert). Auf der rechten Seite der Grafik sind die Grauwerte für die betrachteten Abtastpunkte (zeilenweise für jedes N-Tupel) aufgeführt, jede Zeile entspricht daher einem spezifischen Merkmal des abgebildeten Gesichtes.

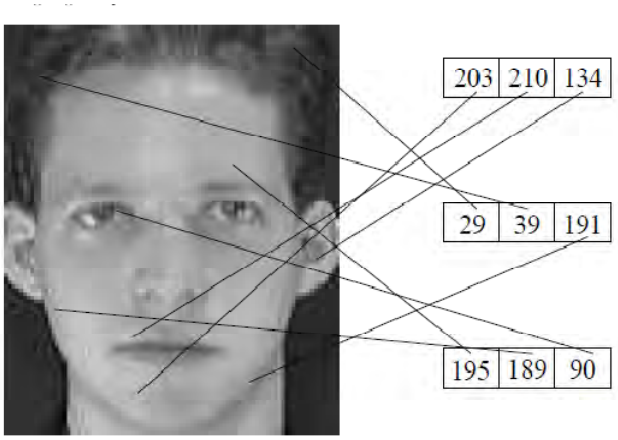


Abbildung 2.3. Gesichtserkennung mithilfe von N-Tupel-Systemen (Grafik entnommen aus: [20]). Auf der rechten Seite der Abbildung sind drei N-Tupel der Länge $N = 3$ (ein N-Tupel je Zeile) abgebildet. Die einzelnen Zahlen geben die Grauwerte der betrachteten Abtastpunkte an.

Um jedes einzelne dieser ermittelten Merkmale festhalten zu können bzw. mit bereits gespeicherten Merkmalen abgleichen zu können, sind die zuvor erwähnten Look-up-Tabellen (*LUTs*) erforderlich. Je N-Tupel erzeugt das System mehrere *LUTs* (eine pro Person); eine bestimmte *LUT* ist daher einem N-Tupel und einer Person (Klasse) zugeordnet. Hierbei ist zu beachten, dass die erzeugte Gruppe von N-Tupeln für alle Klas-

sen (Personen) identisch sein muss, damit die spätere Suche eines Bildes im trainierten System Erfolg hat.

Für jedes Merkmal lässt sich eine eindeutige Adresse berechnen, die dazu genutzt wird, ein Element innerhalb der entsprechenden *LUT* anzusprechen[3].

Häufig sind die Elemente einer *LUT* schlicht einzelne Bits, die angeben, ob ein Merkmal während des Trainings aufgetreten ist oder nicht. Alternativ ist auch die Verwendung von Zählern möglich, die die Anzahl der Zugriffe auf ein Element erfassen.

Während des Trainings wird die Gruppe der N-Tupel über jedes einzelne Trainingsbild gelegt und jeweils ein Satz von Merkmalen erzeugt. Für jedes einzelne Merkmal kann das System eine Adresse berechnen und das entsprechende Element in der zugeordneten *LUT* (die Person zu dem Bild ist während des Trainings noch bekannt) setzen bzw. inkrementieren. Daher müssen alle Elemente aller *LUTs* vor dem Training auf Null gesetzt werden.

Sucht man nach dem Training die Person zu einem Bild, muss lediglich die *LUT*-Gruppe (*LUTs* einer Person bzw. Klasse) gefunden werden, für die die meisten Merkmale übereinstimmen. Hierzu legt man wieder alle N-Tupel über das zu klassifizierende Bild und berechnet alle Adressen zu den extrahierten Merkmalen. Anschließend summiert man alle Treffer für die *LUTs* einer Klasse. Die Klasse, für die man die meisten Treffer findet, ist vermutlich die gesuchte.

Normalerweise sind deutlich mehr N-Tupel nötig als im obigen Beispiel aufgeführt, wobei deren Anzahl unter anderem auch von der Größe (Anzahl der Pixel) des Bildes abhängig ist.

Das grundsätzliche Konzept der N-Tupel-Systeme lässt sich leicht auf Brettspiele übertragen. Wie im nächsten Abschnitt beschrieben wird, lassen sich N-Tupel über ein Spielfeld (anstelle von Bildern) legen, wobei nun einzelne Spielfeldzellen – anstatt von Pixeln – als Abtastpunkte dienen. Da im Regelfall nur ein Spielfeld (eine Klasse) betrachtet wird, ordnet man jedem N-Tupel lediglich eine *LUT* zu[4]. Die Elemente der *LUTs* sind allerdings keine binären Werte mehr, stattdessen kommen Gewichte (im Fließkomma-Format, mit doppelter Genauigkeit) zum Einsatz. Auch das Training der *LUTs* nimmt deutlich an Komplexität zu, Genaueres wird in Abschnitt 2.3.2 erläutert.

Als Hauptvorteil von N-Tupel-Systemen nennt *Simon M. Lucas* unter anderem die konzeptionelle Einfachheit, die unkomplizierte Implementierung, sowie eine Leistungsfähigkeit die mit komplexeren, aber langsameren Verfahren vergleichbar ist [20] .

[3] Jede *LUT* muss daher so viele Elemente besitzen, wie Merkmale möglich sind.

[4] Wie wir allerdings später sehen werden, ist beim *Vier Gewinnt* eine Erweiterung auf zwei LUTs (eine je Spieler) nötig.

Weitere Informationen zur Mustererkennung mithilfe von N-Tupel-Systemen beschreiben *Morciniec* und *Rohwer* [21] , die in ihrer Arbeit N-Tupel-Systeme mit anderen Verfahren vergleichen. Auch in ihrer Arbeit wird an vielen Stellen auf die außerordentlichen Geschwindigkeitsvorteile, die N-Tupel-Systeme bieten, hingewiesen.

2.3.2 Anwendung von N-Tupel-Systemen auf Brettspiele

Einem durchschnittlichen menschlichen Spieler ist es nahezu unmöglich eine *Vier-Gewinnt*-Stellung als Ganzes zu untersuchen und daraus die richtigen Schlüsse zu ziehen. Vielmehr wird ein Mensch versuchen einzelne Merkmale oder Muster auf dem Spielfeld zu erkennen und diese zu einem Gesamtbild zusammenzusetzen. So sind beim Spiel *Vier Gewinnt* insbesondere die Anzahl und die Anordnung der einzelnen Drohungen für den Spielausgang entscheidend. Daraus ergeben sich verschiedene Strategien, die beide Spieler verfolgen müssen.

Mithilfe von *N-Tupel-Systemen* versucht man genau diesen obigen Sachverhalt umzusetzen. Es werden jeweils mehrere *Teilausschnitte* des Spielfeldes untersucht und diese Teilresultate zu einem *Gesamtergebnis* kombiniert. Hierzu benötigt der Entwickler keine besonderen Kenntnisse des betrachteten Spiels, er muss sich also beispielsweise keine komplexen Eingabekodierungen für neuronale Netze oder Ähnliches überlegen; dies wird bereits vom N-Tupel System übernommen, das mit wenigen Eingabegrößen auskommt und daraus einen komplexen *"Feature-Raum"* [6] erzeugt.

Ein N-Tupel-System besteht in aller Regel aus mehreren unterschiedlichen N-Tupeln, wobei einzelne N-Tupel endliche Listen der Form $\mathrm{T} = (\tau_0, \tau_1, \dots, \tau_{N-1})$ darstellen und jeweils eine Teilmenge aller *Abtastpunkte* P (engl. *sampling points*) enthalten ($\{\tau_0, \tau_1, \dots, \tau_{N-1}\} \subseteq P$). Für klassische Brettspiele entspricht daher jeder Abtastpunkt den Koordinaten $p_j = (x_j, y_j)$ einer Spielfeldzelle. Das Spiel *Tic Tac Toe* hätte $|P| = 9$ und *Vier Gewinnt* $|P| = 42$ mögliche Abtastpunkte, wobei nicht in jedem N-Tupel-System alle Spielfeldzellen zwangsläufig abgetastet werden. Die *Länge* N eines N-Tupels könnte sich für beide Spiele demnach im Bereich von $1 \leq N \leq 9$ bzw. $1 \leq N \leq 42$ bewegen[5].

Je nach Spiel ist die Belegung einer Spielfeldzelle des Spielfeldes mit mehreren unterschiedlichen Figuren möglich, insgesamt daher m Belegungen. Für jede der m möglichen Belegungen ordnet man der entsprechenden Zelle eine Zahl $s[p_j] \in \{0,1, \dots, m-1\}$ zu, wobei $s[p_j] = 0$ beispielsweise eine leere Zelle repräsentieren könnte.

[5] Wie wir später sehen werden, ist die minimale sowie maximal mögliche Länge für die meisten Brettspiele nicht empfehlenswert bzw. nicht realisierbar.

Nach welchen Kriterien die Abtastpunkte für ein N-Tupel ausgewählt werden, ist nicht vorgeschrieben. So kann man beispielsweise für jedes N-Tupel eine gewisse Zahl an zufälligen Punkten auswählen oder zusammenhängende Ketten erzeugen. Je nach Spiel können unter Umständen auch andere Arten von N-Tupeln gute Ergebnisse liefern. *Lucas* erzeugte die N-Tupel nach einem "*Random Walk*"-Verfahren [6] , bei dem – ausgehend von einem zufälligen Startpunkt – die Abtastpunkte durch zufällige Schritte in benachbarte Felder ausgewählt wurden.

Nun ist es so, dass jedes N-Tupel T eine sogenannte *Look-up-Tabelle* (*LUT*) – im einfachsten Fall ein gewöhnliches Array – zugeordnet bekommt. Anhand der aktuellen Belegung der einzelnen Abtastpunkte eines N-Tupels berechnet man einen Tabellenindex, mit dem ein einzelnes Element in der zugehörigen *LUT* angesprochen wird. Welches Element angesprochen wird, hängt von dem derzeitigen Zustand des N-Tupels und somit unmittelbar vom Zustand des Spielfeldes ab.

Die Größe einer *LUT* ergibt sich aus der Länge des N-Tupels und der Anzahl an möglichen Zuständen pro Abtastpunkt:

$$|LUT| = m^N \tag{2.5}$$

Bei Spielen mit vielen Figuren, kann die *LUT* daher schnell sehr groß werden, sodass man die Tupel-Längen auf kleinere Werte beschränken muss.

Eine *Indexierungsfunktion* für ein N-Tupel lässt sich folgendermaßen beschreiben[6]:

$$\xi(\mathrm{T}, \mathrm{s_t}) = \sum_{i=0}^{N_\mathrm{T}-1} s_t[\tau_i] \cdot m^i \tag{2.6}$$

Die obige Formel erzeugt eine natürliche Zahl (inklusive der Null) der Länge N im Stellenwertsystem der Basis m. Die Reihenfolge der Abtastpunkte innerhalb eines N-Tupels ist unerheblich, solange diese – einmal festgelegt – im Weiteren beibehalten wird. Dies ist vor allem für die Indexberechnung innerhalb der einzelnen *LUTs* wichtig [6] .

Nach diesen Vorüberlegungen ist es nun möglich, eine lineare Spielfunktion mithilfe von N-Tupel Systemen zu definieren. Betrachtet man die Elemente der Look-up-Tabellen als einfache *Gewichte*, lässt sich der Wert der Spielfunktion – abhängig vom aktuellen Spielzustand – als Summe einzelner Gewichte darstellen. Hierzu addiert man – wie in [6] vorgeschlagen – schlicht die Gewichte aller N-Tupel, die durch die Indexierungsfunktion adressiert werden. Die Spielfunktion lässt sich dann wie folgt beschreiben:

[6] In der Praxis bietet es sich an, zur Berechnung des Indexes das *Horner-Schema* heranzuziehen; dies sollte etwas Rechenzeit einsparen.

$$V(\mathrm{s_t}) = \sum_{\mathrm{i}} LUT_{(\mathrm{T_i},\mathrm{t})}[\xi(\mathrm{T}_i, \mathrm{s_t})] \tag{2.7}$$

Häufig wird auch eine weitere Funktion verwendet, die den Wertebereich der Spielfunktion auf ein gewisses Intervall beschränkt. Solch eine Funktion stellt beispielsweise der *Tangens-Hyperbolicus* dar, mit der die Spielfunktion folgende Form erhält:

$$\bar{V}(\mathrm{s_t}) = \tanh\big(V(\mathrm{s_t})\big) = \tanh\left(\sum_{\mathrm{i}} LUT_{(\mathrm{T_i},\mathrm{t})}[\xi(\mathrm{T}_i, \mathrm{s_t})]\right) \tag{2.8}$$

Ein Hauptvorteil von N-Tupel-Systemen – im Vergleich zu vielen anderen Systemen – liegt darin, dass erstere ein sehr gutes Laufzeitverhalten besitzen. Der Rechenaufwand für die Indexberechnung verhält sich logarithmisch zur *LUT*-Größe und linear zur Anzahl der N-Tupel. Weiterhin werden nur wenige Gewichte benötigt, um die Ausgabe der Spielfunktion zu bestimmen.

2.3.3 Ausnutzung von Spielfeldsymmetrien

Vielfach sind unterschiedliche Spielstellungen aufgrund von *Symmetrien* (Spiegel- und Rotationssymmetrien) identisch. In den Spielen *Tic Tac Toe* oder *Othello* können beispielsweise bis zu acht unterschiedliche Stellungen aufgrund von Spiegelungen oder Rotationen gleich sein, beim *Mühle*-Spiel sind sogar noch mehr Symmetrien möglich (Vertauschen von innerem und äußerem Ring.). Angesichts der Schwerkrafts-Charakteristik des *Vier-Gewinnt*-Spiels, führen hier lediglich Spiegelungen an der mittleren Achse zu äquivalenten Stellungen.

Eine Spielfunktion sollte im Idealfall daher für solch äquivalente Spielzustände das gleiche Resultat liefern. Dieser Sachverhalt kann für das Training der Spielfunktion und für die Stellungsbewertung ausgenutzt werden.

So erreicht man deutlich schneller erste Trainingserfolge, da viele weitere Situationen gelernt werden können, die das Training nie oder nur selten erreicht. Um die symmetrischen Stellungen für die Berechnung Spielfunktion mit einzubeziehen, ist es zweckmäßig, die Summe über die Ausgabe der Spielfunktion für alle symmetrischen Stellungen vorzunehmen [6] :

$$V_{Sym}(\mathrm{s_t}) = \sum_{x \in \mathrm{SYM}(\mathrm{s_t})} V(x) \tag{2.9}$$

wobei $\mathrm{SYM}(\mathrm{s_t})$ die Menge aller äquivalenten symmetrischen Stellungen zu einem einzelnen Spielzustand $\mathrm{s_t}$ beschreibt.

Ein etwas anderer Ansatz könnte folgendermaßen aussehen: Beim Erzeugen *eines* N-Tupels generiert man parallel dazu weitere N-Tupel, die jedoch *die* Abtastpunkte

enthalten, die aufgrund von Rotationen und Spiegelungen äquivalent sind. Dieser Gruppe von N-Tupeln wird anschließend dieselbe Look-up-Tabelle zugeordnet.

2.3.4 Konzipierung des Eingabevektors für lineare Netze

Die Funktionsweise eines N-Tupel-Systems mit Look-up-Tabellen entspricht im Wesentlichen dem eines linearen Netzes, das die Ausgabe mithilfe eines Gewichts- und Feature-Vektors berechnet. Im Programm selbst sollte jedoch auf die Verwendung von Vektoren verzichtet werden, da diese unnötig Ressourcen (Speicher sowie Rechenzeit) beanspruchen; vor allem, wenn viele und lange N-Tupel Verwendung finden. Beim Einsatz des TD(λ)-Algorithmus' (siehe Abschnitt 2.4) mit $\lambda \neq 0$ könnte der Einsatz von Vektoren jedoch nötig werden, sodass sich die Verwendung eines linearen Netzes anbietet. Sollte daher solch ein lineares Netz eingesetzt werden, verzichtet man auf die Look-up-Tabellen des N-Tupel Systems. Stattdessen legt man ein lineares Netz mit einer gewissen Zahl an Gewichten an und ermittelt für jeden Spielzustand einen geeigneten Feature-Vektor. Um den Wert der Spielfunktion zu berechnen ist es dann vollkommen ausreichend, wenn man dem linearen Netz einen Feature-Vektor übergibt.

Zunächst ist die Anzahl der Gewichte des Netzes zu ermitteln, aus der sich gleichzeitig die Länge des Feature-Vektors ergibt:

$$c = \sum_{\mathrm{i}} m^{N_i} \tag{2.10}$$

Ein Vektor, der alle c Gewichte enthält, hat die Form:

$$w = \begin{pmatrix} w_0 \\ w_1 \\ \vdots \\ w_{c-1} \end{pmatrix} \tag{2.11}$$

Um die weiter oben diskutierte Indexierungsfunktion $\xi(\mathrm{T}_\ell, \mathrm{s_t})$ weiterhin verwenden zu können, bietet es sich an, die Struktur der Look-up-Tabellen auf den Gewichts-Vektor zu übertragen. Damit die neue Indexierungsfunktion auch weiterhin eine bijektive Abbildung darstellt, also jedem N-Tupel einen eindeutigen Index zuweist, wird zu $\xi(\mathrm{T}_\ell, \mathrm{s_t})$ ein Offset addiert, um eine Verschiebung innerhalb des Vektors zu erreichen. Legt man die Gewichte der einzelnen N-Tupel $(\mathrm{T}_0, \mathrm{T}_1, \ldots)$ hintereinander im Vektor ab, ist die Berechnung des Indexes für ein Tupel T_ℓ direkt möglich:

$$\tilde{\xi}(\mathrm{T}_\ell, \mathrm{s_t}) = \xi(\mathrm{T}_\ell, \mathrm{s_t}) + \sigma(\ell) \tag{2.12}$$

wobei ℓ die Position – beginnend mit Null – des entsprechenden N-Tupels in dem Vektor $(\mathrm{T}_0, \mathrm{T}_1, \ldots)$ beschreibt.

Daraus ergibt sich ein Offset $\sigma(\ell)$ von:

$$\sigma(\ell) = \begin{cases} 0, & f\ddot{u}r\ \ell = 0 \\ \sum_{i=0}^{\ell-1} m^{N_i}, & sonst \end{cases} \tag{2.13}$$

Im Prinzip entspricht die gewählte Darstellung daher einer direkten Hintereinanderreihung aller *LUTs* des ursprünglichen N-Tupel-Systems.

Mit diesen Informationen ist es nun möglich, einen Feature-Vektor $g(\mathrm{s_t})$ des aktuellen Spielzustandes zu erzeugen, der auf den einzelnen N-Tupeln basiert. Hierzu legt man zunächst einen Nullvektor der Länge *c* an. Abhängig vom aktuellen Spielzustand werden die einzelnen Elemente des Feature-Vektors nach folgender Vorschrift aktualisiert[7]:

$$g_i(\mathrm{s_t}) \leftarrow g_i(\mathrm{s_t}) + 1 \quad \forall\, i \in \{\tilde{\xi}(\mathrm{T_0}, \mathrm{s_t}), \tilde{\xi}(\mathrm{T_1}, \mathrm{s_t}), \dots\} \tag{2.14}$$

Auch die symmetrischen Spielstellungen $\mathrm{SYM}(\mathrm{s_t})$ lassen sich ohne Weiteres mit einbeziehen (vgl. Abschnitt 2.3.3). Mithilfe dieses Feature-Vektors kann die Ausgabe des linearen Netzes anschließend sehr leicht berechnet werden:

$$V(\mathrm{s_t}) = f\big(w, g(\mathrm{s_t})\big) = w \cdot g(\mathrm{s_t}) \tag{2.15}$$

Auch hier ist die Verwendung einer Aktivierungsfunktion wie dem Tangens-Hyperbolicus denkbar. Die Spielfunktion in Gleichung (2.15) liefert die gleichen Ergebnisse, wie das zuvor beschriebene N-Tupel System mit mehreren Look-up-Tabellen, der nötige (Rechen-) Aufwand ist jedoch deutlich größer. Möglichweise kann der beschriebene Feature-Vektor auch als Eingabe anderer Netztypen dienen; in wie weit dies sinnvoll ist, muss im Einzelfall geprüft werden.

Zusätzlich zu den Gewichten *LUTs* ist die Verwendung eines Bias-Wertes – also eines unabhängigen Gewichtes – empfehlenswert.

[7] Die Schreibweise $g_i(\mathrm{s_t}) = 1$ ist unter gewissen Umständen auch denkbar. Bei Ausnutzung von Spielfeldsymmetrien, kann ein einzelner Index jedoch mehrmals auftreten, sodass $g_i(\mathrm{s_t}) = 1$ nicht in jedem Fall korrekt wäre; es sei denn, dies wird gewünscht. Es konnte bei den Untersuchungen zum *Vier-Gewinnt*-Spiel allerdings *kein Unterschied* zwischen beiden Varianten festgestellt werden.

2.4 TD-Learning in Kombination mit N-Tupel-Systemen

Im vorherigen Kapitel wurde eine mögliche Approximierung der Spielfunktion mithilfe von N-Tupel-Systemen beschrieben. Ein Verfahren, das die Spielfunktion trainiert, ist allerdings noch nicht diskutiert worden. Ein besonders geeigneter Ansatz stellt das *Temporal Difference Learning* dar, mit dessen Hilfe *Reinforcement*-Probleme gelöst werden können.

Ein Problem, dass bei praktisch allen Brettspielen auftritt, besteht darin, dass die direkte Bewertung und Einordnung einer Aktion nicht möglich ist. Erst nach einer gewissen Anzahl von Aktionen kann man eine Aussage über die Qualität der gewählten Aktionsfolge treffen; nämlich dann, wenn das Spiel beendet wird. Für den Lernprozess ist daher nicht eine einzelne Aktion entscheidend, sondern eine vollständige Sequenz derer. Eine einzelne Aktion ist erst dann als gut einzustufen, wenn diese Teil einer erfolgreichen Sequenz ist.

Probleme dieser Art können mithilfe des *Reinforcement Learnings* behandelt werden: Je nach Ausgang des Spiels erhält der Agent eine *Belohnung* oder eine *Bestrafung* (engl. *Reward*). So wird versucht, die Spielfunktion quasi rückwärts zu erlernen, indem man für jede Aktion die Bewertung des aktuellen Spielzustandes s_t etwas in die Richtung des Zielzustandes s_{t+1} anpasst. Letztendlich besteht das Ziel des Trainings also darin, mithilfe von Belohnungen und Bestrafungen eine Spielfunktion zu erlernen, um Aktionen auszuführen, die den voraussichtlichen Nutzen maximieren [12] .

2.4.1 Der TD(λ)-Algorithmus

Mithilfe des in [12] und [22] beschriebenen TD(λ)-Algorithmus' ist es möglich, einen Agenten zu trainieren, der alleine durch *Self-Play* (Trainingsspiele gegen sich selbst) eine Spielfunktion erlernt. Weiteres Expertenwissen fließt nicht in den Trainingsprozess mit ein. Hierzu werden eine ganze Reihe von Spielen simuliert, an deren Ende jeweils ein Reward vergeben wird. Der Algorithmus ist in leicht abgewandelter Form in Listing 2.1 aufgeführt.

Der Algorithmus beschreibt genau einen Spielverlauf. Je nach Spiel können einige Hundert bis zu einige Millionen Trainingsdurchgänge notwendig werden, um erste brauchbare Resultate zu erzielen.

Nach den erfolgten Initialisierungen führt der Algorithmus für beide Spieler abwechselnd Halbzüge aus. Erst wenn ein terminierender Zustand – also eine Gewinnstellung oder ein Unentschieden – erreicht wird, ist dieser Trainingsvorgang abgeschlossen.

Input: Anziehender Spieler p_0 [=+1 ("X") oder -1 ("O")], Initialposition s_0, sowie eine (teiltrainierte) Funktion $f(w;\ g(s_t))$ zur Berechnung der Spielfunktion $V(s_t)$

1: Setze $t \coloneqq 0$ und $e_0 \coloneqq \nabla_w f(w;\ g(s_0))$ ▷ s_0 gehört zu Spieler $-p_0$

2: **for** ($p \coloneqq p_0; s_t \notin S_{Final}\ ; p \leftarrow (-p),\ t \leftarrow t+1$) { ▷ Tausche Spieler in jedem Durchgang

3: $V_{old} \coloneqq f(w;\ g(s_t))$ ▷ w ändert sich durchgängig

4: Generiere Zufallszahl $q \in [0,1]$

5: **if** ($q < \varepsilon$)

6: Wähle After-State s_{t+1} zufällig aus ▷ Explorative Move

7: **else**

8: Wähle After-State s_{t+1}, der $p \cdot \begin{cases} R(s_{t+1}), & \text{für } s_{t+1} \in S_{Final} \\ f(w;\ g(s_{t+1})), & sonst \end{cases}$ maximiert ▷ Greedy-Move (Exploitation)

9: Bestimme Response $V(s_{t+1}) \coloneqq f(w;\ g(s_{t+1}))$ ▷ Lineares oder neuronales Netz

10: Bestimme Reward $r_{t+1} \coloneqq R(s_{t+1})$ ▷ Aus der Spielumgebung

11: Targetsignal $T_{t+1} \coloneqq \begin{cases} r_{t+1}, & \text{für } s_{t+1} \in S_{Final} \\ \gamma \cdot V(s_{t+1}), & sonst \end{cases}$ ▷ Trennung von Response und Reward i.d.R. nötig

12: Fehlersignal $\delta_t \coloneqq T_{t+1} - V_{old}$

13: **if** ($q \geq \varepsilon$ **or** $s_{t+1} \in S_{Final}$)

14: Lernschritt $w \leftarrow w + \alpha \delta_t e_t$ ▷ Für einen greedy bzw. terminierenden Halbzug

15: $e_{t+1} \coloneqq \gamma \lambda e_t + \nabla_w f(w;\ g(s_{t+1}))$ ▷ Berechne Eligibility-Traces

16: } ▷ Ende des Self-Plays

Listing 2.1. Inkrementeller TD(λ)-Algorithmus für Brettspiele ([12] ,[22]).

In jedem Durchgang benötigt man den Wert der Spielfunktion V_{old} für den aktuellen Spielzustand; hier wird das N-Tupel-System befragt. Wichtig ist, dass man hierfür nicht den Response des vorherigen Spielzustandes verwendet, sondern den Wert tatsächlich vollständig neu berechnet, da sich praktisch in jedem Schleifendurchlauf einzelne Gewichte des Systems ändern [12] . Dies gilt insbesondere für neuronale bzw. lineare Netze, bei denen man unter Umständen doppelte Berechnungen vermeiden möchte; diese mehrfache Berechnung kann man nicht ohne Weiteres umgehen, der erhöhte Rechenaufwand muss daher in Kauf genommen werden.

Sich ständig wiederholende Spielverläufe vermeidet man, indem jeder Halbzug mit einer gewissen Wahrscheinlichkeit ε zufällig ausgewählt wird. Solche Halbzüge bezeichnet man auch als *Explorative-Moves*. Ansonsten wählt der Algorithmus für den jeweiligen Spieler einen Halbzug, der nach Befragung der Spielfunktion am vielversprechendsten erscheint (*Exploitation*).

Bei der Berechnung des *Targetsignals* T_{t+1} ist auch die Schreibweise $T_{t+1} = r_{t+1} + \gamma \cdot V(s_{t+1})$ denkbar, da der Reward für non-terminale Zustände idealerweise gleich

Null ist und umgekehrt die Spielfunktion für terminale Spielzustände Null liefert. In der Praxis ist dies allerdings oft – unter anderem auch bei Verwendung eines N-Tupel-Systems – nicht der Fall, sodass eine getrennte Betrachtung nötig wird [12] .

Ein Lernschritt findet immer dann statt, wenn ein Greedy-Zug gespielt oder ein terminaler Zustand erreicht wurde. Der Vektor e_t beschreibt hierbei die sogenannten *Eligibility-Traces*. Wird der Parameter λ (*Trace Decay*) ungleich Null gewählt, fließen die Eligibility-Traces der vorherigen Spielzustände in den aktuellen Vektor mit ein. In dieser Arbeit soll jedoch lediglich der Fall $\lambda = 0$ betrachtet werden, sodass der Algorithmus schlichtweg den Gradienten $e_\mathrm{t} \coloneqq \nabla_\mathrm{w} f\big(w;\ g(s_t)\big)$ berechnet.

Der *Discount-Faktor* γ zinst die Werte der Spielfunktion für die Zielzustände etwas ab, da in der Regel nicht sicher ist, dass diese Zielzustände tatsächlich die bestmöglichen sind. Üblich sind laut [12] Werte im Intervall von $\gamma \in [0{,}9\ ;\ 1]$.

Mithilfe der *Lernschrittweite* α vermeidet man zu starke Änderungen der Gewichte in die eine oder andere Richtung. Typischerweise wird die Lernschrittweite im Laufe des Trainings immer weiter verkleinert, um Anfangs eine hohe Konvergenzgeschwindigkeit zu erhalten und im weiteren Verlauf stark oszillierendes Verhalten zu vermeiden. In der Praxis hat sich eine exponentielle Abnahme der Schrittweite bewährt. So sollte im Idealfall das Fehlersignal im Laufe des Trainings minimiert werden.

Letztendlich handelt es sich hierbei um ein klassisches Gradientenabstiegsverfahren mit variabler Lernschrittweite, vorausgesetzt es wird $\lambda = 0$ gewählt.

2.4.2 Optimierung der Lernschritte

Wie weiter oben bereits erwähnt wurde, sollte man im Programm auf die Verwendung von Vektoren verzichten, sofern dies nicht unumgänglich ist. Vor allem der Feature-Vektor des N-Tupel-Systems kann unter Umständen sehr groß werden, sodass in der Praxis deutliche Laufzeitnachteile zu erwarten sind, sollte man keine geeignete Alternative finden. In Abschnitt 2.3.2 ist bereits ein vergleichsweise schnelles Verfahren zur Berechnung der Spielfunktion vorgestellt worden, für die einzelnen Lernschritte im TD(λ)-Algorithmus ist eine effizientere Herangehensweise noch nicht bekannt. Für den Optimierungsansatz entscheidend sind die Eligibility-Traces, die in Listing 2.1 folgendermaßen beschrieben werden:

$$e_{\mathrm{t}+1} \coloneqq \gamma\lambda e_\mathrm{t} + \nabla_\mathrm{w} f\big(w;\ g(s_{t+1})\big) \tag{2.16}$$

Sollte es möglich sein, die Vektorschreibweise der Eligibility-Traces in eine etwas andere Darstellung zu überführen, ist auch eine effizientere Behandlung in der Implementierung denkbar.

Für $\lambda \neq 0$ ist eine deutliche Vereinfachung allerdings nicht ohne Weiteres vorstellbar. Da dies in der Praxis oft ohnehin nicht vorteilhaft ist, wird im Weiteren ausschließlich der Fall $\lambda = 0$ betrachtet, sodass sich obige Gleichung etwas vereinfachen lässt:

$$e_{\mathrm{t}} \coloneqq \nabla_{\mathrm{w}} f\big(w;\ g(s_t)\big) \tag{2.17}$$

Bei der – mithilfe des N-Tupel Systems realisierten – Spielfunktion handelt es sich um eine Funktion der Form[8]:

$$\bar{V}(\mathrm{s_t}) = f\big(w;\ g(\mathrm{s_t})\big) = \tanh\big(w \cdot g(\mathrm{s_t})\big) \tag{2.18}$$

bzw.:

$$f\big(w;\ g(\mathrm{s_t})\big) = \tanh\left(\sum_k w_k \cdot g_k(\mathrm{s_t})\right) \tag{2.19}$$

Der Gradient lässt sich für diese Funktion vergleichsweise leicht ermitteln:

$$\begin{aligned} \nabla_{\mathrm{w}} f\big(w;\ g(s_t)\big) &= \left[1 - \tanh^2\left(\sum_k w_k \cdot g_k(\mathrm{s_t})\right)\right] \cdot g(\mathrm{s_t}) \\ &= \left[1 - f\big(w;\ g(\mathrm{s_t})\big)^2\right] \cdot g(\mathrm{s_t}) \end{aligned} \tag{2.20}$$

Bei Verwendung der Aktivierungsfunktion (Tangens-Hyperbolicus) liefert der Gradient wieder den Feature-Vektor $g(\mathrm{s_t})$, der mit einem Vorfaktor multipliziert wird (2.20). Dieser Vorfaktor wird zu Eins, sollte man auf die Aktivierungsfunktion verzichten.

Nachdem der Gradient bekannt ist, lässt sich ein Lernschritt in leicht abgewandelter Form notieren:

$$w \leftarrow w + \alpha\delta_t \left[1 - f\big(w;\ g(\mathrm{s_t})\big)^2\right] \cdot g(\mathrm{s_t}) \tag{2.21}$$

Nun kann man die Tatsache ausnutzen, dass oftmals große Teile des Feature-Vektors lediglich Null-Elemente enthalten. Ein einzelnes Gewicht ändert sich während des Lernschrittes nur dann, wenn der entsprechende Eintrag im Feature-Vektor ungleich Null ist. Beim Einsatz eines N-Tupel-Systems betrifft dies ausnahmslos Einträge, die von der Indexierungsfunktion $\tilde{\xi}(\mathrm{T}_\ell, \mathrm{s_t}) = \xi(\mathrm{T}_\ell, \mathrm{s_t}) + \sigma(\ell)$ adressiert wurden. Es ist also vollkommen ausreichend, wenn der Wert

$$\Delta w = \alpha\delta_t \left[1 - f\big(w;\ g(\mathrm{s_t})\big)^2\right] \tag{2.22}$$

zu den – durch die Indexierungsfunktion adressierten – Gewichte addiert wird:

[8] Die Ermittlung der Vektoren w und $g(\mathrm{s_t})$ und deren Besonderheiten sind bereits in Abschnitt 2.6 diskutiert worden.

$$w_i \leftarrow w_i + \Delta w \cdot g_i(\mathrm{s_t}) \qquad \forall\, g_i(\mathrm{s_t}) \neq 0 \tag{2.23}$$

bzw.:[9]

$$w_i \;\leftarrow\; w_i + \Delta w \qquad \forall\, i \in \left\{\tilde{\xi}(\mathrm{T}_0, \mathrm{s_t}), \tilde{\xi}(\mathrm{T}_1, \mathrm{s_t}), \ldots\right\} \tag{2.24}$$

Sollte man auf ein lineares Netz mit einem individuellen Gewichts-Vektor verzichten und stattdessen die Verwendung mehrerer Look-up-Tabellen bevorzugen, wird der Gewichts-Vektor schlicht in die entsprechenden *LUTs* zerlegt. Formal ändert sich hierdurch jedoch nichts. Da keine Offsetberechnungen mehr nötig sind, kommt wieder die Indexierungsfunktion $\xi(\mathrm{T}_\ell, \mathrm{s_t})$ zum Einsatz. Auch die Berechnung des Skalarproduktes $w \cdot g(\mathrm{s_t})$ zur Bestimmung der Spielfunktion kann vermieden werden, sodass man kürzere Laufzeiten erreicht, indem man die Spielfunktion nach Formel (2.7) bzw. (2.9) berechnet.

Der Übergang zu einem N-Tupel-System mit getrennten Look-up-Tabellen – so wie es auch *Lucas* in [6] einsetzt – kann folgendermaßen beschrieben werden:

$$LUT_{\mathrm{T}}[\xi(\mathrm{T}, \mathrm{s_t})] \leftarrow LUT_{\mathrm{T}}[\xi(\mathrm{T}, \mathrm{s_t})] + \Delta L \tag{2.25}$$

Analog zu Gleichung (2.22) gilt:

$$\Delta L = \alpha \delta_t [1 - \bar{V}(\mathrm{s_t})^2] \tag{2.26}$$

Auch an dieser Stelle verzichtet man auf den Faktor $[1 - \bar{V}(\mathrm{s_t})^2]$, sollte die Spielfunktion $V(\mathrm{s_t})$ (keine Aktivierungsfunktion) zum Einsatz kommen.

Es ist völlig ausreichend, ΔL einmalig zu berechnen. Diesen Wert addiert man anschließend auf alle betroffenen Gewichte.

Letztendlich kann man beim Einsatz von Look-up-Tabellen vollständig auf die zeit- und speicherintensive Berechnung von Vektoren (Feature- bzw. "Lernschritts"-Vektoren) während des Trainings verzichten, da nur wenige Komponenten der jeweiligen Vektoren für weitere Betrachtungen relevant wären. Dies ist insbesondere dann von Vorteil, wenn sehr viele Trainingsspiele nötig sind.

[9] Vgl. Gleichung (2.14 Analog hierzu). Bei der Ausnutzung von Spielfeldsymmetrien erweitert sich die Liste der N-Tupel um *die* der spiegel- bzw. rotationssymmetrischen Stellungen.

3 Konzeption der Lern-Experimente

Bevor mit den eigentlichen Untersuchungen begonnen wird, soll an dieser Stelle kurz beschrieben werden, wie auf Basis der in Kapitel 2 beschriebenen Grundlagen die weiteren Arbeitsschritte festgelegt und geeignete Forschungsfragen formuliert wurden.

3.1 Festlegung der Arbeitsschritte

Zunächst ist die Implementierung eines allgemeingültigen N-Tupel-Systems mit den dazugehörigen Schnittstellen nötig, um eine Spielfunktion approximieren zu können. Das System kann weitestgehend unabhängig von den weiteren Programmkomponenten erstellt werden. Auch erste Tests sind bereits in begrenztem Umfang möglich.

Als Ausgangspunkt für die weiteren Arbeiten dient ein bereits existierendes Software-Framework ([12] ,[22]) für *RL* und *Tic Tac Toe*.

Beim Spiel *Tic Tac Toe* handelt es sich um ein Strategiespiel, dass dem Spiel *Vier Gewinnt* sehr ähnlich ist. Im Gegensatz zu *Vier Gewinnt* müssen die beiden Spieler jedoch nur versuchen, drei eigene Steine in einer Reihe zu platzieren, um das Spiel zu gewinnen, wobei die Regel der Schwerkraft bei *Tic Tac Toe* keine Anwendung findet. Der Hauptunterschied zu *Vier Gewinnt* liegt jedoch in der deutlich reduzierten Komplexität (das Spielfeld besteht lediglich aus einer 3x3 Matrix).

Zur Einarbeitung in die Thematik erscheint dieses Spiel daher gut geeignet, auch unter dem Gesichtspunkt, dass zentrale Programmteile bereits zur Verfügung stehen. Da das Framework bereits das Erlernen linearer Spielfunktionen erlaubt[10], ist lediglich eine Erweiterung um das N-Tupel-System nötig, um erste Untersuchungen anstellen zu können.

Bevor nun dazu übergegangen werden kann, Untersuchungen am Spiel *Vier Gewinnt* durchzuführen, sind Vergleichsmöglichkeiten zur Einschätzung der Spielstärke nötig. Ein Agent – basierend auf einem Baumsuchverfahren – eignet sich hierfür besonders gut, da – zumindest theoretisch – perfektes Spiel möglich ist und die Spielstärke anhand der Suchtiefe eingestellt werden kann. Agenten mit Baumsuchverfahren haben allerdings den entscheidenden Nachteil, dass schnell eine sehr große Zahl an

[10] Im Prinzip ist es daher ausreichend, eine Funktionalität zur Verfügung zu stellen, die den Feature-Vektor $g(s_t)$ nach der Vorschrift (Formel) (2.14) erzeugt, da alles Weitere (Lernschritte, Berechnung der Ausgabe) vom vorhanden linearen Netz übernommen werden kann. Wie zuvor erwähnt, sollte aus Effizienzgründen allerdings später dazu übergegangen werden, Look-up-Tabellen zu verwenden. Dennoch ist erstere Variante als Vergleichsmöglichkeit gut geeignet.

Spielknoten durchsucht werden müssen. Daher ist insbesondere auf die Effizienz der Algorithmen großer Wert zu legen.

Auch auf die Wiederverwendbarkeit grundlegender Datenstrukturen (Spielfeldrepräsentation) und Methoden (Suche nach Viererketten) sollte zu diesem Zeitpunkt bereits geachtet werden.

Schließlich kann ein *TDL*-Agent für das Spiel *Vier Gewinnt* realisiert und im Anschluss mit den Experimenten begonnen werden. Das Erlernen des kompletten Spiels gleich zu Beginn sollte jedoch vermieden werden, da dies unter anderem die Fehlersuche erschwert. Alternativ könnten beispielsweise zunächst kleinere Spielfelder verwendet werden, um die Komplexität zu reduzieren; dies würde jedoch einen gewissen Mehraufwand bedeuten. Sinnvoller erscheint daher das Erlernen von Endspielsituationen bzw. Eröffnungsphasen. Dadurch wäre es unter anderem auch möglich, die Komplexität schrittweise zu erhöhen. Erst wenn dies zufriedenstellende Resultate liefert, sollte das vollständige Spiel betrachtet werden.

3.2 Zentrale Forschungsfragen

Wie zuvor bereits erwähnt, konnte *Simon M. Lucas* mithilfe eines N-Tupel-Systems in Verbindung mit *TDL* sehr gute Resultate beim Spiel *Othello* erzielen [6] , wobei lediglich 30 N-Tupel mit einer Länge zwischen zwei und sechs verwendet wurden.

Alle beschriebenen Ergebnisse wurden ohne die Zuhilfenahme eines externen Lehrers (reines "*Self-Play*"-Training) oder sonstigem spieltheoretischem Wissen erreicht; lediglich Spielfeldsymmetrien wurden zur Beschleunigung des Trainings ausgenutzt.

Da die Komplexität von *Vier Gewinnt* [8] deutlich unter der von *Othello* liegt ([9] , S. 167), sind hier möglicherweise ähnlich gute Ergebnisse zu erzielen. Hierzu lassen sich eine Reihe von (teilweise offen gehaltenen) Forschungsfragen formulieren, die später beantwortet werden sollen[11]:

1. Lässt sich *RL* mit N-Tupel-Systemen generell auf das Spiel *Vier Gewinnt* übertragen bzw. sind hier ähnlich gute Resultate zu beobachten? Welche Grundvoraussetzungen sind hierfür nötig?
2. Kann ein starker Spieler alleine durch "*Self-Play*" erzeugt werden?
3. Lucas benötigte nur 1250 Trainingsspiele, um eine hohe Spielstärke seines *Othello*-Agenten zu erreichen [6] . Ist *Vier Gewinnt* auch anhand weniger Spiele erlernbar?
4. Welche Konfigurationen sind für ein erfolgreiches Training nötig (sofern dies möglich ist)? Konkret wären insbesondere folgende Fragen von Interesse:
    a. Welche Anzahl und Länge der N-Tupel ist für *Vier Gewinnt* nötig?
    b. Müssen N-Tupel von Hand vorgegeben werden oder ist eine zufällige Generierung möglich? Welche Generierungsvorschriften eignen sich besonders?
    c. Wie müssen die weiteren Parameter (z.B. Lernschrittweite oder Explorationsrate) gewählt werden und sind noch weitere wichtige Punkte zu beachten?
5. Inwieweit lassen sich bestimmte Trainingsresultate unter gewissen Rahmenbedingungen reproduzieren?

Lassen sich obige Fragen nicht bzw. nicht positiv beantworten (aber auch unabhängig hiervon), sind einige alternative Fragestellungen denkbar:

1. Können zumindest die Ursachen für unbefriedigende Trainingsresultate ausfindig gemacht und beschrieben werden?
2. Gelingt es zumindest, kleinere Ausschnitte des Spiels (*Vier Gewinnt*) zu erlernen (z.B. Eröffnungs- oder Endspielsituationen) ?
3. Lassen sich Trainingsresultate durch Ausnutzung gewisser Spielcharakteristika oder durch andere Ansätze, wie z.B. dem "*Überwachten Lernen*" verbessern?

[11] Begriffe wie "gut", "stark" oder "wenig" werden später genauer spezifiziert.

4 Reinforcement-Learning-Experimente mit N-Tupel-Systemen

In diesem Kapitel soll zunächst ein kurzer Überblick über die entwickelte Software gegeben werden. Im Anschluss werden Schritt für Schritt die einzelnen Entwicklungsabschnitte erläutert und die erzielten Resultate diskutiert. So wird beispielsweise zu Beginn das triviale Spiel *Tic Tac Toe* untersucht, das dem Vier-Gewinnt-Spiel relativ ähnlich ist. Anschließend wird dazu übergegangen, die Eröffnungsphasen von *Vier Gewinnt* und letztendlich das vollständige Spiel auf ihre Erlernbarkeit zu überprüfen. Die zentralen Ergebnisse sind bereits in einem Artikel [23] in etwas kompakterer Form dargestellt.

4.1 Trainings- und Testumgebung

4.1.1 Grundlegende Funktionalitäten der Software

Die Trainings- und Testumgebung wurde vollständig mit der Programmiersprache Java innerhalb der Entwicklungsumgebung Eclipse entwickelt. Ausgangspunkt war eine Java-Software für *RL* und *Tic Tac Toe* ([12] , [22]), die im Rahmen dieser Arbeit um sämtliche Aspekte zu N-Tupel-Systemen und zu *Vier Gewinnt* erweitert wurde. Dem Anwender wird eine GUI zur Verfügung gestellt, die eine einfache Bedienung ermöglicht und wichtige Informationen und Vorgänge visualisiert. Einige grundlegende Funktionen sollen im Folgenden dargestellt werden:

- Spiele können im Schritt-für-Schritt Modus mit zwei beliebigen (trainierten) Agenten durchgeführt werden. Falls gewünscht, kann eine Startposition vorgegeben werden.
- Hierbei stehen dem Anwender perfekt spielende Minimax-Agenten, vollständig zufallsbasierte Agenten und schließlich *TDL*-Agenten mit N-Tupel-Systemen zur Verfügung. Auch die Wahl von menschlichen Spielern ist möglich.
- Während der Spiele bekommt man die Stellungsbewertungen der jeweiligen Agenten angezeigt und kann so deren Entscheidungen nachvollziehen.
- Die Spielstärke von Agenten kann mithilfe einer Evaluierungsfunktion eingeschätzt werden (vgl. Abschnitt 4.1.3 und 4.1.4).

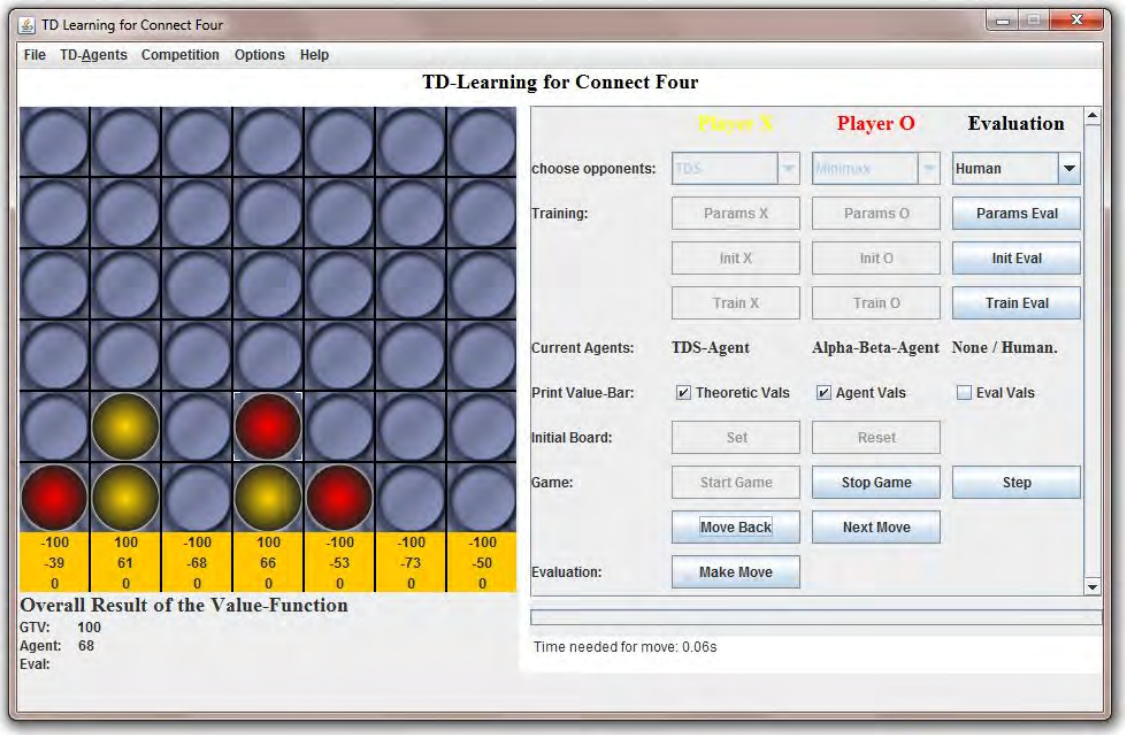


Abbildung 4.1. Hauptfenster der Trainingsumgebung. Dargestellt ist in diesem Fall ein Spiel zwischen einem *TDL*-Agenten als anziehenden Spieler und einem Minimax-Agenten. In der Leiste unter dem Spielfeld sind die spieltheoretischen Werte sowie die Bewertungen des *TDL*-Agenten (da dieser am Zug ist) für die einzelnen Spalten ablesbar.

Da das Training und die Untersuchung von *TDL*-Agenten und den zugehörigen N-Tupel-Systemen im Vordergrund dieser Arbeit steht, stellt das Programm eine ganze Reihe weiterer Möglichkeiten speziell für diese Zwecke zur Verfügung:

- Laden und Speichern von Agenten inklusive aller dazugehörigen *LUTs* und weiterer Datenstrukturen.
- Visualisierung aller N-Tupel eines Systems
- Vorgabe der N-Tupel durch den Anwender
- Umfassende Untersuchung der *LUTs* in separatem Fenster: Hier kann beispielsweise die Ausgabe einer oder mehrerer *LUTs* für individuelle Belegungen der N-Tupel oder für vollständige Spielstellungen betrachtet werden. So ist es für den Anwender möglich, nachzuvollziehen wie das System die Adressierung innerhalb der *LUTs* vornimmt und Teilergebnisse zu einem Gesamtresultat kombiniert.
- Training von *TDL*-Agenten mit einer konsolenbasierten Version des Programms – insbesondere für UNIX-Systeme. Die Ausgabe aller trainingsrelevanten Daten erfolgt im CSV-Format (*Comma-Separated Values)*.

Für weitere Funktionen und Details sei auf die englischsprachige Hilfedatei im Anhang verwiesen; diese kann auch direkt von der GUI aus aufgerufen werden.

4.1.2 Weitere Details zur Software und Trainingskonfiguration

Bevor ein Trainingsvorgang für einen *TDL*-Agenten gestartet werden kann, müssen eine ganze Reihe von Parametereinstellungen vorgenommen werden. Bestimmte Parameterkonstellationen werden in den nachfolgenden Abschnitten aufgegriffen, an dieser Stelle sollen jedoch noch verschiedene Konfigurationen der Explorationsrate sowie die Verwendung von Hashtabellen diskutiert werden.

Für die Explorationsrate ε lassen sich ein Initial- sowie ein Finalwert festlegen. Der Anwender kann ferner zwischen vier Funktionen auswählen, die während des Trainings zu unterschiedlichen Kurven führen.

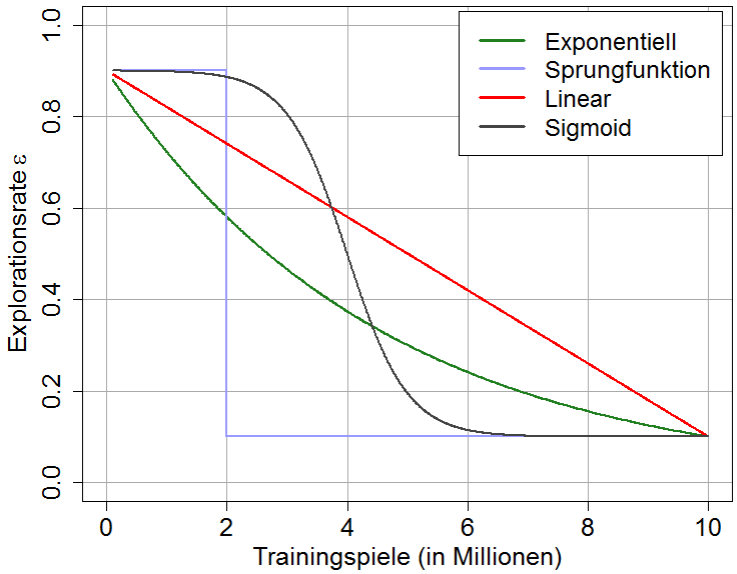


Abbildung 4.2. Dargestellt sind die vier möglichen Anpassungsfunktionen der Explorationsrate ε. Beim Einsatz einer Sigmoid- oder einer Sprungfunktion lassen sich der Wendepunkt bzw. die Sprungstelle durch den Anwender vorgeben. In diesem Fall fällt ε in allen vier Fällen innerhalb von 10 Mio. Spielen von 0,9 auf 0,1 ab.

Bei Verwendung einer *Sigmoidfunktion* (graue Kurve[12]) ergibt sich im Regelfall ein umgekehrt S-Förmiger Verlauf. Die Funktion wird beschrieben durch:

$$\varepsilon(n) = (\epsilon_N - \epsilon_0) \cdot S\left(\frac{n - n_{wp}}{N} m\right) + \epsilon_N \tag{4.1}$$

Hierbei entspricht n der aktuellen Zahl an Trainingsspielen, ϵ_0 und ϵ_N dem Initial- bzw. Finalwert und N der Gesamtzahl an Trainingsspielen. Die Funktion lässt mithilfe des Parameters n_{wp} – der den Wendepunkt der Funktion darstellt – entlang der Abszisse ver-

[12] Die Angabe der Farbe bezieht sich für die nachfolgenden Angaben jeweils auf Abbildung 4.2.

schieben. Mithilfe der Variable m kann man die Steigung verändern. Zurzeit ist diese im Programm fest verdrahtet und nimmt den Wert $m = 10$ an. Weiterhin gilt[13]:

$$S(x) = \frac{1}{e^{2x} + 1} = \frac{1 - \tanh(x)}{2} \tag{4.2}$$

Die *Sprungfunktion* (blaue Kurve) benötigt lediglich einen zusätzlichen Parameter, der die Sprungstelle vom Initial- auf den Finalwert beschreibt, die *lineare* (rote Kurve) bzw. *exponentielle*[14] (grüne Kurve) Anpassungsfunktionen benötigen diesen nicht. Für den exponentiellen Verlauf gilt:

$$\varepsilon(n) = \epsilon_0 \cdot a^n \tag{4.3}$$

bzw.

$$\varepsilon(n) = \varepsilon(n-1) \cdot a \text{ mit } \varepsilon(0) = \epsilon_0 \tag{4.4}$$

mit

$$a = \sqrt[N]{\frac{\epsilon_N}{\epsilon_0}} \tag{4.5}$$

Generell lässt sich sagen, dass niedrige Werte für die Explorationsrate das Training verlangsamen, da häufiger auf das N-Tupel-System zugegriffen wird. Vor Trainingsbeginn kann der Discount-Faktor γ auf einen konstanten Wert eingestellt werden, der Wert des Parameters λ beträgt Null und ist nicht änderbar (siehe Abschnitt 2.4.2).

Da in den nächsten Abschnitten im Regelfall Trainingskonfigurationen mit bis zu 10 Mio. Trainingsspielen beschrieben werden, ist besonderer Wert auf die Laufzeit der trainingsrelevanten Methoden zu legen.

Daher werden auch sogenannte Bitboards als grundlegende Datenstruktur zur Spielfeldrepräsentation des TD-Agenten eingesetzt[15]. Dies beschleunigt die Trainingsvorgänge deutlich, da viele grundlegende Operationen – beispielsweise die Erkennung von Terminalzuständen – mit wenigen CPU-Befehlen auskommen. Details hierzu können in der Projektarbeit ([24] , S. 27 ff.) nachgelesen oder dem Quelltext (*c4.ConnectFour*) entnommen werden.

[13] Im Programm kommt ausschließlich die zweite Variante der Formel (4.1) zum Einsatz, die den Tangens-Hyperbolicus beinhaltet. Anstelle der Standardfunktion *Math.tanh()* wird eine eigene Funktion genutzt, die auf einem Tabelleninterpolationsverfahren basiert und ca. um das Dreifache schneller ist (mit vernachlässigbarem Genauigkeitsverlust).

[14] Auch die Lernschrittweite folgt diesem exponentiellem Verlauf. Im Gegensatz zur Explorationsrate lässt sich hierfür zurzeit auch keine andere Anpassungsfunktion wählen.

[15] Dies gilt allerdings nur für das Vier Gewinnt Spiel. *Tic Tac Toe* hat eine deutlich geringere Komplexität, sodass Bitboards nicht vonnöten sind. Nichtsdestotrotz enthält die Klasse *diverses.CountPositions* eine rudimentäre Umsetzung des Bitboard-Prinzips, dass sich leicht auf den entsprechenden TD-Agenten übertragen ließe.

Wie bereits erwähnt wurde, erzeugt das Programm für jedes N-Tupel unabhängige Look-up-Tabellen, die mit weiteren Informationen in eigenständigen Objekten verwaltet werden. Dieser Ansatz entspricht zwar nicht ganz dem eines klassischen linearen Netzes, das alle Gewichte in einem einzelnen Vektor führt, dennoch sollten beide Systeme nach außen hin das gleiche Verhalten aufweisen. Weiterhin wird in diesem Programm auf die Erzeugung ganzer Vektoren verzichtet, da im Regelfall nur wenige Komponenten innerhalb dieser von Bedeutung sind. Ein vom N-Tupel-System generierter Feature-Vektor würde hauptsächlich Null-Elemente enthalten, die für weitere Betrachtungen überflüssig wären. Man spart daher Rechenzeit und Speicher ein, wenn im weiteren Verlauf lediglich die wenigen relevanten Komponenten des Vektors herangezogen werden (vgl. Abschnitt 2.3.4 und 2.4.2).

Um die Komplexität des *Vier Gewinnt* Spiels zu reduzieren, besteht außerdem die Möglichkeit, die einzelnen Trainingsspiele bei einer gewissen Anzahl von Spielsteinen auf dem Brett zu unterbrechen. Der Reward – daher der spieltheoretische Wert - muss anschließend von der Alpha-Beta Suche ermittelt werden. So kann man dann beispielsweise einen *TDL*-Agenten trainieren, der die Eröffnungsphase bis zu zwölf Spielsteinen beherrscht. Vor allem lassen sich hierdurch auch die implementierten Algorithmen auf ihre Richtigkeit überprüfen.

Weiterhin stellte sich nach einigen Untersuchungen mit einem *Profiler* heraus, dass insbesondere die Implementierung der *Indexierungsfunktion* $\xi(\mathrm{T}, \mathrm{s_t})$ für Formel (2.6) besonders rechenzeitintensiv ist; vor allem deshalb, weil beim Zugriff auf die Spielfunktion im Regelfall mehrere Dutzend Indexes berechnet werden müssen. Aus diesem Grund wurde sich dazu entschieden, eine optional wählbare Hashtabelle zu konzipieren, die für einzelne Stellungen den entsprechenden Satz an Indexes aller N-Tupel aufnimmt. Insbesondere für Systeme mit vielen N-Tupeln ist diese Maßnahme hilfreich.

Mithilfe des in [25] beschriebenen *Zobrist-Hashings* lassen sich solche Eintragungen in die Hash-Tabelle vergleichsweise schnell vornehmen. Experimentell hat sich bewährt, existierende Einträge im Falle einer Kollision generell zu überschreiben.

Die Größe der Hashtabelle ist auf maximal $2^{16} = 65536$ Einträge beschränkt, im Normalfall kann die Größe auch auf die Hälfte bzw. ein Viertel des Wertes reduziert werden.

In Verbindung mit einigen kleineren Optimierungen reduziert sich die durchschnittliche Laufzeit des Trainings für die Standardkonfiguration (vgl. Abschnitt 4.1.5) von ursprünglich 9h 30min auf weniger als die Hälfte (4h 15min)[16]. Jedes Trainingsspiel benötigt daher im Schnitt eine Zeit 1,5ms.

[16] Die Größe der Hashtabelle war auf $2^{15} = 32768$Einträge begrenzt. Die Durchschnittszeit wurde jeweils anhand von zehn Trainingsdurchgängen bestimmt.

4.1.3 Minimax-Agent für Vergleichszwecke

Im nachfolgenden Abschnitt soll erläutert werden, wie mithilfe eines Agenten (im Weiteren Minimax-Agent) – basierend auf einer klassischen Baumsuche – eine möglichst objektive Einschätzung der Spielstärke anderer Agenten erfolgen kann.

Die Entwicklung eines solchen Agenten ist im Regelfall nicht trivial, sodass oft ein sehr hoher Entwicklungsaufwand nötig wird.

Für die Untersuchungen in dieser Arbeit wurde ein Minimax-Agent entwickelt, der das Spiel *Vier Gewinnt* perfekt beherrscht (daher exakte Stellungsbewertungen vornehmen kann). Einige zentrale Komponenten sollen an dieser Stelle kurz beschrieben werden. Im Projektbericht ([24] , S. 24 ff.) sind die eingesetzten Verfahren und Techniken detailliert ausgeführt.

Ziel der Minimax-Suche ist es, den eigenen Gewinn zu maximieren, bei gleichzeitiger Minimierung des Gewinns des Gegners. Hierzu wird ein *Suchbaum* aufgebaut und alle zukünftigen Spielstellungen bis zu einer gewissen Tiefe analysiert. Bei maximaler Suchtiefe (Alle Blattknoten entsprechen terminalen Spielstellungen) ist eine exakte Stellungsbewertung und somit perfektes Spiel möglich. Bricht die Suche vorzeitig ab (Erreichen des *Suchhorizontes*), sind sogenannte *Evaluierungsfunktionen* nötig, die den spieltheoretischen Wert der Blattknoten einzuschätzen versuchen.

Da der Suchaufwand exponentiell mit der Suchtiefe steigt, können selbst viele Vier-Gewinnt-Stellungen nicht ohne weiteres exakt untersucht werden.

Eine sehr wichtige Technik, um den Suchraum zu verkleinern, stellt die *Alpha-Beta-Suche* dar, mit der häufig große Teile des Spielbaums abgeschnitten werden können.

Der Minimax-Algorithmus untersucht alle Knoten des Spielbaums, unabhängig davon, ob dies im Einzelfall tatsächlich noch nötig ist. Dabei könnte in vielen Fällen auf die Untersuchung ganzer Teilbäume verzichtet werden, da den Opponenten unter Umständen bereits bessere Alternativen vorliegen. Die Alpha-Beta-Suche verwendet nun zwei Variablen (Alpha und Beta), um die bestmöglichen spieltheoretischen Werte bereits untersuchter Teilbäume für beide Spieler festzuhalten. Ist es absehbar, dass der aktuelle Teilbaum für einen der Opponenten nicht als Option in Frage kommt, da dieser bereits eine bessere Alternative kennt, muss der entsprechende Knoten nicht weiter expandiert werden (*Cutoff*). Je eher für beide Spieler die bestmöglichen Varianten gefunden werden, desto mehr und größere Teilbäume lassen sich abschneiden. Daher spielt die *Zugsortierung* eine entscheidende Rolle für die Effizienz der Alpha-Beta-Suche.

Beim Vier-Gewinnt-Spiel kann bereits mit einer simplen Zugsortierung, die lediglich zentrale Spielfeldzellen bevorzugt behandelt, eine hohe Cutoff-Rate erzielt werden. Auch die Bevorzugung von Spielzügen, die (nicht unmittelbare) Drohungen erzeugen und weitere kleinere Maßnahmen beschleunigen die Zugsortierung.

Da die Umstellung von Zugfolgen oft zu ein und derselben Stellung führt, erreicht die Baumsuche viele Spielstellungen mehrfach. Eine erneute Untersuchung solcher Spielstellungen sollte vermieden werden, da ja bereits ein Ergebnis hierfür vorliegt bzw. vorlag. Mithilfe von *Transpositionstabellen* können die spieltheoretischen Werte von Spielstellungen abgespeichert und bei Bedarf wieder entnommen werden um die mehrfache Analyse von Spielstellungen zu vermeiden. Mithilfe der Enhanced Transposition Cutoffs (ETC) [26] lässt sich die Zahl der Cutoffs mithilfe der Transpositionstabellen noch erhöhen.

Da speziell beim *Vier-Gewinnt*-Spiel Eröffnungsstellungen – aufgrund des deutlich größeren Zustandsraumes – deutlich schwieriger zu untersuchen sind, als Endspielsituationen, bietet sich insbesondere für *Vier Gewinnt* an, Eröffnungsdatenbanken zu verwenden. Daher wurde unter anderem eine Datenbank berechnet, die alle notwendigen Spielstellungen mit 12 Steinen enthält und die dazugehörigen spieltheoretischen Werte.

Eine eher technische Optimierung der Suche stellen die sogenannten Bitboards dar. Hierbei repräsentieren mehrere Bitfelder (genau zwei für *Vier Gewinnt*) das Spielfeld. Jedes Bitfeld nimmt jeweils Spielsteine gleicher Art (gelbe bzw. rote Steine) auf, wobei ein Bit einer Zelle des Spielfeldes zugeordnet wird.

Wie stark sich jede Optimierung letztendlich auf die Laufzeit der Baumsuche auswirkt, kann nicht ohne Weiteres bestimmt werden, da die einzelnen Optimierungsschritte zum Teil wieder andere in ihrer Effektivität beeinflussen.

Die Alpha-Beta-Suche in Verbindung mit der guten Zugsortierung konnte deutlich über 90% der Knoten im Vergleich zur Minimax-Suche einsparen und stellt daher die zentrale Optimierung des Minimax-Agenten dar.

Durch die Einführung der Transpositionstabellen entsteht ein – nicht zu vernachlässigender Overhead – der allerdings durch die große Zahl an Cutoffs mehr als kompensiert wird. Die Laufzeit kann durch die Transpositionstabellen im Schnitt etwa halbiert werden.

Mithilfe der eingesetzten Eröffnungsdatenbanken lassen sich alle Spielstellungen mit 12 oder weniger Spielsteinen innerhalb weniger Millisekunden analysieren (die Untersuchung des leeren Spielfeldes ohne Eröffnungswissen benötigt ca. 3-4 Minuten auf einem Pentium4-Rechner). Nach Verlassen der Eröffnungsphase ist die Alpha-Beta-Suche bereits in der Lage, praktisch jede Spielsituation in deutlich weniger als einer Sekunde zu untersuchen (in der Regel sogar nur wenige Millisekunden).

Auch der Einsatz von Bitboards halbiert die Laufzeit der Suche um knapp die Hälfte.

Zu beachten ist, dass der hier beschriebene Minimax-Agent in *keiner Weise* in das Training eingreifen soll, sondern ausschließlich *Vergleichszwecken* dient.

4.1.4 Einschätzung der Spielstärke

Bevor mit den verschiedenen Untersuchungen begonnen werden kann, ist zunächst noch eine wichtige Frage zu beantworten: Wie kann man die Spielstärke eines *Vier Gewinnt*-Agenten geeignet messen? Es muss ein Mechanismus gefunden werden, der die Spielstärke in irgendeiner Weise möglichst objektiv beurteilt und eine gewisse Vergleichbarkeit zwischen verschiedenen Agenten möglich macht.

Um die Vergleichbarkeit zu erreichen, bietet es sich beispielsweise an, anhand gewisser Kriterien eine *Wertungszahl* zu bestimmen (z.B. eine natürliche Zahl). So wurde etwa beim Schachspiel die sogenannte *ELO-Zahl* eingeführt, die die Spielstärke für Schachspieler beschreibt.

Nun muss noch geklärt werden, welche Kriterien für eine Bewertung beim *Vier Gewinnt* Spiel herangezogen werden. Wertungsspiele gegen einen Menschen kommen nur bedingt in Frage, da deren Spielstärke unbekannt ist und die Spiele sehr zeitaufwändig wären. Dies gilt auch für viele (frei verfügbaren) Programme.

Aus diesem Grund wurde entschieden, den in Abschnitt 4.1.3 beschriebenen Agenten zu benutzen, der auf der klassischen Baumsuche basiert. Dieser Agent verfügt über umfangreiches Eröffnungswissen und verwendet Transpositionstabellen sowie einige weitere Techniken, die exakte Stellungsbewertungen innerhalb von Sekundenbruchteilen ermöglichen. Daher erscheint dieser Agent besonders als Vergleichsmöglichkeit geeignet, ohne jedoch in den eigentlichen Trainingsprozess einzugreifen (es findet kein überwachtes Lernen statt).

Um nun die Spielstärke eines *TDL*-Agenten zu bestimmen, werden 50 Spiele – vom leeren Spielfeld aus – gegen den Minimax-Agenten durchgeführt. Da der anziehende Spieler (Gelb) bei perfektem Spiel immer gewinnt, kommt der (perfekte) Minimax-Agent nur als Nachziehender in Frage.

Die Vergleichsspiele könnten zwar auch ausgehend von anderen Positionen als dem leeren Spielfeld erfolgen. Allerdings gibt es hierbei ein Problem: Der *TDL*-Agent startet jedes Trainingsspiel von einem leeren Feld aus, schließlich besteht das Hauptziel darin, das Spiel ausgehend von dieser Position zu beherrschen. Folgestellungen, die der *TDL*-Agent – aufgrund der Bewertung durch die Spielfunktion – als wenig erstrebenswert betrachtet, können nur dann erreicht werden, wenn es sich um explorative Halbzüge handelt. Da die Explorationsrate normalerweise im Laufe des Trainings sehr stark abfällt, kommt es – aufgrund des approximierenden Verhaltens der Spielfunktion – daher dazu, dass die Gewichte der unwahrscheinlichen Spielstellungen zugunsten der relevanten Stellungen angepasst werden, um den durchschnittlichen Fehler für letztere zu minimieren. Stellungen, die aus Sicht des *TDL*-Agenten (ausgehend vom leeren Spielfeld) irrelevant sind, werden daher im Laufe des Trainings verlernt.

Erlernt der Agent beispielsweise, dass sein erster Stein in die mittlere Spalte gesetzt werden muss, um das Spiel zu gewinnen, wird er den ersten Halbzug nur dann in ande-

re Spalten setzen, wenn er zu einem explorativen Zug gezwungen wird. Im Regelfall untersucht der Agent jedoch immer nur den einen Teilbaum, bei dem er seinen ersten Spielzug in die mittlere Spalte gesetzt hat. Die weiteren sechs Teilbäume werden vernachlässigt, sodass nach und nach viele Stellungen (abgesehen von den Transpositionen) innerhalb dieser verlernt bzw. nie erlernt werden können.

Daher macht es im Regelfall wenig Sinn, auf Grundlage anderer Stellungen als dem leeren Spielfeld Vergleichsspiele durchzuführen, da der Agent hierfür nicht trainiert wurde.

Um das ständige Ablaufen des gleichen Spiels zu vermeiden – sowohl Minimax- als auch *TDL*-Agent verhalten sich im Regelfall nahezu vollständig deterministisch – spielt der Minimax-Agent in der Eröffnungsphase zufällige Varianten, sofern er mehrere gleichgute Varianten zur Verfügung hat. Unabhängig hiervon werden Fehler des *TDL*-Agenten unmittelbar bestraft; bei Fehlern die eine Niederlage erwarten lassen, kann das Spiel vorzeitig beendet werden, da der Minimax-Agent einen Sieg nicht wieder hergeben wird.

Für jedes Spiel, das der *TDL*-Agent gewinnt, bekommt dieser einen Punkt. Auch Unentschieden werden mit 0,5 Punkten honoriert[17]; für Niederlagen gibt es keinen Punkt. Anschließend bestimmt das Programm anhand des arithmetischen Mittelwertes aller 50 Teilbewertungen die Wertungszahl des Agenten, die im Idealfall bei 1,0 liegt.

Während des Trainings wird, sofern nicht anders erwähnt, jeweils im Abstand von 10^5 Trainingsspielen die Spielstärke des *TDL*-Agenten geschätzt und aufgezeichnet. Um die Reproduzierbarkeit der Trainingsergebnisse zu überprüfen, wird das Training für eine Konfiguration zehnmal wiederholt und im Anschluss das *arithmetische Mittel* und die *Standardabweichung* für diese zehn Durchgänge ermittelt und ins Diagramm eingetragen.

Wichtig ist, dass der *TDL*-Agent – sofern nicht anders erwähnt – keine Baumsuche irgendeiner Art verwendet. Um einen geeigneten Folge-Zug zu ermitteln, wird lediglich der aktuelle Spielknoten expandiert und die Folgezustände durch das N-Tupel-System bewertet.

4.1.5 Festlegung der Basiskonfiguration für das Training von Vier Gewinnt

Bevor tatsächlich mit den einzelnen Tests begonnen wird, ist es zunächst sinnvoll, eine Basiskonfiguration des *TDL*-Agenten festzulegen, von der aus man in unterschiedliche Richtungen Untersuchungen anstellen kann. So lassen sich die gemachten Ergeb-

[17] Unentschieden treten allerdings äußerst selten auf, in den meisten Fällen maximal in einem bis zwei von 50 Spielen.

nisse leichter bewerten und in einem Zusammenhang mit den anderen Resultaten bringen.

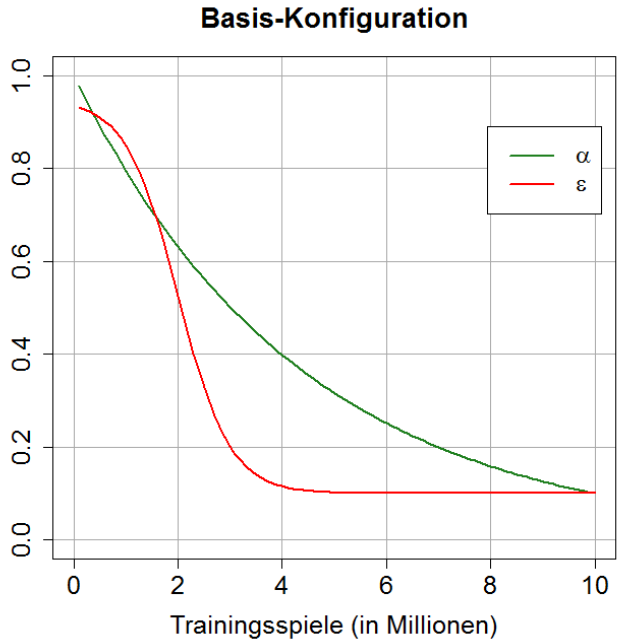


Abbildung 4.3 Verlauf der Explorationsrate ε und der Lernschrittweite α für die Basis-Konfiguration des *TDL*-Agenten. Für ε kommt eine Sigmoid-Funktion zum Einsatz, α fällt exponentiell auf den Endwert ab.

Sofern nicht anders beschrieben, finden standardmäßig folgende Trainingseinstellungen für den *TDL*-Agenten Verwendung, die nach einigen Vorversuchen als geeignet erschienen:

- Je Trainingsdurchgang werden 10 Mio. *Self-Play-Spiele* absolviert.
- Das N-Tupel-System besteht aus *70 verschiedenen 8-Tupel*, die mittels der Generierungsvorschrift *"Random Walk"* (acht zusammenhängende Abtastpunkte) erzeugt wurden. Es werden daher in jedem Fall die gleichen 70 Tupel eingesetzt.
- Die *Lernschrittweite* α fällt exponentiell von $\alpha_{Start} = 0{,}01$ auf $\alpha_{Final} = 0{,}001$.
- Der Verlauf der *Explorationsrate* ε folgt dem Verlauf einer Sigmoidfunktion (vgl. Formel (4.1)) und fällt von $\varepsilon_{Start} = 0{,}95$ auf $\varepsilon_{Final} = 0{,}1$. Der Wendepunkt der Funktion liegt bei $n_{wp} = 2 \cdot 10^6$ Trainingsspielen.
- Die Verwendung von Symmetrien ist aktiviert.
- Alle Gewichte des N-Tupel-Systems werden mit Null initialisiert.
- Es wird keine Baumsuche eingesetzt (Suchtiefe von einem Halbzug).

4.2 Untersuchungen für eine reduzierte Spielkomplexität

Nachdem alle nötigen Trainingsalgorithmen sowie das dazugehörige N-Tupel-System implementiert sind, müssen noch die im Programmcode vorhandenen Fehler ausgemerzt werden. Einige dieser Fehler kann man durch eine gezielte Untersuchung einzelner Methoden lokalisieren, letztendlich lassen sich jedoch einige Testläufe, die alle Trainingskomponenten – in direktem Zusammenspiel miteinander – untersuchen, nicht vermeiden.

Soll der *TDL*-Agent einen Trainingsdurchlauf für das Spiel *Vier Gewinnt* vornehmen, treten eine Reihe von Problemen auf, die die Fehlersuche deutlich erschweren. Beispielsweise wäre aufgrund der großen *Zustandsraumkomplexität* von *Vier Gewinnt* eine große Zahl an Trainingsspielen nötig, bis erste Erfolge zustande kämen. Die Analyse des Trainings – z.B. mithilfe eines Debuggers – müsste sich allerdings aus Zeitgründen auf wenige Spiele beschränken.

Weiterhin deuten schlechte Trainingsresultate nicht immer zwangsläufig auf Programmierfehler hin. Bei der Vielzahl an Trainingsparametern reichen oft schon kleine Änderungen an den Einstellungen dazu aus, dass das Training vollständig fehlschlägt. Auch geeignete Parameterkonfigurationen sind zu Beginn noch unbekannt.

Nicht zuletzt erschweren die vielen Zufallskomponenten im Training die Untersuchungen, da kein Trainingsspiel dem Anderen gleicht.

Um die oben genannten Probleme zumindest teilweise zu umgehen, werden im Folgenden zwei Ansätze diskutiert, die das Training vereinfachen. Zum Einen wird das Brettspiel *Tic Tac Toe* betrachtet, das dem *Vier Gewinnt* sehr ähnlich ist, aber eine wesentlich kleinere Zustandsraumkomplexität besitzt.

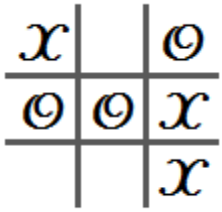
Weiterhin kann man die Komplexität des Trainings dadurch reduzieren, dass der *TDL*-Agent lediglich Eröffnungs- bzw. Endspielphasen erlernt. Dies soll im Anschluss diskutiert werden.

Eine weitere Möglichkeit bestünde darin, das Spielfeld des *Vier-Gewinnt*-Spiels zu verkleinern. Dieser Ansatz soll hier jedoch nicht weiter verfolgt werden.

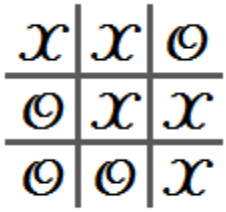
4.2.1 Anwendung des TD-Learning mit N-Tupel-Systemen auf Tic Tac Toe

Tic Tac Toe ist ein Strategiespiel für zwei Personen, dass dem *Vier Gewinnt* sehr ähnlich ist. Jedoch wird das Spiel auf einem Spielfeld mit nur drei Zeilen und drei Spalten gespielt. Auch die *Schwerkraftskomponente* fällt weg, sodass jedes leere Feld direkt belegbar ist.

Die beiden Kontrahenten tragen jeweils abwechselnd ihr Zeichen in eines der leeren Felder. Der anziehende Spieler verwendet hierbei im Regelfall ein Kreuz, der Nachziehende einen Kreis. Ziel beider Spieler ist es, als erstes drei eigene Zeichen in einer Reihe zu platzieren, wodurch das Spiel gewonnen ist (vgl. Abbildung 4.4); dies kann horizontal, vertikal oder diagonal geschehen. Gelingt das keinem der beiden Kontrahenten, so geht das Spiel nach dem letzten möglichen Zug Unentschieden aus. Bei perfektem Spiel Beider ist das immer der Fall.



(a) Eine typische *Tic Tac Toe* Stellung nach sechs Halbzügen.



(b) Der anziehende Spieler gewinnt das Spiel mit seinem letzten Zug.

Abbildung 4.4. Beispielstellungen für das Spiel *Tic Tac Toe*.

Da *Tic Tac Toe* nur 5478 ([9] , S. 159) legale Stellungen besitzt und aufgrund der Ähnlichkeiten zum *Vier Gewinnt*, scheint dieses Spiel daher gut geeignet, um hieran erste Versuche mit dem *TDL*-Agenten durchzuführen.

Für die erste Untersuchung wurde ein einzelnes N-Tupel der Länge $N = 9$ gewählt. Das Tupel deckt daher das komplette Spielfeld ab. Bei drei möglichen Zuständen (leer, X und O) der einzelnen Abtastpunkte ergibt sich eine Größe der zugehörigen Look-up-Tabelle von $3^9 = 19683$ Gewichten. Der Vorteil dieser Wahl liegt darin, dass jeder Spielzustand auf genau ein Gewicht der *LUT* abgebildet wird, wodurch man widersprüchliche Lernsignale während des Trainings vermeidet. Diese Darstellung entspricht einer einfachen tabellarischen Spielfunktion.

Bei ausreichender Anzahl an Trainingsspielen sollten die Gewichte aller *nonterminalen Stellungen* adressiert – und zum Großteil auch verändert – worden sein[18].

[18] Da beim TD-Learning (vgl. Abschnitt 2.4.1) nach Erreichen von terminalen Stellungen der aktuelle Trainingsvorgang abgebrochen, gleichzeitig aber der Lernschritt nur für den vorherigen Spielzustand durchgeführt wird, kann die Spielfunktion keine Aussage zu terminalen Stellungen machen. Für diese Zustände befragt man die Reward-Funktion $R(s_t)$.

Tic Tac Toe besitzt genau 4520 solcher non-terminalen Spielzustände[19]; während des Trainings können daher (max.) 4520 der insgesamt 19683 Gewichte innerhalb der *LUT* trainiert werden. Sind nach Training mehr Gewichte geändert worden, liegt ganz offensichtlich ein Fehler im Programm vor.

Nach der Korrektur diverser Programmierfehler und der Ermittlung geeigneter Trainingsparameter konnte ein sehr starker *TDL*-Agent trainiert werden (siehe Abbildung 4.5). Auch hier wird – wie beim *Vier Gewinnt* – ein perfekt spielender Minimax-Agent als Vergleichsmöglichkeit herangezogen. Niederlagen gegen den Minimax-Agenten bewertet die Evaluierung mit dem Wert -1 und Unentschieden mit 0. Siege gegen den Minimax-Spieler sind nicht möglich. Das Endresultat des *TDL*-Agenten liegt daher im Intervall $[-1;\ 0]$. Für exaktere Ergebnisse wird das Training zehnmal wiederholt und in den Diagrammen das *arithmetische Mittel* und die *Standardabweichung* aufgetragen.

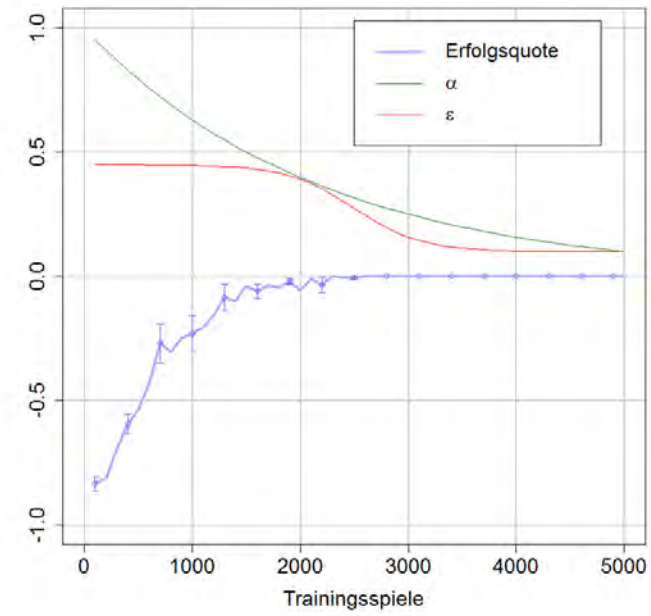


Abbildung 4.5. Erste Untersuchungen für einen *TDL*-Agenten mit N-Tupel-System. Dargestellt ist die *Erfolgsquote* des Agenten mit einem 9-Tupel für das Spiel *Tic Tac Toe*; weiterhin sind die *Explorationsrate* ε sowie die *Lernschrittweite* α aufgetragen. Für das Lernen nutzt das System Spiegel- und Rotationssymmetrien aus. Dadurch erreicht der Agent nach etwa 2500 Trainingsspielen die bestmögliche Erfolgsquote.

Für die ersten Untersuchungen hat sich das 9-Tupel als gut geeignet erwiesen, vor allem deshalb, weil die Wahl der Trainingsparameter eine eher untergeordnete Rolle gespielt hat. So konnte man auch ohne umfangreiche Tests gute Trainingsresultate erzielen.

[19] Die Berechnung für die Anzahl der non-terminalen Stellungen kann im Anhang nachgeschlagen werden. Weiterhin ist dort auch ein Java-Programm (*diverses.CountPositions*) aufgeführt, das die errechnete Zahl bestätigt.

Abschließend soll noch ein *TDL*-Agent vorgestellt werden, der mit lediglich vier 3-Tupel (Abbildung 4.6) perfektes Spiel erlernt. Insgesamt enthält das N-Tupel-System $4 \cdot 3^3 = 108$ Gewichte, also deutlich weniger als das zuvor diskutierte System mit nur einem 9-Tupel.

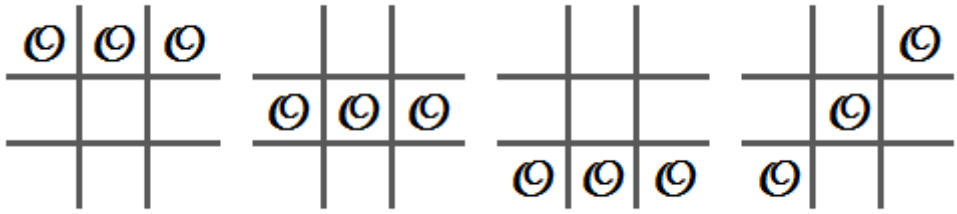


Abbildung 4.6. Abgebildet sind die Abtastpunkte der vier 3-Tupel, die für einen perfekt spielenden *TDL*-Agenten ausreichend sind.

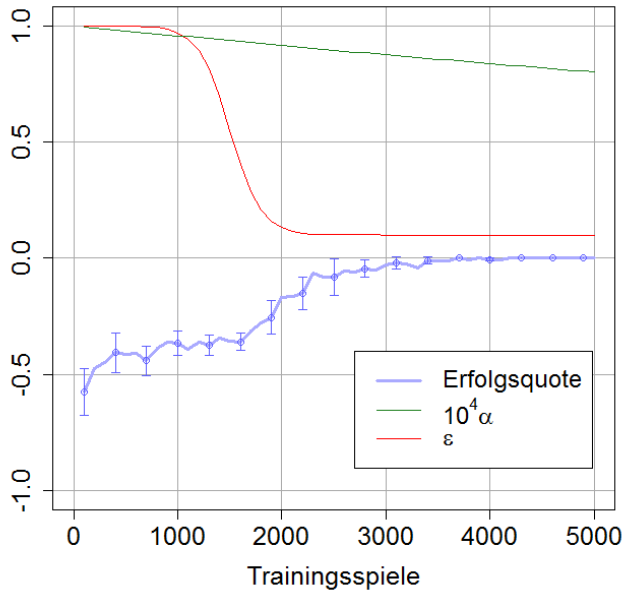


Abbildung 4.7. Training des *TDL*-Agenten für *Tic Tac Toe*. Das N-Tupel-System basiert auf vier 3-Tupel . Nach etwa 4000 Trainingsspielen wird perfektes Spiel erreicht. Wie im Beispiel zuvor nutzt das System Spiegel- und Rotationssymmetrien aus.

Für das Training (Resultat in Abbildung 4.7) wurden wieder Spiegel- sowie Rotationssymmetrien ausgenutzt, die Optimierung der Explorationsrate ε und der Lernschrittweite α gestaltete sich jedoch etwas schwieriger. Als Anpassungsfunktion für ε kommt eine Sigmoidfunktion zum Einsatz, α verhält sich nahezu linear. Zwar spielt der *TDL*-Agent nach etwa 4000 Trainingsspielen perfekt, die Lerngeschwindigkeit liegt jedoch unter dem des 9-Tupel-Systems.

4.2.2 Beschränkung des Trainings auf die Eröffnungsphase von Vier Gewinnt

Nachdem das TD-Learning mit N-Tupel-Systemen erfolgreich auf das triviale Spiel *Tic Tac Toe* angewendet wurde, sind nun erste Untersuchungen am Spiel *Vier Gewinnt* zweckmäßig. Allerdings ist es zu Beginn noch nicht empfehlenswert, über den kompletten Zustandsraum des Spiels zu trainieren.

Stattdessen soll der *TDL*-Agent zunächst Eröffnungsphasen bis zu einer geringen Anzahl an Halbzügen erfolgreich erlernen, da der Zustandsraum hierdurch stark verkleinert ist (siehe Tabelle 4.1). Erreicht das TD-Learning die vorgegebene Halbzüge-Grenze, bricht die Trainingsroutine das *Self-Play-Spiel* ab und vergibt vorzeitig den entsprechenden Reward. Hierzu muss ein Minimax-Agent den theoretischen Spielausgang bestimmen. Da hier jedoch nur Eröffnungsphasen mit bis zu einem Dutzend Spielsteinen betrachtet werden, kann der Minimax-Agent den Reward innerhalb kürzester Zeit ermitteln, indem er auf das umfangreiche Eröffnungswissen zurückgreift (vgl. [24] , S. 40).

Bei einer Begrenzung der Trainingsspiele auf beispielsweise zwei Halbzüge, sollte der Agent bereits nach wenigen Iterationen lernen, dass der erste Halbzug des Anziehenden in die mittlere Spalte gesetzt werden muss. Ist dieser erfolgreich, kann man die Halbzüge-Grenze sukzessive erhöhen und die Untersuchungen wiederholen. Dadurch vergrößert sich der Zustandsraum Schritt für Schritt und die Trainingskomplexität nimmt kontinuierlich zu.

Tabelle 4.1. Anzahl der Blattknoten bzw. die Zustandsraumkomplexität für eine vorgegebene Anzahl an Halbzügen[20]. Auch bei einer Begrenzung auf max. 12 Spielsteine, ist die Zustandsraumkomplexität noch etwa um den Faktor $2{,}5 \cdot 10^5$ kleiner als die des vollständigen Spiels mit bis zu 42 Steinen (letzter Wert aus [8] entnommen).

Halbzüge	Blattknoten	Zustandsraum
0	1	1
1	7	8
2	49	57
3	238	295
4	1.120	1.415
5	4.263	5.678
6	16.422	22.100
7	53.955	76.055
8	181.597	257.652
9	534.085	791.737
10	1.602.480	2.394.217
11	4.231.877	6.626.094
12	11.477.673	18.102.767
⋮	⋮	⋮
42	?	$4.531.985.219.092 \approx 4{,}53 \cdot 10^{12}$

Für die einzelnen Trainingsdurchgänge wird folgende Konfiguration verwendet:

- Je Trainingsdurchgang werden eine Mio. *Self-Play-Spiele* absolviert
- In jedem Fall werden die *gleichen* 70 Tupel der Länge $N = 8$ verwendet, die mittels der Generierungsvorschrift *"Random Walk"* (acht zusammenhängende Abtastpunkte) erzeugt wurden.
- Die *Lernschrittweite* α fällt exponentiell von $\alpha_{Start} = 0{,}01$ auf $\alpha_{Final} = 0{,}001$.
- Der Verlauf der *Explorationsrate* ε folgt dem Verlauf einer Sigmoidfunktion (vgl. Formel (4.1)) und fällt von $\varepsilon_{Start} = 0{,}95$ auf $\varepsilon_{Final} = 0{,}1$. Der Wendepunkt der Funktion liegt bei $n_{wp} = 3{,}2 \cdot 10^5$ Trainingsspielen.
- Jedem N-Tupel wird genau eine *LUT* zugeordnet.
- Je Abtastpunkt sind drei Zustände (Gelb, Rot, Leer) möglich.
- Die Verwendung von Symmetrien ist aktiviert.
- Es wird keine Baumsuche eingesetzt (Suchtiefe von einem Halbzug).

Die Einschätzung der Spielstärke (vgl. Abschnitt 4.1.4) bleibt im wesentlichen gleich. Allerdings müssen die Spiele gegen den Minimax-Agenten selbstverständlich bei Erreichen der Halbzüge-Grenze angehalten und der theoretische Spielausgang bestimmt werden, um die geeignete Wertungszahl des *TDL*-Agenten zu ermitteln.

[20] Das Programm zur Berechnung der Werte ist im Anhang aufgeführt (*miscellaneous.CountPositionsC4*). Die Zustandsraumkomplexität für eine bestimmte Halbzüge-Grenze ergibt sich aus der Summe über die Anzahl aller Blattknoten bis dahin.

Einige Resultate für die Basis-Konfiguration sind in Abbildung 4.8 dargestellt (6,8,10 und 12-Halbzüge-Grenze). Auffallend ist, dass erste Lernerfolge erst sehr spät zu verzeichnen sind (nach ca. 400.000 Spielen). Als Ursache konnte vor allem die ungünstige Wahl der Explorationsrate (Wendepunkt) ausgemacht werden. Aber auch die Erfolgsquote erreicht – wider Erwarten – in keinem Fall die 100%-Marke.

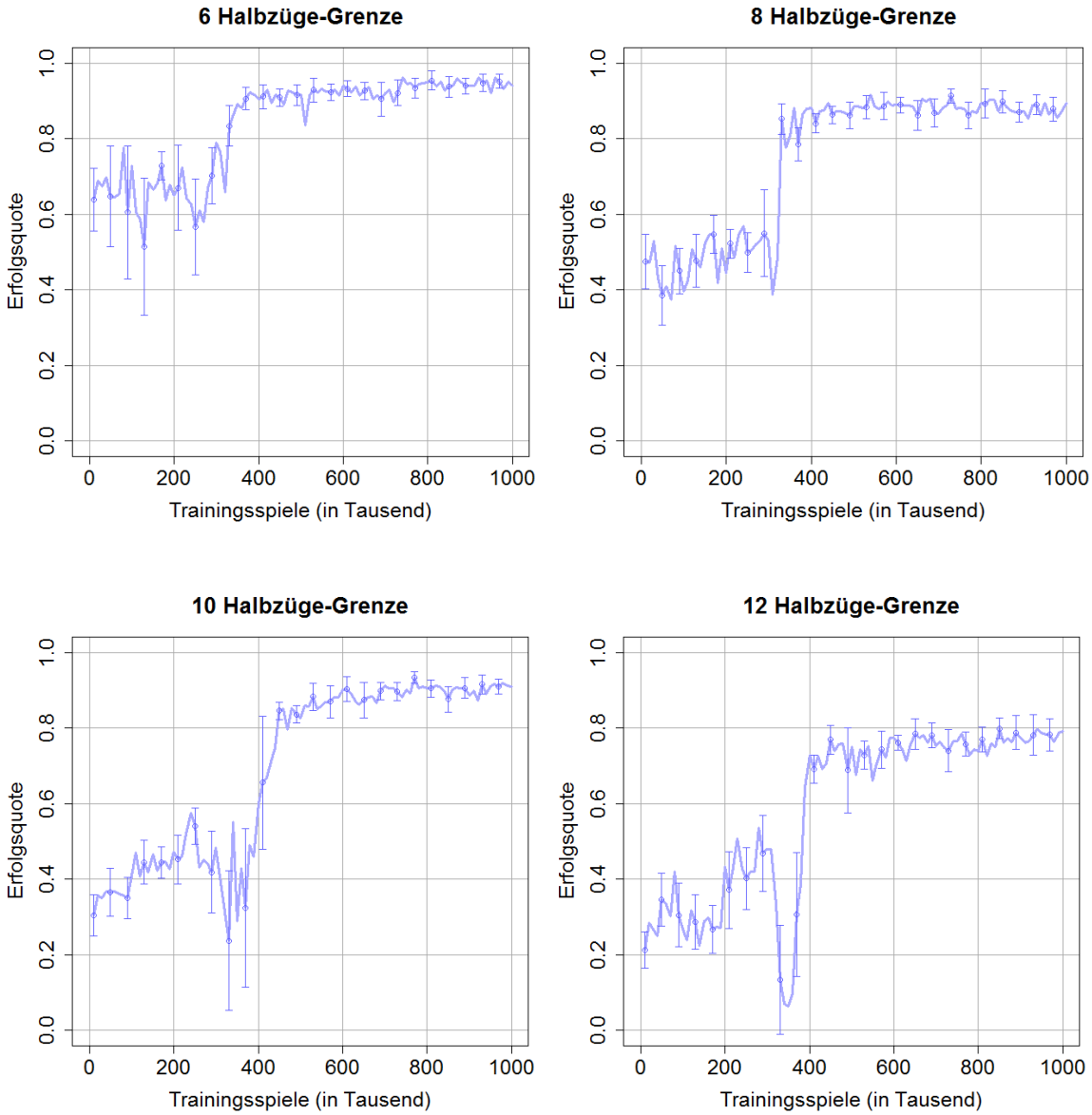
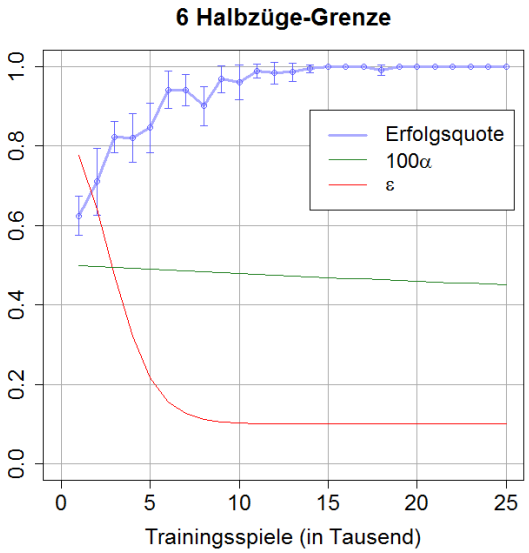


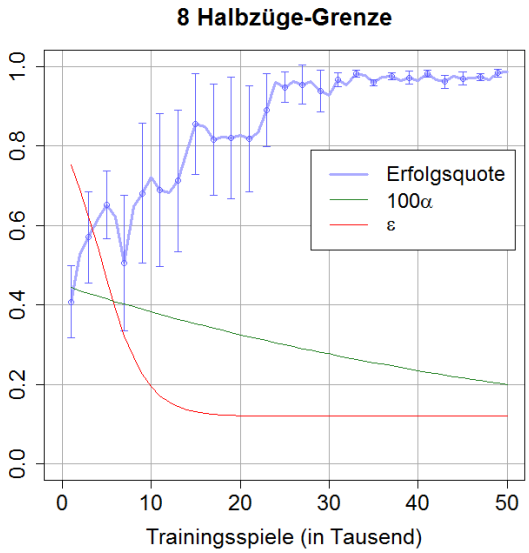
Abbildung 4.8. Trainingsresultate des *TDL*-Agenten für die einzelnen Eröffnungsphasen mit max. 6, 8, 10 und 12 Halbzügen. Bezeichnend ist in allen vier Fällen, dass vor allem die Lerngeschwindigkeit sehr schlecht ist. Das bestmögliche Resultat wird jeweils nach etwa 400.000 Trainingsspielen erreicht.

Weitere Untersuchungen konnten jedoch schnell zeigen, dass die Zahl der Trainingsspiele, sowie auch Anzahl und Länge der N-Tupel sich reduzieren lassen und gleichzeitig ein besseres Endresultat möglich ist. Dennoch ist nach wie vor eine ver-

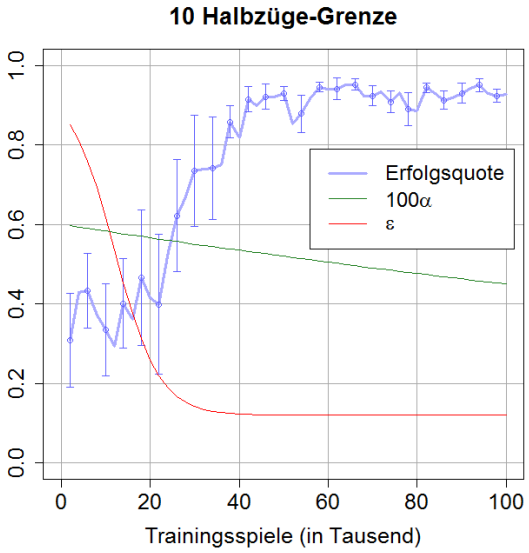
gleichsweise große Zahl an Trainingsspielen nötig. Die neuen Ergebnisse sind in Abbildung 4.9 dargestellt[21].



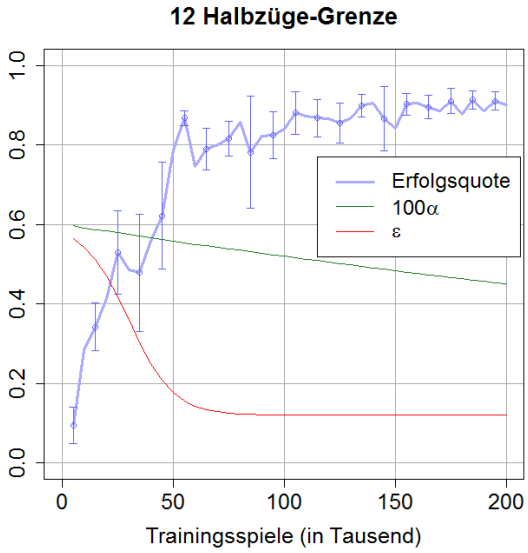
(a) Verwendung von 30 Tupel der Länge 5.



(b) Verwendung von 30 Tupel der Länge 6.



(c) Verwendung von 30 Tupel der Länge 7.



(d) Verwendung von 30 Tupel der Länge 7.

Abbildung 4.9. Neue Trainingsresultate für die einzelnen Eröffnungsphasen mit max. 6, 8, 10 und 12 Halbzügen. Für die vier Eröffnungsphasen wurden die Trainingsparameter jeweils unterschiedlich gewählt; so kommen beispielsweise unterschiedliche N-Tupel-Systeme zum Einsatz, die allerdings – einmal generiert – beibehalten werden. Die in *c)* und *d)* eingesetzten N-Tupel sind identisch.

[21] Genaue Konfigurationen, sowie die vollständigen Trainingsergebnisse sind im Anhang aufgeführt.

Als Grund für die nun deutlich besseren Ergebnisse lässt sich die Kombination verschiedener Faktoren ausmachen:

Zum Einen war der ursprüngliche Wendepunkt der Explorationsrate zu groß gewählt worden. Dies hat – bei einem Initialwert von $\varepsilon_{Start} = 0{,}95$ – zur Folge, dass zu Beginn des Trainings die Lernschritte fast ausschließlich bei Erreichen von terminalen Spielsituationen (Erreichen der Halbzüge-Grenze) stattfinden, da die Zahl der explorativen Züge deutlich überwiegt; das Lernen von non-terminalen Stellungen erfolgt in diesem Zeitraum kaum. Wenn man nun bedenkt, dass bei 6 Halbzügen nur etwa 16.000 Blattknoten möglich sind, der Wendepunkt der Explorationsrate jedoch bei $n_{wp} = 3{,}2 \cdot 10^5$ liegt, wird schnell ersichtlich, dass diese Spanne deutlich zu groß gewählt ist. Dies gilt auch für die größeren Halbzüge-Grenzen 8, 10 und 12; unter anderem deshalb, weil das TD-Learning nicht jede (terminale) Position besuchen muss, um diese zu erlernen (Generalisierung durch das N-Tupel-System).

Weiterhin erreicht man durch die Verkürzung der N-Tupel einen weiteren Vorteil: Wie man Abbildung 4.9 entnehmen kann, ist eine Tupel-Länge von $N = 8$ oder länger nicht zwingend nötig, um eine gute Approximation der Spielfunktion zu erhalten. Durch die Verkürzung der N-Tupel auf jeweils sechs bzw. sieben Abtastpunkte, verkleinert man die Anzahl der Gewichte um einen Faktor neun bzw. drei, wodurch weniger Lernschritte nötig sind, um erste Trainingserfolge des *TDL*-Agenten festzustellen. In gewisser Weise besteht daher ein Zusammenhang zwischen Tupel-Länge und Anzahl der nötigen Trainingsspiele.

Nicht zuletzt kann man mithilfe einiger Testläufe auch die Wahl der weiteren Parameter – vor allem für die Lernschrittweite α und die Explorationsrate ε – verbessern, um das Lerntempo und in kleinerem Maße die resultierende Spielstärke des Agenten in den jeweiligen Situationen zu steigern.

Letztendlich lässt sich keine generelle Aussage zur optimalen Konfiguration des *TDL*-Agenten machen, da die Wahl der einzelnen Parameter situationsabhängig ist und sich diese zum Teil auch gegenseitig beeinflussen.

Zunächst muss ein geeignetes Verhältnis zwischen explorativen Zügen und Greedy-Zügen gefunden werden, wobei die explorativen Züge zu Beginn des Trainings überwiegen sollten. Fällt die Explorationsrate zu früh ab, kann der Spielbaum nicht ausreichend erforscht werden, was sich häufig in einer geringeren Spielstärke bemerkbar macht. Fällt die Explorationsrate dagegen zu spät ab, ist das Lerntempo des *TDL*-Agenten oft nicht ausreichend.

Die Länge und die Anzahl der N-Tupel sollte der Größe des Zustandsraums angepasst werden. Sollte der Zustandsraum nur sehr klein sein, ist im Regelfall auch nur eine kleine Zahl von kürzeren N-Tupeln nötig, sodass sich in vielen Fällen auch die Zahl der Trainingsspiele reduzieren lässt.

Je nach Trainingsdauer kann auch etwas mit der Wahl der Lernschrittweite experimentiert werden.

4.3 Fehlende Codierung des Spielers am Zug

Nachdem der *TDL*-Agent für die Eröffnungsphase von *Vier Gewinnt* zufriedenstellende Ergebnisse liefert, wird nun dazu übergegangen, das vollständige Spiel zu erlernen. Allerdings sind die bestmöglichen Resultate – trotz verschiedenster Parameterkonfigurationen – zunächst sehr ernüchternd (Abbildung 4.10).

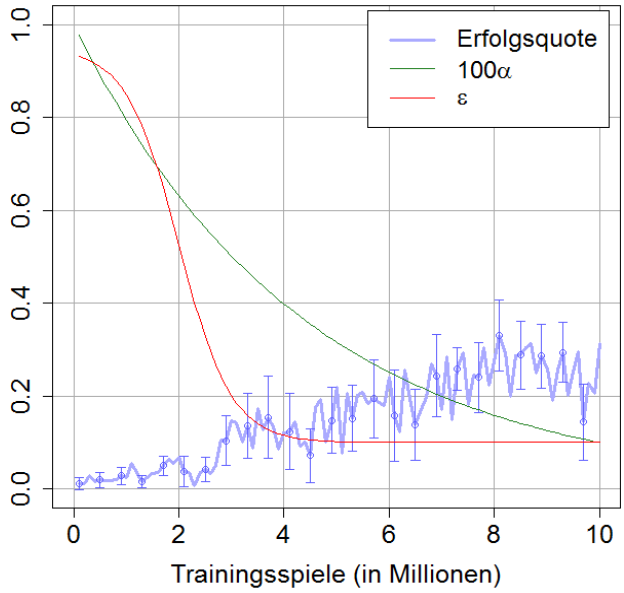


Abbildung 4.10. Trainingsresultat für die Basiskonfiguration des *TDL*-Agenten, allerdings noch mit genau einer *LUT* je N-Tupel und drei Zuständen pro Spielfeldzelle. Die Erfolgsquote liegt im Schnitt bei ca. 20-30% (nach 6 Mio. Spielen), der *TDL*-Agent kann also auch nach 10 Mio. Trainingsspielen kaum ein Spiel gegen den Minimax-Agenten gewinnen.

In den folgenden Abschnitten soll der wesentliche Grund für das Scheitern des Trainings beschrieben und im Anschluss ein Lösungsansatz vorgestellt werden, der die Spielstärke des *TDL*-Agenten deutlich verbessert.

4.3.1 Grundlegende Problematik der fehlenden Spielerinformation

Abbildung 4.11 enthält eine *Vier Gewinnt* Stellung mit einer unmittelbaren Drohung für den anziehenden Spieler. Da Spieler Rot am Zug ist, kann dieser die Drohung in seinem nächsten Halbzug neutralisieren und das Spiel würde regulär fortgesetzt; bei perfektem Spiel beider Kontrahenten endet das Spiel letztendlich Unentschieden.

Das gekennzeichnete 4-Tupel enthält jedoch nur *die* drei Steine, die die Drohung erzeugen und ansonsten keine weiteren Informationen. Es ist völlig unerheblich welcher Spieler am Zug ist, für diese Kombination wird immer der gleiche Eintrag in der zugeordneten Look-up-Tabelle angesprochen.

Sollte beispielsweise Spieler Rot einen Stein in die a-Spalte werfen, ändert sich die Belegung des 4-Tupels nicht. Spieler Gelb könnte das Spiel anschließend im nächsten Halbzug gewinnen. Man kann also keine Aussage darüber machen, wie diese Konstellation des 4-Tupels letztendlich zu bewerten ist, da sich in diesem Fall – abhängig vom aktuellen Spieler – unterschiedliche Spielresultate ergeben.

Genau dieses Problem wurde bei den Untersuchungen der *LUTs* immer wieder sichtbar, insbesondere dann, wenn Belegungen einzelner N-Tupel ganz offensichtlich auf einen Sieg im nächsten Halbzug hindeuteten, aber der entsprechende *LUT*-Eintrag eine gegenteilige Einschätzung signalisierte.

Weiterhin ist es naheliegend, dass diese Problematik nicht nur für N-Tupel gilt, die unmittelbare Drohungen enthalten; viele Ausschnitte des Spielzustandes lassen sich möglicherweise unterschiedlich bewerten, je nachdem welcher Spieler am Zug ist.

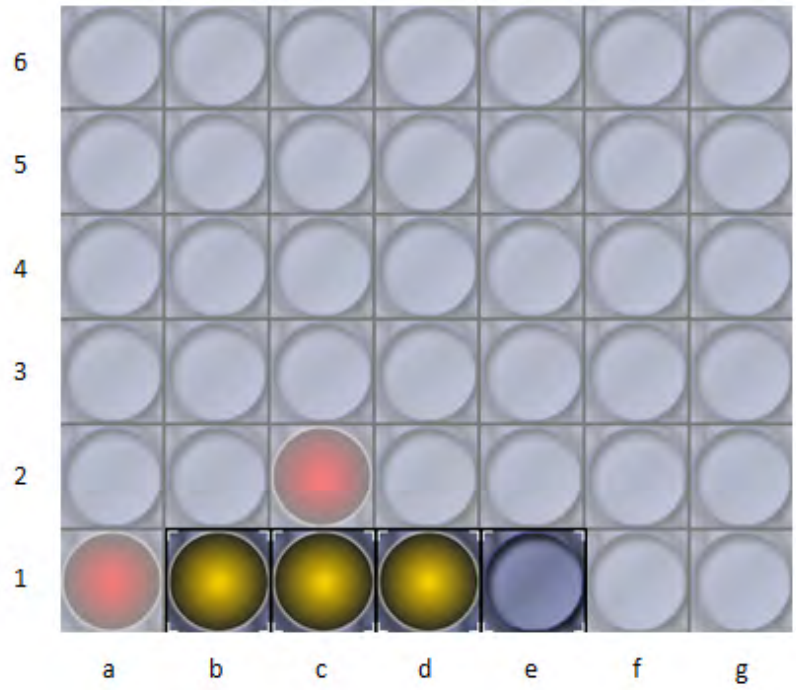


Abbildung 4.11. Stellung mit einer unmittelbaren Drohung für den anziehenden Spieler. Das betrachtete 4-Tupel wird durch die weißen Markierungen in den jeweiligen Spielfeldecken gekennzeichnet.

Da bis zu diesem Zeitpunkt keine Möglichkeit besteht, zwischen den beiden Spielern zu unterscheiden, geht diese Information innerhalb des N-Tupel-Systems verloren. Zwar nimmt man durch die Verwendung von N-Tupel-Systemen generell einen gewissen Informationsverlust in Kauf, den man jedoch dadurch zu kompensieren versucht, indem man durch wiederkehrende Muster innerhalb der N-Tupel einen Spielzustand einzuschätzen lernt. Die verlorene Spielerinformation scheint allerdings dazu zu führen, dass aufgrund vieler widersprüchlicher Lernsignale ein guter Lernprozess kaum möglich ist.

Eine Ausnahme stellt ein N-Tupel maximaler Länge dar, da dieses das komplette Spielfeld enthält[22]. Solch ein N-Tupel widerspricht jedoch dem Grundgedanken der hinter solch einem N-Tupel System steht und ist in der Praxis – aufgrund des hohen Speicherbedarfs – längst nicht immer realisierbar.

4.3.2 Einführung zweier Look-up-Tabellen je N-Tupel

Um das Problem der fehlenden Spielerinformation im N-Tupel-System lösen zu können, müssen die Lernsignale für beide Spieler getrennt behandelt werden, um zu vermeiden, dass die (möglicherweise gegensätzlichen) Signale auf dieselben Gewichte treffen. Eine Möglichkeit bestünde zum Beispiel darin, die Größe der aktuellen *LUTs* zu verdoppeln und dann – je nach Spieler – auf die erste oder die zweite Hälfte zuzugreifen.

In dieser Arbeit kommt ein etwas anderer Lösungsweg zum Einsatz: Für jedes N-Tupel erzeugt das System zwei *LUTs*, eine je Spieler. Beim Zugriff auf die Spielfunktion entscheidet das Programm dann anhand des Spielzustandes, welche der beiden Tabellen verwendet wird.

Tatsächlich ist der *TDL*-Agent mithilfe dieser sauberen Trennung der *LUTs* in der Lage, eine wesentlich höhere Spielstärke zu erreichen. So steigt die Erfolgsquote (bei ansonsten gleichen Trainingsbedingungen) von ehemals etwa 25% auf über 80% an (Abbildung 4.12). Auch die Standardabweichung ist zum Ende des Trainings hin kleiner, was auf eine bessere Reproduzierbarkeit der zehn Trainingsdurchläufe hindeutet.

[22] Mit der Einschränkung, dass der Spieler am Zug eindeutig aus der aktuellen Position bestimmbar sein muss. Bei *Vier Gewinnt* oder *Tic Tac Toe* ist das der Fall, beim *Schachspiel* allerdings nicht.

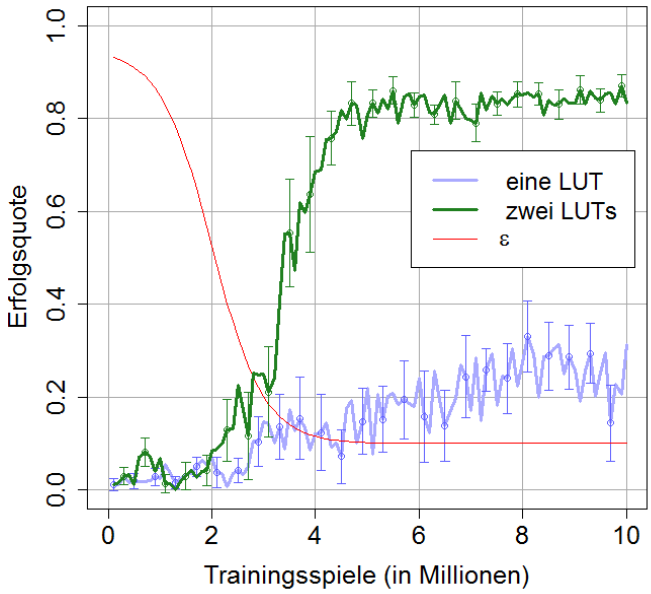


Abbildung 4.12. Trainingserfolg eines *TDL*-Agenten, bei Verwendung von zwei *LUTs* je N-Tupel bzw. einer *LUT* je Spieler. Zum Vergleich ist noch einmal die Lernkurve eines Agenten aufgetragen, der nur eine *LUT* je N-Tupel nutzt. Die übrigen Trainingsparameter wurden beibehalten.

4.3.3 Zusammenfassung

Beim Einsatz von N-Tupel Systemen kann ein Problem aufgrund der Tatsache auftreten, dass für jedes N-Tupel lediglich Ausschnitte des Spielzustandes erfasst werden. In gewisser Weise ist dies auch das Ziel eines N-Tupel Systems, man möchte ja schließlich das *Gesamtbild* in kleinere *Bruchstücke* zerlegen, um in diesen nach gewissen Mustern und Merkmalen zu suchen.

Probleme können jedoch unter Umständen dann auftreten, wenn der Spieler am Zug nicht mit codiert wird.

Die einzelnen N-Tupel enthalten lediglich eine Folge von Abtastpunkten, aus denen man den aktuellen Spieler im Regelfall nicht wieder rekonstruieren kann. Im Regelfall lässt sich der aktuelle Spieler daher nicht ohne weiteres aus einer Teilmenge von Abtastpunkten extrahieren, sodass für zwei identische N-Tupel-Folgen auf denselben *LUT*-Eintrag zugegriffen wird, obwohl sich die beiden zugrundeliegenden Spielzustände hinsichtlich des aktuellen Spielers möglicherweise unterscheiden.

Zusammenfassend lässt sich sagen, dass durch die Einführung zweier *LUTs* je N-Tupel eine signifikante Steigerung der Spielstärke des *TDL*-Agenten zu verzeichnen ist. Daher kommt diese Verbesserung – sofern nicht anders erwähnt – auch in den folgenden Untersuchungen zum Einsatz.

4.4 Differenzierte Betrachtung leerer Spielfeldzellen

Viele Zweipersonen-Brettspiele wie *Mühle*, *Dame*, *Othello* oder auch *Vier Gewinnt* kommen mit zwei Arten von Spielsteinen aus. Dadurch nehmen die Zellen des Spielfeldes in den meisten Fällen einen von drei Zuständen an (Farbe X, Farbe Y und Leer). Diese Zustände können im Regelfall auch für die Abtastpunkte eines N-Tupel-Systems übernommen werden ($m = 3$, für die Indexierungsfunktion Formel (2.6)). Für *Vier Gewinnt* ist beispielsweise die Codierung $0 = \text{Leer}$, $1 = \text{Gelb}$ und $2 = \text{Rot}$ möglich. Allerdings sind noch weitere Alternativen denkbar. Eine Möglichkeit soll im Weiteren beschrieben werden.

4.4.1 Verwendung von vier möglichen Zuständen je Abtastpunkt

Da beim *Vier-Gewinnt*-Spiel die Schwerkraftsregel zum Tragen kommt, sind nicht alle leeren Felder direkt im nächsten Halbzug erreichbar. Daher kann es auch Sinn machen, zwischen leeren, erreichbaren Zellen und leeren, unerreichbaren Zellen zu unterscheiden, sodass jeder Abtastpunkt des N-Tupel-Systems, statt ursprünglich drei, nun theoretisch vier Zustände ($m = 4$) annehmen könnte (siehe Abbildung 4.13). Dadurch ergibt sich die Codierung $0 = \text{Leer (Unereichbar)}$, $1 = \text{Gelb}$, $2 = \text{Rot}$ und $3 = \text{Leer (Erreichbar)}$. Allerdings vergrößert sich dadurch die einzelnen Look-up-Tabellen deutlich; der Speicherbedarf einer *LUT* für ein 8-Tupel ist beispielsweise zehnmal größer, wenn von vier Zuständen ausgegangen wird. Das Verhältnis zwischen den beiden *LUT*-Größen bei einer Länge N des Tupels beträgt allgemein:

$$\frac{|LUT_{m=3}|}{|LUT_{m=4}|} = \left(\frac{3}{4}\right)^N \tag{4.6}$$

Hierbei gibt $|LUT|$ die Anzahl der Gewichte innerhalb der entsprechenden *LUT* an. Ob $m = 3$ oder $m = 4$ mögliche Zustände je Zelle gewählt werden und ob die Resultate einen höheren Speicherbedarf rechtfertigen, liegt im Ermessen des Programmbenutzers.

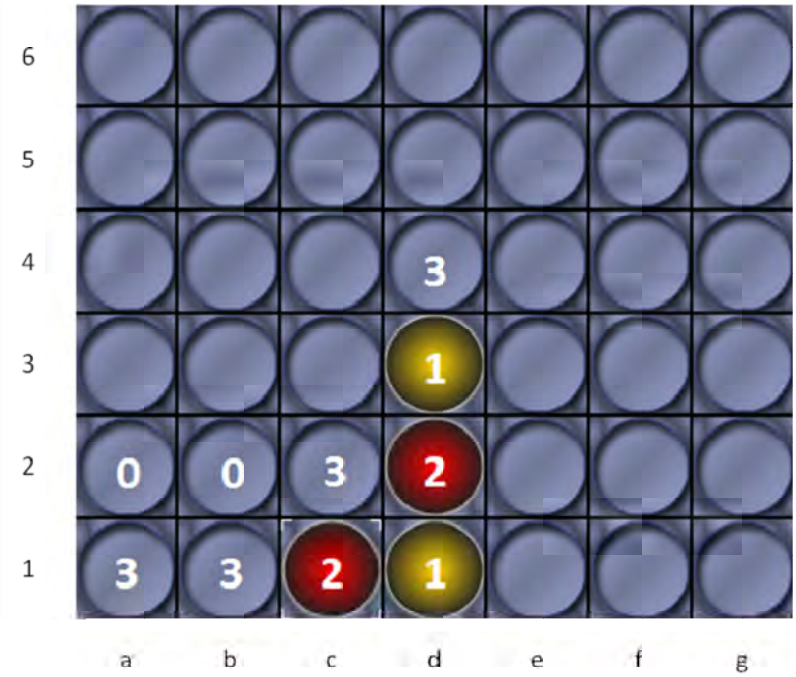


Abbildung 4.13. Zustände einzelner Zellen für eine Spielstellung mit der Codierung $m = 4$. Leere Zellen in der untersten Zeile sind in jedem Fall im nächsten Halbzug (direkt) erreichbar (Zustand: 3). Ansonsten sind leere Zellen erst dann direkt erreichbar, wenn sich unmittelbar darunter ein Spielstein befindet. Zellen, die nicht im nächsten Halbzug belegt werden können, besitzen automatisch den Zustand 0.

Für das Training des Standard *TDL*-Agenten mit der Variante $m = 4$ ist eine höhere Lerngeschwindigkeit im Vergleich zur Codierung $m = 3$ zu beobachten (Abbildung 4.14). Weiterhin ist das Endresultat auch besser und die Standardabweichung etwas kleiner (gute Reproduzierbarkeit des Trainings).

In Abbildung 4.15 wird erneut Erfolgsquote eines *TDL*-Agenten dargestellt, der nur eine LUT je N-Tupel nutzt, diesmal jedoch mit der Konfiguration $m = 4$. Allerdings ist keine bemerkenswerte Verbesserung der Spielstärke festzustellen. Dies zeigt erneut, wie essentiell wichtig die Einführung von je zwei *LUTs* (Abschnitt 4.3) tatsächlich war.

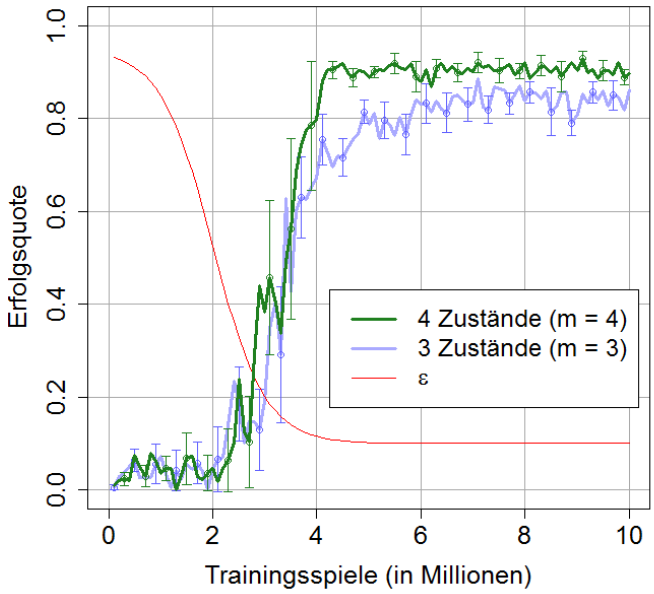


Abbildung 4.14. Training eines *TDL*-Agenten mit Basis-Konfiguration für drei bzw. vier Zuständen je Spielfeldzelle. Für die Codierung $m = 4$ ist die endgültige Spielstärke mit etwa 91% ein wenig besser als für $m = 3$, mit ca. 81% (jeweils das arithmetische Mittel der Erfolgsquote zwischen sechs und zehn Millionen Spielen).

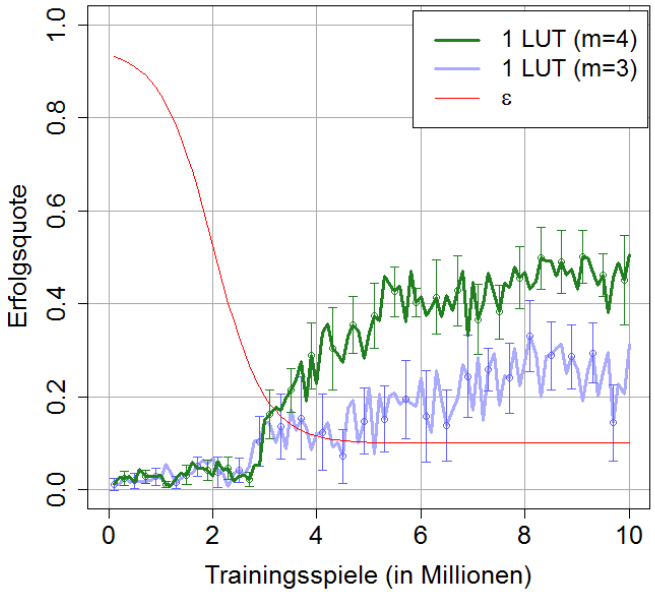


Abbildung 4.15. Vergleich der Konfigurationen $m = 3$ und $m = 4$ für den Fall, dass nur eine *LUT* je N-Tupel zum Einsatz kommt. Auch die Verwendung von $m = 4$ bringt hier keine wesentliche Verbesserung.

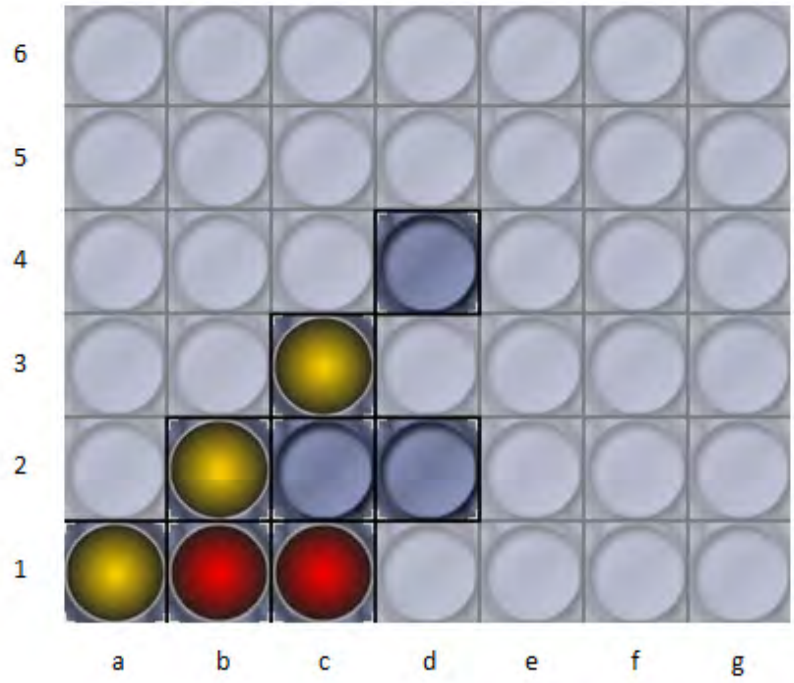


Abbildung 4.16. Eine nicht realisierbare Belegung für ein 8-Tupel. Es ist nicht möglich, dass sich der gelbe Spielstein in der c-Spalte oberhalb eines leeren Feldes befindet. Das zugehörige Gewicht in der entsprechenden *LUT* wird daher während des gesamten Trainings nicht von der Indexierungsfunktion adressiert und bleibt somit auf Null (sofern nicht zufällig initialisiert).

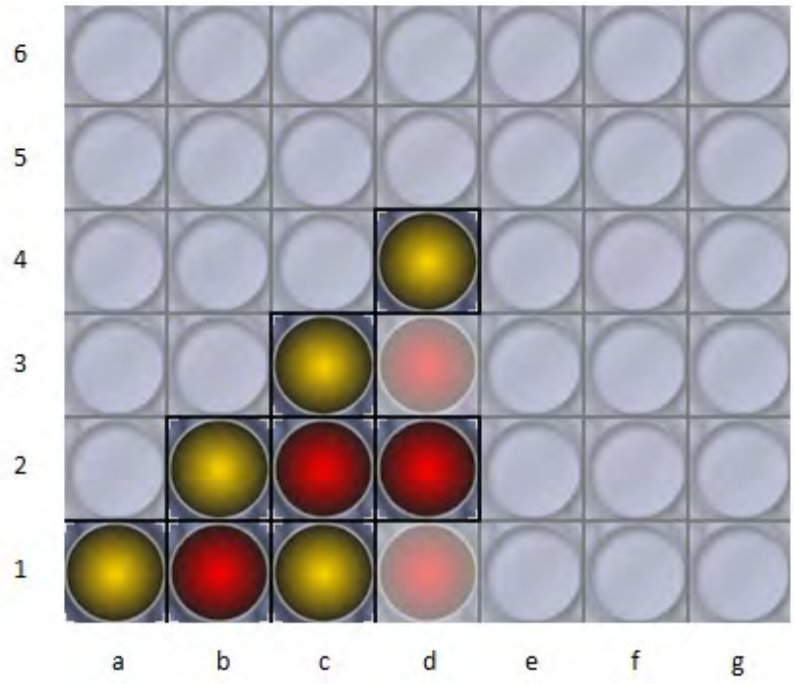


Abbildung 4.17. Ein weiterer, nicht realisierbarer N-Tupel-Zustand. Dies liegt daran, dass terminale Spielstellungen nicht gelernt werden. Das zugehörige Gewicht in der *LUT* bleibt daher während des gesamten Trainings unverändert.

4.4.2 Nicht realisierbare N-Tupel-Zustände

Untersucht man nach dem Training die *LUTs* des *TDL*-Agenten, so wird man beobachten können, dass viele Gewichte bei dem Initialwert Null belassen wurden. Das Training konnte also für keinen Lernschritt die dazu nötige Belegung der N-Tupel erreichen. Bei genauerer Betrachtung sieht man jedoch schnell, dass die meisten dieser N-Tupel-Zustände nicht realisierbar sind. So sind beispielsweise Belegungen mit Gewinnpositionen (vier Steine einer Farbe in einer Reihe) genauso unmöglich, wie Belegungen mit Spielsteinen oberhalb einer leeren Spielfeldzelle (Abbildung 4.16 und Abbildung 4.17). All diese Kombinationen sind in den *LUTs* enthalten, während eines Spiels jedoch nicht erreichbar.

Mithilfe von Formel (4.7) kann eine obere Schranke L_{T} für die realisierbaren Zustände eines N-Tupels $\mathrm{T} = \big((\mathrm{x}_0, \mathrm{y}_0), (\mathrm{x}_1, \mathrm{y}_1), \dots\big)$ errechnet werden[23]. Allerdings ist die tatsächliche Zahl oft noch etwas kleiner, da in der Rechnung terminale Zustände (Gewinnpositionen) unter Umständen noch mit enthalten sind.

$$L_{\mathrm{T}} = \prod_{i=0}^{6} S(i, 0) \tag{4.7}$$

Hierbei können die möglichen Kombinationen für alle Spalten einzeln bestimmt und im Anschluss miteinander multipliziert werden um das Gesamtergebnis zu erhalten. Die rekursiv definierte Funktion $S_m(x, y)$ in Formel (4.8) liefert das Resultat für eine Spalte x, beginnend mit Zeile y (wobei $x, y \in \mathbb{N}_0$):

$$S_m(x, y) = \begin{cases} S(x, y+1)(p(x, y)+1) + p(x, y)\big(z(y) - 2(m-3)p(x, y+1)\big), & y < 6 \\ 1, & y \geq 6 \end{cases} \tag{4.8}$$

Der Wert $m \in \{3,4\}$ gibt an, ob drei oder vier Zustände je Spielfeldzelle angenommen werden. Weiterhin prüft eine Funktion $p(x, y)$ ob das Wertepaar (x, y) einen Abtastpunkt des N-Tupels T darstellt:

$$p(x, y) = \begin{cases} 1, & (x, y) \in \mathrm{T} \\ 0, & sonst \end{cases} \quad mit \quad x, y \in \mathbb{N}_0 \tag{4.9}$$

Außerdem gilt:

$$z(y) = \begin{cases} m-2, & 1 \leq y < 6 \\ 1, & y = 0 \\ 0, & sonst \end{cases} \quad mit \quad y \in \mathbb{N}_0 \tag{4.10}$$

[23] Weitere Details hierzu finden sich im Anhang. Auch die entsprechenden Programme bzw. Maple-Skripte, die diese Formeln umsetzen, sind dort aufgeführt.

Für die Sonderfälle $m = 3$ oder $m = 4$ kann $S_m(x, y)$ etwas umgeschrieben werden, sodass sich die Handhabung ein wenig vereinfacht:

$$S_4(x,y) = \begin{cases} S(x,y+1), & y < 6 \ \wedge \ (x,y) \notin \mathrm{T} \\ 2\big(S(x,y+1) - p(x,y+1)\big) + z(y), & y < 6 \ \wedge \ (x,y) \in \mathrm{T} \\ 1, & y \geq 6 \end{cases} \tag{4.11}$$

$$S_3(x,y) = \begin{cases} S(x,y+1), & y < 6 \ \wedge \ (x,y) \notin \mathrm{T} \\ 2S(x,y+1) + 1, & y < 6 \ \wedge \ (x,y) \in \mathrm{T} \\ 1, & y \geq 6 \end{cases} \tag{4.12}$$

Für $S_3(x, y)$ kann man auch vollständig auf die rekursive Schreibweise verzichten. Es reicht schlicht aus, die Abtastpunkte innerhalb einer Spalte zu zählen:

$$S_3(x,y) = 2^{k+1} - 1 \quad mit \quad k = \sum_{j=5}^{y} p(x,j) \tag{4.13}$$

Die Varianten (4.11) bzw. (4.13) eignen sich insbesondere für handschriftliche Berechnungen, Formel (4.8) ist gut algorithmisch umsetzbar und universell einsetzbar.

In Abbildung 4.18 ist für verschiedene Fälle dargestellt, wie viel Prozent der theoretisch möglichen Zustände tatsächlich realisierbar sind, bzw. wie viel Prozent der Gewichte innerhalb der *LUTs* für das Training nutzbar sind. Jede Zahl wurde anhand von 10^7 N-Tupeln bestimmt[24]. Für die Berechnung der aufgeführten Resultate gilt:

$$R_{\mathrm{T}} = \frac{L_{\mathrm{T}}}{m^N} = \frac{1}{m^N} \prod_{i=0}^{6} S(i,0) \quad mit \quad m \in \{3,4\} \tag{4.14}$$

Generell lässt sich sagen, dass mit der Konfiguration $m = 4$ weniger Belegungen erreichbar sind (relativ) als mit $m = 3$. Darüber hinaus unterscheiden sich die Zahlen auch für die Generierungsvorschriften *"Random Walk"* und *"Random Points"*. Da bei einem *Random Walk* die Wahrscheinlichkeit größer ist, dass mehr Abtastpunkte zusammenhängend innerhalb einer Spalte liegen, ist die Anzahl realisierbarer N-Tupel-Zustände kleiner als für *Random Points.* So sind bei einen *Random Walk* der Länge $N = 8$ im Schnitt weniger als acht Prozent der theoretisch möglichen Belegungen realisierbar. Mehr als 92% der Gewichte innerhalb der *LUT* können daher nicht für das Training genutzt werden.

In Abbildung 4.19 ist dargestellt, wie groß die Unterschiede zwischen den Konfigurationen mit $m = 3$ und $m = 4$ sind. Wie bereits zuvor erwähnt, sind alle Zahlen – aufgrund der terminalen Stellungen – lediglich als obere Schranken zu verstehen und daher in der Praxis noch etwas kleiner.

[24] Das zugehörige Java-Programm (*miscellaneous.CountRealizableStates*) kann im Anhang nachgeschlagen werden.

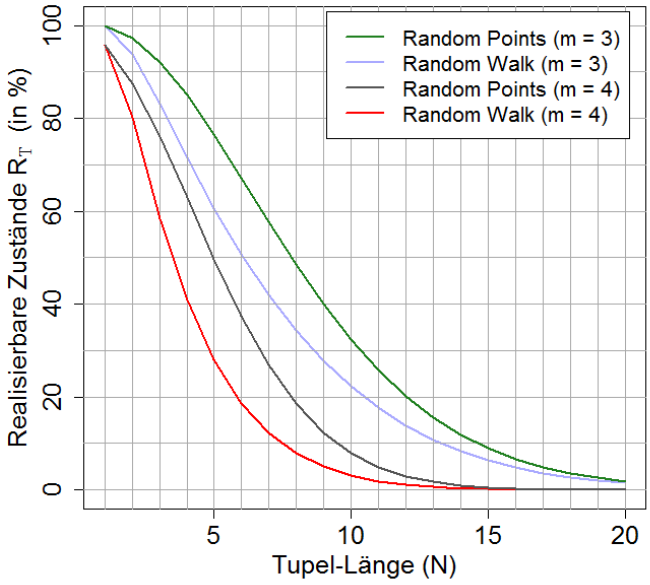


Abbildung 4.18. Tatsächlich realisierbare N-Tupel-Zustände (in Prozent) für Tupel-Längen zwischen 1 und 20, wobei Längen kleiner als 3 bzw. größer als 12 in der Praxis vermutlich nicht realistisch sind. Die Generierungsmethoden *"Random Walk"* und *"Random Points"* werden getrennt betrachtet, da sich für diese unterschiedliche Zahlen ergeben. Jeder Wert ist anhand von 10^7 zufällig erzeugten Tupeln errechnet worden.

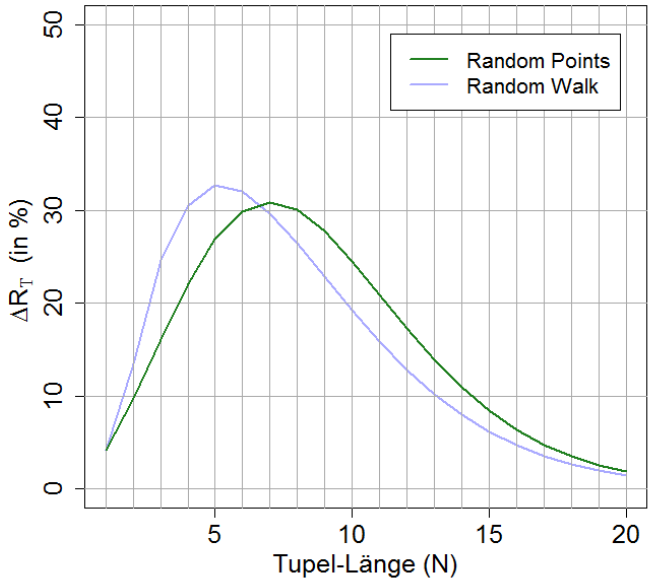


Abbildung 4.19. Differenz (der relativen Werte) zwischen den realisierbaren Zuständen für $m = 3$ und den realisierbaren Zuständen für $m = 4$, jeweils für die Generierungsvorschriften *"Random Points"* und *"Random Walk"*. Beispielsweise sind für ein *Random-Walk*-Tupel der Länge sieben etwa 30% mehr N-Tupel-Zustände möglich, daher auch 30% der *LUT*-Einträge mehr nutzbar, wenn $m = 3$ anstelle von $m = 4$ gewählt wird.

Für ein N-Tupel-System bestehend aus 70 8-Tupeln sind etwa 70 MB an Arbeitsspeicher nötig, um alle *LUTs* zu realisieren (bei zwei *LUTs* je N-Tupel und acht Bytes je *LUT*-Eintrag). Von diesen 70 MB stehen im Schnitt ca. 18% (13 MB) dem Training zur Verfügung, der verbleibende Speicher wird nicht benötigt. Initialisiert man zu Beginn des Trainings alle Gewichte mit Null, behalten ca. 72% der Einträge diesen Null-Wert auch während des Trainings bei. Dies hat den Vorteil, dass beim Abspeichern des trainierten *TDL*-Agenten auf die Festplatte – bei Verwendung eines geeigneten Kompressionsverfahrens – ein sehr hoher Kompressionsgrad erreicht wird. Inwieweit sich die Struktur der *LUTs* im Arbeitsspeicher optimieren lassen, um den Speicher effizienter nutzen zu können, ist noch zu prüfen. Da der Zugriff auf die *LUTs* strengen Effizienzbedingungen unterworfen ist, scheint dies nicht ohne Weiteres möglich zu sein.

4.5 Unterschiedliche Typen von N-Tupel-Sets im Vergleich

Wie zuvor bereits erwähnt, können einige verschiedene Ansätze zur Erzeugung der N-Tupel in Betracht gezogen werden. So kann der Programmbenutzer alle N-Tupel eines Systems selbst vorgeben oder zufällig generieren lassen. Für das automatische Erzeugen stehen die Generierungsverfahren *"Random Walk"* (wie es auch Lucas in [6] verwendet) und *"Random Points"* zur Verfügung. Die Tupel werden hierbei vollständig zufällig erzeugt, es fließt keinerlei Wissen irgendeiner Form in den Generierungsprozess mit ein. Neben dem Generierungsverfahren spielt auch die Anzahl oder die Länge der N-Tupel eine entscheidende Rolle für den Trainingserfolg des *TDL*-Agenten.

4.5.1 Die Generierungsvorschriften "Random Walk" und "Random Points"

Beim Random Walk beginnt der Algorithmus mit einem beliebigen Startpunkt auf dem Spielfeld. Im Anschluss wird eines der max. acht möglichen Nachbarzellen als Abtastpunkt reserviert. Dies wird solange fortgesetzt, bis die gewünschte Zahl an Abtastpunkten erreicht ist.

Im Gegensatz zum Random Walk – bei dem alle Abtastpunkte eine zusammenhängende Kette bilden – wählt der "Random Points"-Algorithmus die Abtastpunkte vollkommen zufällig aus, sodass diese für gewöhnlich über das gesamte Spielfeld zerstreut sind. Daher kann (zumindest für *Vier Gewinnt*) in der Regel kein direkter Zusammenhang zwischen den einzelnen Abtastpunkten hergestellt werden, da diese zu weit auseinanderliegen; mit anderen Worten: Verschiedene Belegungen für solche N-Tupel lassen keinerlei Rückschlüsse bezüglich der Stellungseinschätzung zu.

Dies ist bei einem Random Walk anders, da hier alle Abtastpunkte direkt zusammenhängen. Dadurch ist es viel wahrscheinlicher, dass z.B. potentielle Vierer-Ketten abgedeckt werden, die für den Spielausgang eine entscheidende Rolle spielen.

in Abbildung 4.20 sind die Ergebnisse des Trainings für beide Generierungsverfahren dargestellt. In beiden Fällen wurden zehn Trainingsläufe mit jeweils *neuen* N-Tupel-Sets ausgeführt; wobei den Systemen 70 (Random Walk) bzw. 100 (Random Points) Tupel der Länge $N = 8$ zugestanden wurden. Trotz dieser größeren Anzahl an N-Tupeln, konnte der *TDL*-Agent mit den Random Points keine zufriedenstellenden Resultate erzielen. Daher spielen diese in den weiteren Betrachtungen keine Rolle.

Während der Arbeiten an den Generierungsvorschriften kam die Überlegung auf, dass möglicherweise bessere Ergebnisse erzielt würden, wenn der "Random Walk"-Algorithmus mit einer höheren Wahrscheinlichkeit in Richtung der zentralen Spielfeldzellen wandert, da deren Belegung eine für den Spielausgang wesentliche Bedeutung zukommt. Da der Algorithmus diese Zellen ohnehin öfter besucht (Abbildung 4.21), ohne dass hierauf gezielt Einfluss genommen wird, wurde die Idee allerdings wieder verworfen.

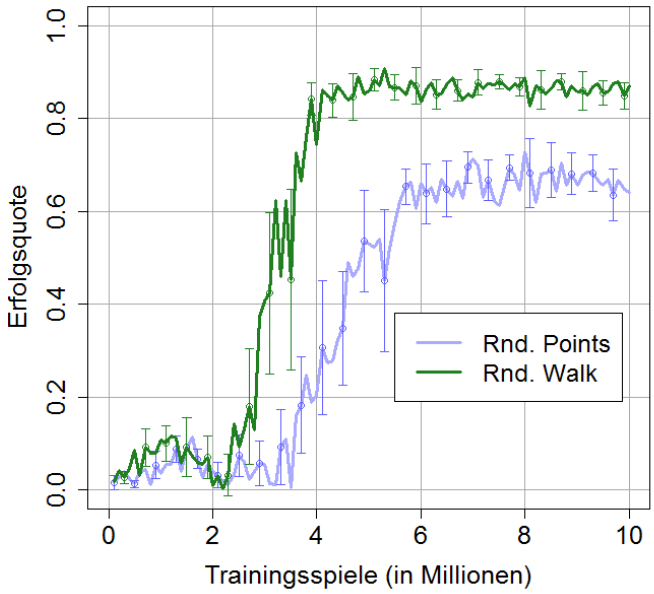


Abbildung 4.20. Die Generierungsmethoden "Random Walk" und "Random Points" im Vergleich zur Basiskonfiguration mit 70 8-Tupeln. Es werden jeweils 100 "Random Points"- bzw. 70 "Random Walk"-Tupel eingesetzt. In allen 10 Trainingsdurchläufen erzeugt das Programm *neue* N-Tupel-Sets. Ansonsten werden die Parameter der Basiskonfiguration beibehalten.

6	8	15	19	21	19	15	8
5	15	25	32	36	32	25	15
4	17	30	39	43	39	30	17
3	17	30	39	43	39	30	17
2	14	25	32	35	32	25	15
1	8	15	19	21	19	15	8
	a	b	c	d	e	f	g

Abbildung 4.21. Häufigkeit (in ‰), in der die Spielfeldzellen vom "Random Walk"-Algorithmus besucht werden. Die Werte wurden anhand von 10^6 Tupeln der Länge *acht* ermittelt. Allerdings sind für andere Tupel-Längen hiervon abweichende Ergebnisse zu erwarten.

4.5.2 Einfluss der Tupel-Anzahl auf das Trainingsergebnis

Untersucht man N-Tupel-Systeme, die sich nur hinsichtlich der Anzahl ihrer N-Tupel unterscheiden, so sollte ein System mit mehr N-Tupeln im Schnitt ein besseres Trainingsresultat liefern, als ein System mit nur wenigen N-Tupeln, da ersteres erwartungsgemäß einen größeren Feature-Raum ermöglicht.

Daher ist es verwunderlich, dass die Systeme bestehend aus 120 8-Tupeln in Abbildung 4.22 schlechtere Ergebnisse erzielen, als die mit nur 70 8-Tupeln (aus Abschnitt 4.5.1 übernommen). In beiden Fällen wurden für jeden neuen Trainingslauf völlig zufällige N-Tupel-Sets erzeugt und auch die übrigen Parameter sind beibehalten worden.

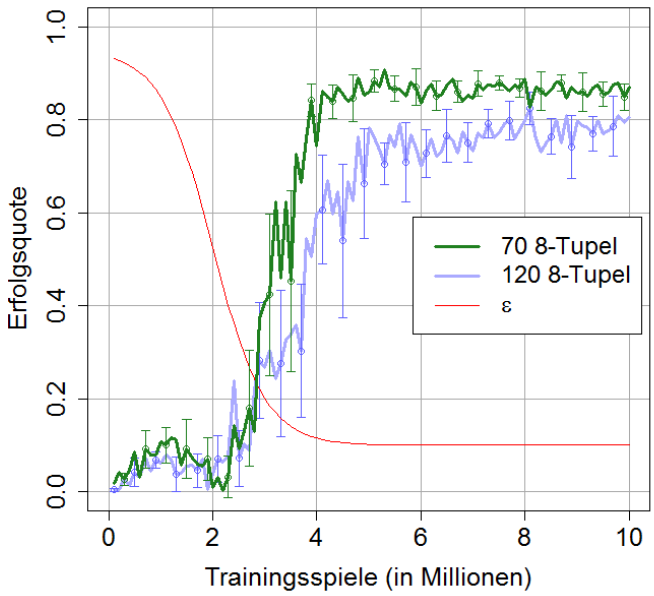


Abbildung 4.22. Unterschiedliche N-Tupel-Systeme mit 70 bzw. 120 Tupel der Länge acht im Vergleich. Wider Erwarten liegt das Trainingsresultat für nur 70 Tupel höher als für 120, obwohl alle N-Tupel nach derselben Vorschrift generiert wurden und auch die übrigen Parameter gleich sind.

Da der einzige Punkt, in dem sich die beiden Systeme unterscheiden, in der Anzahl der N-Tupel besteht, liegt die Vermutung nahe, dass ein oder mehrere Parameter der Basis-Konfiguration sich negativ auf das Training auswirken, sobald eine größere Zahl an N-Tupel zum Einsatz kommt.

Nach einigen Untersuchungen konnte festgestellt werden, dass die Ausgabe der Spielfunktion $V(s_t)$ sich sehr häufig bereits im Grenzbereich bewegte (für 120 N-Tupel), die durch die Aktivierungsfunktion vorgegeben ist. Die ursprüngliche Ausgabe des N-Tupel-Systems befindet sich daher häufig im Bereich der Sättigung des *Tangens-Hyperbolicus*. Da die Spielfunktion nun über $2 \cdot 120$ (bei Ausnutzung von Spiegelsymmetrien) Gewichte summieren muss, statt ursprünglich $2 \cdot 70$, kann hieraus möglicherweise ein Problem entstehen.

Angenommen der Durchschnittswert eines einzelnen Gewichtes läge bei w: Für 70 N-Tupel wäre die Ausgabe der Spielfunktion $V(\mathrm{s_t}) = 140w$, für 120 Tupel jedoch $V(\mathrm{s_t}) = 240w$, also etwa das Doppelte. Übergibt man diese Werte der Aktivierungsfunktion $\bar{V}(\mathrm{s_t}) = \tanh\big(V(\mathrm{s_t})\big)$, ist man möglicherweise zu weit in den Sättigungsbereich der Funktion hineingekommen.

Daher wurde entschieden, die Ausgabe der Spielfunktion testweise zu skalieren, bevor diese an die Sigmoid-Funktion weitergereicht wird. Es ergibt sich folgende Form:

$$\bar{V}(\mathrm{s_t}) = \tanh\big(k \cdot V(\mathrm{s_t})\big) \tag{4.15}$$

Hierdurch ergibt sich auch eine Änderung des Gradienten im TD-Learning Algorithmus (vgl. Formel (2.20)):

$$\begin{aligned} \nabla_{\mathrm{w}} f\big(w;\, g(s_t)\big) &= k \cdot \left[1 - \tanh^2\left(\sum_k w_k \cdot g_k(\mathrm{s_t})\right)\right] \cdot g(\mathrm{s_t}) \\ &= k \cdot \left[1 - f\big(w;\, g(\mathrm{s_t})\big)^2\right] \cdot g(\mathrm{s_t}) \end{aligned} \tag{4.16}$$

Der Faktor k wird für dieses Beispiel mit dem Wert

$$k = \frac{70}{120} \approx 0{,}58 \tag{4.17}$$

belegt.

Die neuen Ergebnisse für die angepasste Spielfunktion mit dem zusätzlichen Skalierungsfaktor k sind in Abbildung 4.23 aufgeführt. Tatsächlich wird jetzt die Spielstärke erreicht, die zu Beginn erwartet wurde.

Abbildung 4.24 zeigt die Erfolgsquote des *TDL*-Agenten für Tupel der Länge $N = 8$ in Abhängigkeit von deren Anzahl. Auch hier kommt der neue Faktor k zum Einsatz. Es wurden je 10 Trainingsläufe durchgeführt, wobei in jedem Lauf ein neues N-Tupel-System generiert wurde. Da der *TDL*-Agent in allen Trainingsläufen nach spätestens 7 Mio. Spielen seine endgültige Spielstärke erreicht, wird das arithmetische Mittel und die Standardabweichung über alle Werte zwischen den 7-10 Mio. Spielen errechnet, für jeden Punkt werden daher alle $10 \cdot 30 = 300$ Werte herangezogen.

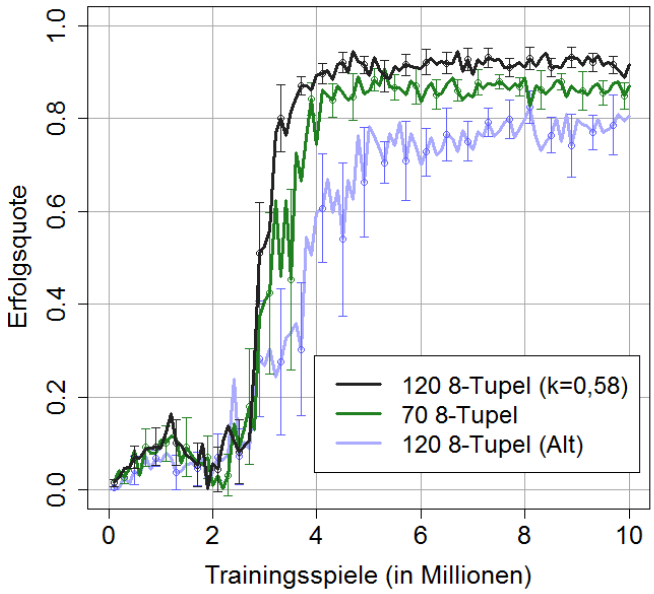


Abbildung 4.23. Neue Resultate für das N-Tupel-System mit 120 8-Tupeln, bei Einführung des Skalierungsfaktors k. Wie zu Beginn erwartet, kann der *TDL*-Agent nun eine höhere Spielstärke erreichen, als mit einem System bestehend aus 70 8-Tupeln.

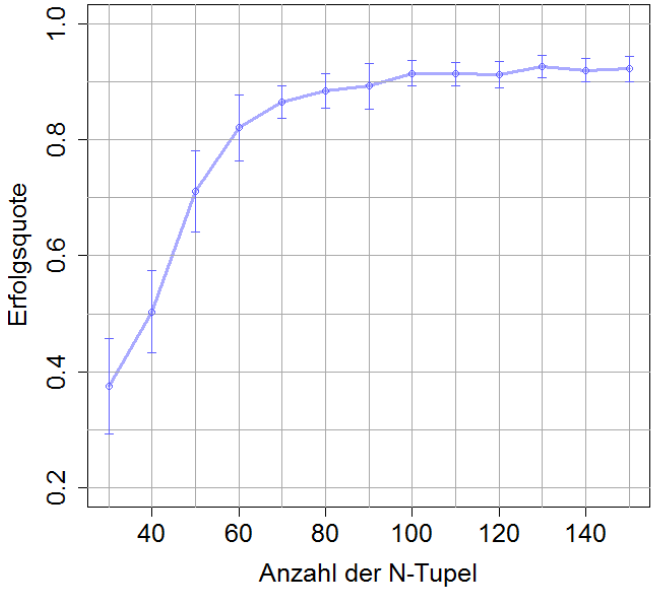


Abbildung 4.24. Resultierende Erfolgsquote des *TDL*-Agenten für Tupel der Länge $N = 8$, in Abhängigkeit von der Anzahl der Tupel. Zur Bestimmung der Ergebnisse wurden je 10 Trainingsläufe durchgeführt und das arithmetische Mittel für die Werte zwischen 7-10 Mio. Trainingsspielen errechnet (es wurden daher 300 Werte für jeden Punkt herangezogen).

4.5.3 Reproduzierbarkeit der Trainingsresultate

Ein sehr wichtiges Kriterium zur Beurteilung der Qualität eines Systems ist die Fähigkeit, für jedes Training reproduzierbare Ergebnisse zu erzeugen. Ein N-Tupel-System, bei dem durchgehend stark schwankende Trainingsleistungen zustande kommen, ist im Grunde unbrauchbar.

Für das N-Tupel-System mit Basis-Konfiguration konnte in keinem Trainingsdurchlauf beobachtet werden, dass sich gravierende Unterschiede in der resultierenden Spielstärke des *TDL*-Agenten ergaben. Lediglich zu Beginn des Trainings (bis zu etwa 4 Mio. Trainingsspielen) sind größere Schwankungen vorhanden (siehe Abbildung 4.25). Auch für fast alle anderen N-Tupel-Systeme ließen sich die Ergebnisse stets gut reproduzieren.

In Abbildung 4.26 ist die Erfolgsquote für ein N-Tupel-System (bestehend aus 70 zufällig generierten 8-Tupeln) dargestellt, das sehr wechselhafte Trainingsverläufe erzeugt. Einige Resultate sind – mit einer Erfolgsquote von etwa 90% – recht zufriedenstellend. Andere kommen jedoch selten über die 10%-Marke hinaus.

Für dieses N-Tupel-System sind daher keine reproduzierbaren Ergebnisse möglich. Allerdings wurde dieses Phänomen sehr selten beobachtet. Im Regelfall sind die Ergebnisse für ein spezifisches N-Tupel-System sehr beständig (vorausgesetzt, die übrigen Parameter bleiben unverändert).

In Abschnitt 4.5.1 und 4.5.2 wurden bereits *TDL*-Agenten diskutiert, die in jedem Trainingslauf ein neues N-Tupel-System erzeugen. Auch hier lassen sich in praktisch allen Fällen ähnliche Verläufe beobachten, obwohl die Standardabweichung etwas größer ist, als für ein einzelnes N-Tupel-System.

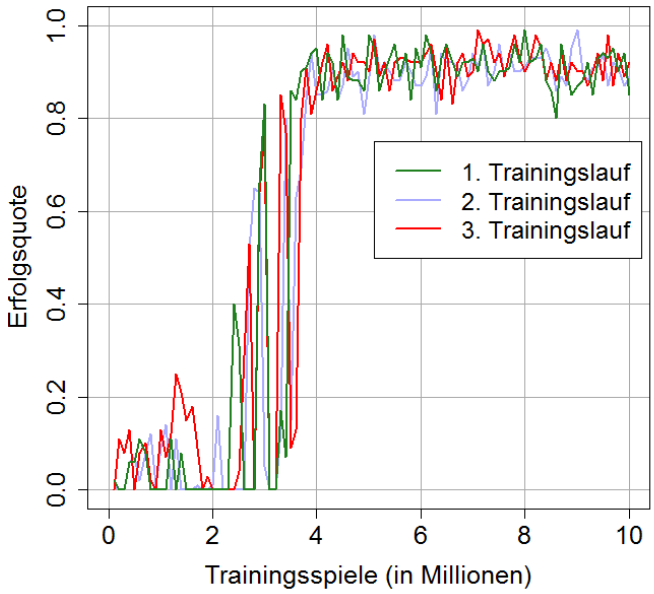


Abbildung 4.25. Drei der zehn Trainingsdurchläufe, die für die Basiskonfiguration durchgeführt wurden. Im Bereich zwischen 2-4 Mio. Trainingsspielen sind größere Abweichungen zu beobachten, ansonsten sind die Kurvenverläufe ähnlich.

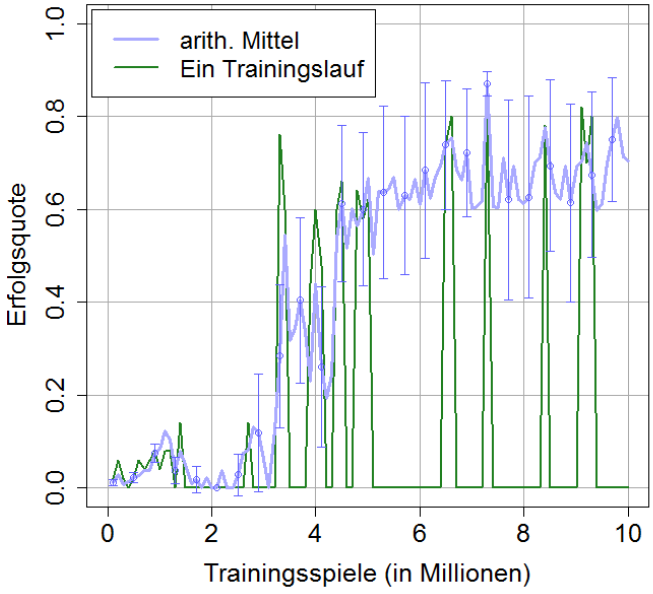


Abbildung 4.26. Sehr wechselhafte Ergebnisse für ein N-Tupel-System – bestehend aus 70 8-Tupeln. Dies wird zum einen durch die Standardabweichung angedeutet, weiterhin ist einer von zehn Trainingsverläufen abgebildet, bei dem die Erfolgsquote über weite Strecken bei 0% liegt.

4.6 Weitere Untersuchungen

4.6.1 Feinjustierung der Trainingsparameter

Die Basiskonfiguration (Abschnitt 4.1.5) des *TDL*-Agenten liefert zwar schon vergleichsweise gute Trainingsresultate. Allerdings ist die Anzahl der Trainingsspiele – nach der die endgültige Spielstärke erreicht wird – sehr hoch. Wie sich nach einigen Untersuchungen herausstellte, ist der Initialwert der Lernschrittweite α_{Start} zu groß gewählt worden, sodass erste sichtbare Erfolge erst nach etwa 4 Mio. Trainingsspielen zustande kommen. Auch die hohe Explorationsrate ε zu Beginn trägt hierzu bei. Daher wurde die Lernschrittweite auf die Werte $\alpha_{Start} = 0{,}004$ und $\alpha_{Final} = 0{,}002$ geändert und der Initialwert der Explorationsrate auf $\varepsilon_{Start} = 0{,}6$ reduziert, bei gleichzeitiger Verschiebung des Wendepunktes der Sigmoid-Funktion an die Stelle $n_{wp} = 10^6$ Trainingsspiele. Die Ergebnisse der durchgeführten Anpassungen sind in Abbildung 4.27 dargestellt.

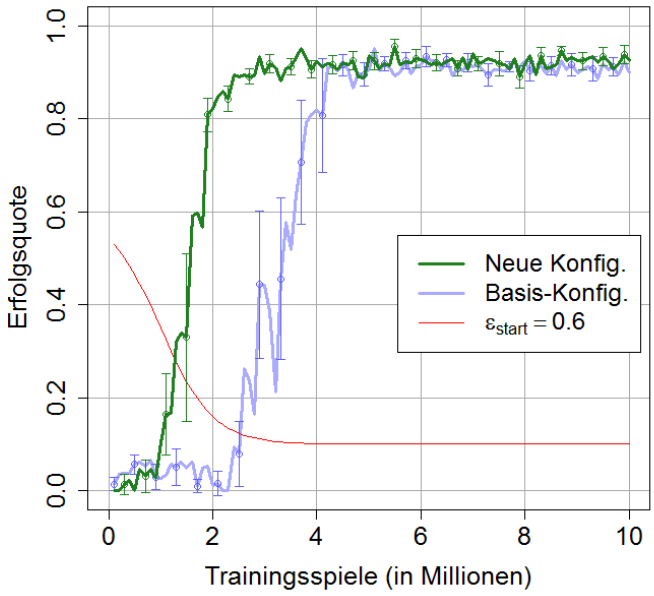


Abbildung 4.27. Verbesserung der Basis-Konfiguration. Lernschrittweite α sowie Explorationsrate ε wurden angepasst, um eine höhere Lerngeschwindigkeit zu erreichen. Die 80%-Marke wird nun bereits nach etwa 2 Mio. Spielen überschritten, anstatt nach 4 Mio. Spielen.

4.6.2 Bedeutung der Spielfeldsymmetrien für das Training

Wie bereits in Abschnitt 2.3.3 erläutert wurde, kann man die Lerngeschwindigkeit des *TDL*-Agenten verbessern, wenn das Training Spielfeldsymmetrien ausnutzt, die einzelnen Lernschritte daher gleichzeitig für alle symmetrischen Spielstellungen durchführt. Insbesondere für Spiele wie *Tic Tac Toe* und *Othello* ist dies vorteilhaft, da bei diesen Spielen – aufgrund von Rotations- und Spiegelsymmetrien – bis zu acht Stellungen äquivalent sind. *Vier Gewinnt* lässt nur Spiegelungen an der mittleren Spalte zu, hier können daher nur zwei Stellungen gleichzeitig erlernt werden.

Verzichtet man auf die Verwendung von Symmetrien – lässt den Agenten also jeweils nur eine Position erlernen – dauert es deutlich länger, bis eine gewisse Anzahl an Knoten des Spielbaumes besucht wurden. Dies kann eine größere Anzahl der Trainingsspielen nötig machen, um eine vergleichbare Spielstärke zu erreichen.

In Abbildung 4.28 ist das Ergebnis dargestellt, das sich ergibt, wenn auf die Ausnutzung der Spielfeldsymmetrien für das Spiel *Vier Gewinnt* verzichtet wird (Beibehaltung der übrigen Parameter). Die Erfolgsquote erreicht hierbei lediglich einen Wert von etwa 70%.

Auch mit verschiedenen anderen Konfigurationen (unterschiedliche Lernschrittweite und Explorationsrate, Verdopplung der Trainingsspiele) kann das Resultat nicht wesentlich verbessert werden.

Die Vermutung, dass durch die Verschiebung des Arbeitsbereiches innerhalb der Aktivierungsfunktion (es werden schließlich nur noch halb so viele Gewichte aufsummiert) und der daraus resultierenden Wahl von $k = 2$ (siehe Abschnitt 4.5.2), führte ebenfalls nicht zu dem gewünschten Resultat.

Letztendlich ist daher davon auszugehen, dass viele N-Tupel-Zustände nicht oft genug erreicht werden, um ein ausreichendes Anlernen der Gewichte zu ermöglichen. Bei der Verwendung von Symmetrien werden jeweils zwei Varianten des Spiels gelernt; zu Beginn des Trainings fast ausschließlich terminale Stellungen, später vorwiegend non-terminale Stellungen. Nachdem die Explorationsrate einen gewissen Wert unterschritten hat, verhält sich der *TDL*-Agent nahezu deterministisch und die Lerngeschwindigkeit nimmt stark zu (vorausgesetzt es wurden genügend terminale Stellungen gelernt). Verzichtet man in dieser Phase des Trainings auf die Verwendung von Symmetrien, werden vermutlich viele Spielstellungen nicht gelernt und der *TDL*-Agent kann für diese später keine vernünftige Einschätzung mehr vornehmen. Dies macht sich letzten Endes in der niedrigeren Spielstärke des Agenten bemerkbar. Daher scheint es nicht so ohne weiteres möglich zu sein, den Verzicht auf Symmetrien durch eine größere Zahl an Trainingsspielen zu kompensieren.

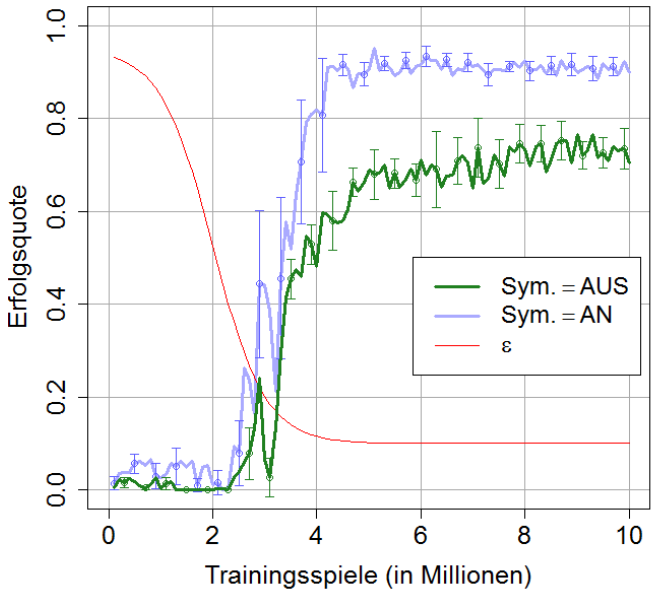


Abbildung 4.28. Verzicht auf die Verwendung von Symmetrien für das Training. Der *TDL*-Agent erreicht im Schnitt eine endgültige Erfolgsquote von etwa 70%, daher ca. 20% niedriger als im Normalfall.

4.6.3 Kombination der Alpha-Beta-Suche mit einem N-Tupel-System

Für alle zuvor dargestellten Ergebnisse wurde keine Baumsuche in irgendeiner Form verwendet; zur Bestimmung des bestmöglichen Folgezuges expandierte der *TDL*-Agent lediglich den aktuellen Spielzustand und wählte im Anschluss den – aus seiner Sicht – bestmöglichen Folgezustand aus.

Da der *TDL*-Agent im Regelfall nicht für jede Stellung eine korrekte Einschätzung vornehmen kann (die Spielfunktion wird lediglich durch das N-Tupel-System approximiert), kommt es durchaus vor, dass ein Spiel nur aufgrund einer Fehleinschätzung verloren geht. So konnte häufig beobachtet werden, dass der *TDL*-Agent – nach Korrektur eines Einzelfehlers – perfekt weiterspielte.

Die einfachste Möglichkeit, um die Wahrscheinlichkeit für das Auftreten gelegentlicher Fehleinschätzungen zu verkleinern, besteht darin, die Suchtiefe des Agenten zu erhöhen. Dazu ist es nötig, die Spielfunktion des Agenten mit einem Baumsuchverfahren zu kombinieren. Erreicht die Baumsuche ihre maximale Suchtiefe – die sogenannten Blattknoten – nimmt die Spielfunktion eine Bewertung des Blattknotens vor und gibt diesen Wert zurück. Werden zuvor schon terminale Stellungen erreicht, ist keine weitere Evaluierung nötig, da bereits ein exakter Wert vorliegt.

Die Suchtiefenerweiterung kann bereits während des Trainings erfolgen. Da sich jedoch hierdurch die Trainingszeit deutlich erhöht, ist dies nur für eine Vergrößerung um einen Halbzug durchgeführt worden (Abbildung 4.29). Eine wesentliche Steigerung der Spielstärke ist jedoch nicht festzustellen.

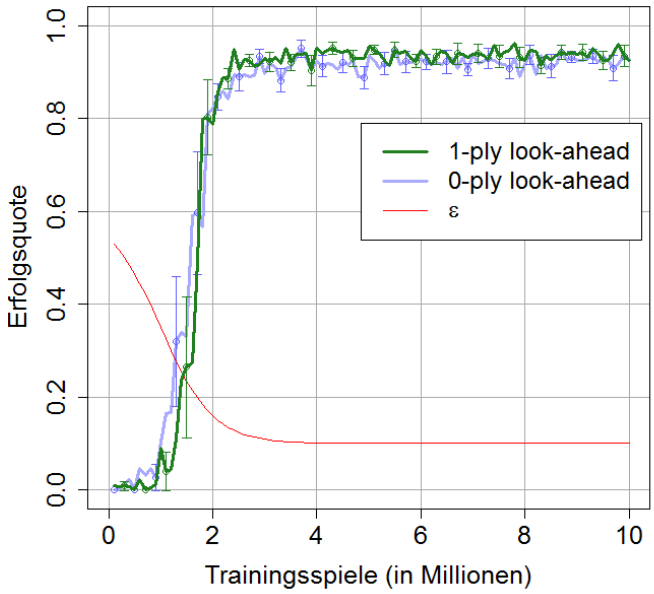


Abbildung 4.29. Erhöhung der Suchtiefe während des Trainings um einen Halbzug(für die verbesserte Parameterkonfiguration aus Abschnitt 4.6.1). Die resultierende Erfolgsquote ist minimal – um etwa 1% – höher.

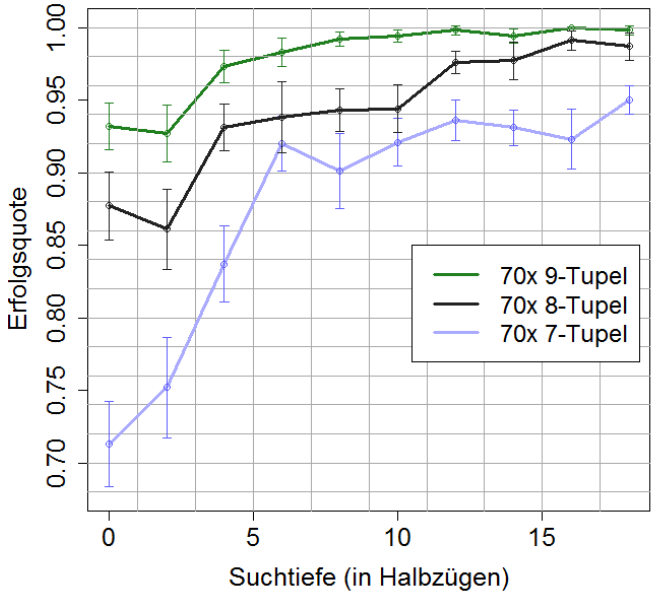


Abbildung 4.30. Kombination eines Alpha-Beta-Agenten mit verschiedenen (zufällig erzeugten) N-Tupel-Systemen. Dargestellt ist die Erfolgsquote in Abhängigkeit der Suchtiefe (0-18 Halbzüge, Schritte von zwei Halbzügen). Für eine Suchtiefe von Null wird unmittelbar das N-Tupel-System befragt.

Eine weitere Möglichkeit besteht darin, ein zuvor trainiertes N-Tupel-System (keine Vergrößerung der Suchtiefe während des Trainings) als Evaluierung an den Blattknoten eines Alpha-Beta-Agenten einzusetzen. Dies ist für verschiedene (70 zufällig erzeugte Random-Walk-Tupel) N-Tupel-Systeme in Abbildung 4.30 durchgeführt worden, wobei jeder Wert anhand von 500 *vollständigen* Spielen gegen den Minimax-Agenten ermittelt wurde. So kann für das betrachtete 70x7-Tupel-System eine Steigerung der Erfolgsquote von etwa 72% auf 92% beobachtet werden, wenn man die Suchtiefe von null auf sechs erhöht. Das System bestehend aus 70 9-Tupeln kann mit acht Halbzügen Suchtiefenerweiterung eine Erfolgsquote von knapp 99% erzielen.

Die Eröffnungsdatenbanken sind für alle Untersuchungen deaktiviert worden.

In Tabelle 4.2 ist die durchschnittliche Evaluierungszeit (Zeit für 50 Testspiele gegen einen Minimax-Agenten) der verschiedenen N-Tupel-Systeme dargestellt. Dies lässt zwar keine exakte Aussage auf die durchschnittliche Laufzeit der Suche (in Kombination mit dem N-Tupel-System) zu. Dennoch werden die Größenordnungen in etwa sichtbar. So dauert *ein Spiel* zwischen dem perfekt spielenden Minimax-Agenten und dem System mit 70 Tupeln der Länge $N = 7$ ca. 1 Sekunde für eine Suchtiefe von Null, bzw. knapp 18 Sekunden ($= 895s/50$) für eine Suchtiefe von 18 Halbzügen.

Tabelle 4.2. Durchschnittliche Evaluierungszeit (in s) für die verschiedenen N-Tupel-Systeme in Abhängigkeit der Suchtiefe. Je Evaluierungsvorgang wurden 50 Spiele gegen einen perfekt spielenden Minimax-Agenten durchgeführt.

Suchtiefe (in Halbzügen)	Evaluierungszeit (in Sek.)		
	70x 7-Tupel	70x 8-Tupel	70x 9-Tupel
0	50	48	41
2	46	44	42
4	41	44	39
6	42	37	38
8	44	38	39
10	46	49	47
12	66	71	68
14	116	152	165
16	401	494	535
18	895	1179	1267

4.6.4 Zugsortierung der Alpha-Beta-Suche mithilfe eines N-Tupel-Systems

Werden N-Tupel-Systeme zur Approximierung der Spielfunktion eingesetzt, sind in aller Regel keine exakten Stellungsbewertungen möglich. Ein gutes System kann aber in vielen Fällen zumindest eine grobe Einschätzung liefern, sodass die Qualität der möglichen Folgezustände in eine richtige Reihenfolge gebracht wird. Dieser Sachverhalt kann man zur Optimierung der Zugsortierung ausnutzen: Die Ausführungsreihenfolge der Züge in einem Knoten ist für die Korrektheit der Alpha-Beta Suche irrelevant,

eine schlechte Zugsortierung wirkt sich lediglich auf die Laufzeit der Suche negativ aus. In der Hoffnung, dass das eingesetzte N-Tupel-System die Folgezustände eines Knoten bezüglich ihrer Qualität in vielen Fällen richtig einschätzen kann, ist ein Einsatz zur Vorsortierung der Züge eine interessante Möglichkeit.

Testweise wurde ein N-Tupel-System bestehend aus 30 7-Tupeln trainiert, das die einfache Zugsortierung in wurzelnahen Knoten übernahm, allerdings nur für die Knoten des anziehenden Spielers bis zum zehnten Halbzug; in den Knoten des Nachziehenden brachte die Zugsortierung durch das N-Tupel-System keine Laufzeitverbesserung. Möglicherweise führen hier noch weitere Tests mit anderen Systemen zum Erfolg.

Die Rechenzeit zur Analyse des leeren Spielfeldes konnte mithilfe dieser Maßnahme auf *weniger als einem Drittel* der ursprünglichen Zeit verkürzt werden und auch für viele andere Stellungen konnte man zum Teil deutliche Verbesserungen beobachten. Insgesamt konnte für alle Stellungen mit null, einem oder zwei Steinen eine Verkürzung der Rechenzeit um *26%* festgestellt werden (die Alpha-Beta-Suche mit der N-Tupel-basierten Zugsortierung war für *78%* der betrachteten Stellungen schneller). Alle Eröffnungsdatenbanken wurden selbstverständlich für die Laufzeitmessungen deaktiviert.

4.6.5 Evaluierung des *TDL*-Agenten mit einem weiteren Programm

Da bisher alle Evaluierungs-Vorgänge mithilfe des ein und desselben Minimax-Agenten vorgenommen wurden, soll zum Ende diesen Kapitels noch ein letzter Vergleich mit einem weiteren Programm beschrieben werden. Als Vergleichsmöglichkeit erscheint hier das *Vier-Gewinnt*-Programm *Mustrum* [27] als besonders geeignet, auch weil das Programm als Nachziehender abwechslungsreiche Spiele ermöglicht. Laut seinem Autor *Lars Bremer* ist *Mustrum* eines der wenigen Programme weltweit, das *Vier Gewinnt* perfekt beherrscht.

Da keine automatisierten Tests möglich waren, wurde lediglich ein bereits trainierter *TDL*-Agent – basierend auf 70 8-Tupeln, ohne einer vorausschauenden Suche – mit dem Programm *Mustrum* (Version 2.0.3) verglichen. Insgesamt wurden 20 Testspiele durchgeführt, von denen der *TDL*-Agent – als Anziehender – *18 Spiele gewann* und ein Spiel verlor; ein Spiel endete Unentschieden. Dies würde umgerechnet einer Erfolgsquote von etwa *92%* entsprechen (mit dem Standard-Minimax-Agenten aus dieser Arbeit wurde eine Erfolgsquote von *88%* bestimmt).

Bei Vertauschung der Rollen beider Spieler (*Mustrum*[25] als Anziehender und *TDL*-Agent als Nachziehender) konnte ein interessantes Ergebnis beobachtet werden: *Mustrum* konnte nicht in jedem Fall die Eröffnung fehlerfrei überstehen, sodass der *TDL*-Agent diese Fehler bestrafen und die entsprechenden Spiele gewinnen konnte. Insgesamt verlor *Mustrum* 8 der 20 Spiele. 12 Spiele konnte *Mustrum* für sich entscheiden; im Schnitt nach etwa 34 Halbzügen. Da die Fehler – bei den verloren gegangen Spielen – allesamt in der Eröffnungsphase passierten, ist es wahrscheinlich, dass ein kleineres Problem mit dem "großen Eröffnungsbuch" von *Mustrum* vorliegt.

Die Spielverläufe und der *TDL*-Agent können im Anhang nachgeschlagen werden.

[25] Das Programm verwendet hierfür das "große Eröffnungsbuch" anstelle des "tiefen Eröffnungsbuches", da dies laut Help-File des Autors für mehr Abwechslung im Spiel führt, wenn *Mustrum* den Anziehenden spielt. An der Spielstärke (Suchtiefe) des Programmes wurden keine Veränderungen durchgeführt.

5 Zusammenfassung und Ausblick

In dieser Arbeit ist die erstmalige Anwendung einer *RL*-Trainingsumgebung mit N-Tupel-Systemen zur Funktionsapproximierung auf das Spiel *Vier Gewinnt* untersucht worden.

Bei ersten Experimenten mit dem trivialen Spiel *Tic Tac Toe* und mit verschiedenen Vier-Gewinnt-Eröffnungsphasen (Abschnitt 4.2) konnten schnell sehr gute Trainingsresultate beobachtet werden, sodass bereits relativ früh dazu übergegangen wurde, dass vollständige *Vier-Gewinnt*-Spiel zu untersuchen. Die hierzu durchgeführten Untersuchungen waren jedoch zunächst ernüchternd. Der *TDL*-Agent konnte als Anziehender nur sehr wenige Spiele gegen einen perfekt spielenden Minimax-Agenten für sich entscheiden. Erst die Einführung von einer Look-up-Tabelle (*LUT*) je Spieler – insgesamt also zwei *LUTs* je N-Tupel – konnte die Spielstärke des *TDL*-Agenten erheblich steigern (Abschnitt 4.3).

Anhand der – in Kapitel 3 – formulierten Forschungsfragen, lassen sich die erzielten Ergebnisse etwas genauer bewerten:

1. *RL* in Kombination mit N-Tupel-Systemen konnte erfolgreich auf das Spiel *Vier Gewinnt* (über den kompletten Zustandsraum) angewendet werden. Bei geeigneter Konfiguration (Punkt 4) ist der *TDL*-Agent in der Lage, als anziehender Spieler, ohne Verwendung eines Baumsuch-Verfahrens, im Schnitt über 90% der Spiele gegen einen perfekten Minimax-Spieler zu gewinnen.
   Der größte Unterschied in der Konzeption des N-Tupel-Systems im Vergleich zu Lucas' Lösung [6] besteht darin, dass beim *Vier-Gewinnt*-Spiel die Verwendung zweier *LUTs* je N-Tupel essentiell für den Erfolg des Trainings sind.
   Mit einigen weiteren kleineren Verbesserungen, wie der Verwendung von jeweils vier Zuständen pro Zelle (Abschnitt 4.4) oder der Ausnutzung von Spielfeldsymmetrien (Abschnitt 4.6.2), lässt sich die Spielstärke des Agenten weiter verbessern. Während *Lucas* nur eine Verschlechterung der Lerngeschwindigkeit beobachten konnte, wenn auf die Symmetrien verzichtet wurde [6] , wirkt sich dies beim *Vier Gewinnt* vor allem auf die resultierende Spielstärke des Agenten negativ aus.
2. Der *TDL*-Agent wurde allein durch "*Self-Play*" trainiert, er erlernte *Vier Gewinnt* also ausschließlich anhand von Trainingsspielen gegen sich selbst. Der Einsatz eines externen Lehrer-Signals war daher für das Training nicht weiter nötig; der zuvor erwähnte Minimax-Agent wurde lediglich zur Evaluation der Spielstärke eingesetzt. Und auch sonst hatte der *TDL*-Agent keinen Zugriff auf spieltheoretisches Wissen irgendeiner Form.

3. Die Hypothese, dass *Vier Gewinnt* auch nur anhand weniger Trainingsspiele erlernbar ist, konnte nicht bestätigt werden. Während *Lucas* für das *Othello*-Spiel lediglich 1250 Trainingsspiele benötigte, um einen starken Spieler zu erzeugen [6] , sind hier etwas mehr als 2 Mio. Spiele nötig (Abschnitt 4.6.1). Dieser große Unterschied ist insbesondere deshalb erstaunlich, weil *Othello* ([9] , S. 167) einen deutlich größeren Zustandsraum als *Vier Gewinnt* [8] besitzt und daher zunächst als schwieriger zu erlernen scheint.
Eine Ursache könnte darin liegen, dass bei Ausnutzung von Spielfeldsymmetrien beim *Othello*-Spiel je Lernschritt gleichzeitig acht Stellungen gelernt werden, beim Vier-Gewinnt-Spiel lediglich zwei, wodurch beim *Vier Gewinnt* deutlich mehr Trainingsspiele nötig wären, um die gleiche Anzahl an Stellungen zu erlernen.
Eine weitere Erklärung könnte sein, dass *Lucas* für sein N-Tupel-System weniger und kürzere N-Tupel verwendet (lediglich 30 Tupel der Länge 2 bis 6) [6] , als in dieser Arbeit (siehe Punkt 4) zum Einsatz kommen. Dadurch wird die Spielfunktion des *Othello*-Agenten in einem deutlich größeren Umfang generalisiert, sodass in jedem Lernschritt eine größere Zahl an ähnlichen Stellungen erlernt werden (vgl. [11] , S. 259).

4. Die Konfigurationen für ein erfolgreiches Training sind nicht generell bestimmbar. Vor allem für die Lernschrittweite α (unter anderem abhängig von Zahl und Länge der N-Tupel) und die Explorationsrate ε müssen – je nach Situation – geeignete Konfigurationen gefunden werden. Unpassende Belegungen dieser Parameter führen zwar nicht zwangsläufig dazu, dass das Training fehlschlägt, allerdings kann die Lerngeschwindigkeit oder die resultierende Spielstärke etwas schlechter ausfallen. Die N-Tupel selbst wurden völlig zufällig, mithilfe des *Random-Walk-Verfahrens* (Abschnitt 4.5.1) generiert. Erwartungsgemäß konnte ein Anstieg der Spielstärke mit größerer Anzahl und Länge der N-Tupel beobachtet werden. Allerdings steigt der Speicherbedarf hierdurch deutlich an. Ein N-Tupel-System basierend auf 70 8-Tupeln reicht jedoch aus, um eine Erfolgsquote von etwa 90% gegen einen perfekt spielenden Minimax-Agenten zu erzielen (mit der Basiskonfiguration aus Abschnitt 4.1.5).

5. Insgesamt lassen sich die Trainingsdurchgänge sehr gut reproduzieren. Wiederholt man das Training für eine vorgegebene Konfiguration, so sind in der Regel keine größeren Abweichungen zwischen zwei Trainingsläufen zu beobachten. Auch die resultierende Spielstärke des Agenten kann in jedem Lauf neu erzielt werden.
Lässt man alle N-Tupel in jedem Trainingsdurchgang wieder neu erzeugen (bei Beibehaltung der übrigen Parameter wie Tupel-Länge und -Anzahl), können sich bei der resultierenden Spielstärke bzw. Erfolgsquote etwas größere Abweichungen (von etwa 10%) ergeben. In Ausnahmefällen kann das Training auch vollständig scheitern. Dies wurde jedoch nur äußerst selten beobachtet.

Auch wenn in dieser Arbeit bereits eine ganze Reihe von Untersuchungen angestellt und beschrieben worden sind, bleibt noch viel Raum für weitere Diskussionen und Entwicklungen in andere Richtungen. Einige Ideen konnten bereits in dieser Arbeit umgesetzt werden, viele andere konnten aus Zeitgründen jedoch nicht weiter verfolgt werden.

So werden beispielsweise die N-Tupel zurzeit lediglich nach einem "Random-Walk"-Verfahren generiert; eine interessante Fragestellung könnte jedoch darin liegen, zu prüfen, inwieweit sich die Generierungsvorschriften der N-Tupel optimieren lassen. Es wäre ebenfalls zu untersuchen, ob sich weitere Kriterien zur Bewertung der Qualität einzelner N-Tupel finden lassen, um so ungeeignete Tupel aus einem System aussortieren bzw. ersetzen zu können.

Aufgrund einer weiteren Frage, die darin besteht, ob zwei gute N-Tupel-Systeme in Kombination ein noch besseres Ergebnis liefern, könnte ein etwas anderer Ansatz verfolgt werden: Man erzeugt zunächst eine ganze Reihe von kleineren N-Tupel-Systemen und versucht anschließend durch verschiedene Kombination derer, das Ergebnis sukzessive zu verbessern.

Weiterhin sind bisher nur Systeme mit N-Tupeln gleicher Länge betrachtet worden. Möglicherweise kann man hier durch eine größere Vielfalt der N-Tupel gleichwertige oder sogar bessere Resultate erzielen.

Aufgrund des einfachen Konzeptes, der unkomplizierten Implementierung und der (nahezu) universellen Einsetzbarkeit der N-Tupel-Systeme, lassen sich eventuell bei anderen Brettspielen ähnliche Erfolge erzielen. Spiele wie etwa *Mühle* erscheinen hier besonders vielversprechend. Zum einen ist die Zustandsraumkomplexität des *Mühle*-Spiels (etwa 10^{10} Spielzustände) vergleichsweise überschaubar ([9] , S. 165) und zum anderen sind nur wenige Zustände je Abtastpunkt (nämlich genau drei) eines N-Tupels nötig, sodass sich der erforderliche Speicher – bedingt durch die Größe der *LUTs* – im Rahmen hält. Eventuell kann man sogar einen Schritt weiter gehen und das außerordentlich komplexe Spiel *Go* (beginnend mit kleineren Spielfeldgrößen) untersuchen.

Abschließend lässt sich festhalten, dass *RL* in Kombination mit N-Tupel-Systemen außerordentlich gute Ergebnisse bei den Spielen *Othello* [6] und *Vier Gewinnt* liefert. Dies liegt insbesondere an der beträchtlichen Zahl an Features (die ansonsten vom Entwickler selbst selektiert und kombiniert werden müssen), die das System generiert. Der Entwickler benötigt kein besonderes spieltheoretisches Wissen des betrachteten Brettspiels. Zwar werden nicht alle erzeugten Features hilfreich sein, das System lernt jedoch, diese außer Acht zu lassen, da aufgrund der widersprüchlichen Lernsignale keine Generalisierung möglich ist. Mithilfe der geeigneten Features kann der Agent sich wiederholende Muster erkennen und ist somit in der Lage, viele Spielsituationen korrekt einzuschätzen.

Literaturverzeichnis

[1] John McCarthy (2007): WHAT IS ARTIFICIAL INTELLIGENCE? Computer Science Department, Standford University (zuletzt abgerufen am: 13.06.2012).
http://www.formal.stanford.edu/jmc/whatisai/whatisai.html

[2] Karsten Bauermeister (o. J.): SCHACHCOMPUTER GESCHICHTE
(zuletzt abgerufen am: 13.06.2012).
http://www.schachcomputer.at/gesch1.htm

[3] Raúl Rojas et al. (o. J.): Konrad Zuses Plankalkül – Seine Genese und eine moderne Implementierung. Institut für Informatik, Freie Universität Berlin; S. 3-4.
www.zib.de/zuse/Inhalt/Programme/Plankalkuel/Genese/Genese.pdf

[4] M. Campbell et al. (2001): Deep Blue. IBM T.J. Watson Research Center Yorktown Heights.
http://sjeng.org/ftp/deepblue.pdf

[5] C. E. Shannon (1950): Programming a Computer for Playing Chess. In: *Philosophical Magazine*, Ser.7, Vol. 41, No. 314; S. 2.

[6] S. M. Lucas (2008): Learning to Play Othello with N-Tuple Systems. In: *Australian Journal of Intelligent Information Processing*, v. 4; S. 1-20.
http://algoval.essex.ac.uk/rep/games/NTOthello/NTupleOthello.pdf

[7] V. Allis (1988): A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins. *Masters Thesis*. Department of Mathematics and Computer Science, Vrije Universiteit Amsterdam.
www.connectfour.net/Files/connect4.pdf

[8] J. Tromp: John's Connect Four Playground (zuletzt abgerufen am: 13.06.2012).
http://homepages.cwi.nl/~tromp/c4/c4.html

[9] V. Allis (1994): Searching for Solutions in Games and Artificial Intelligence. *PhD thesis*. Department of Computer Science, University of Limburg.
http://fragrieu.free.fr/SearchingForSolutions.pdf

[10] R. S. Sutton, A. G. Barto (1998): Reinforcement Learning: An Introduction. MIT Press, Cambridge, MA.
Online-Version des Buches:
http://webdocs.cs.ualberta.ca/~sutton/book/ebook/the-book.html

[11] L. P. Kaelbling et al. (1996): Reinforcement Learning: A Survey. In: *Journal of Artifcial Intelligence Research 4*; S. 237-285.
http://www.ceng.metu.edu.tr/~polat/ceng580/kaelbling96a.pdf

[12] W. Konen (2008): Reinforcement Learning für Brettspiele: Der Temporal Difference Algorithmus. *Technical Report*, Institut für Informatik, Fachhochschule Köln, Gummersbach.
http://www.gm.fh-koeln.de/~konen/Publikationen/TR_TDlambda.pdf

[13] A.L. Samuel (1959): Some Studies in Machine Learning Using the Game of Checkers. *IBM journal of Research and Development*, 3(3); S. 210-229.
http://www.cs.unm.edu/~terran/downloads/classes/cs529-s11/papers/samuel_1959_B.pdf

[14] G. Tesauro (1996): Temporal Difference Learning and TD-Gammon . In: *Communications of the ACM*, Vol. 38, No. 3.
http://www.research.ibm.com/massive/tdl.html

[15] M. Stenmark (2005): Synthesizing board evaluation functions for Connect-4 using machine learning techniques. *Master Thesis*, Østfold University College, Norwegen.

[16] P. Sommerlund (1996): Artificial Neural Nets Applied to Strategic Games; o. O.
http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.56.4690&rep=rep1&type=pdf

[17] D. Curran, C. O'Riordan (2004): Evolving Connect-4 playing neural networks using cultural learning. *Techincal Report NUIG-IT-081204*. Department of Information Technology , National University of Ireland, Galway.
http://www.cs.ucc.ie/~dc17/pubs/CurranAICS2004.pdf

[18] J. Schwenk (2008): Reinforcement Learning zum maschinellen Erlernen von Brettspielen am Beispiel des Strategiespiels „4-Gewinnt". *Diplomarbeit*, Fachhochschule Köln, Campus Gummersbach.
http://www.gm.fh-koeln.de/~konen/Diplom+Projekte/PaperPDF/DiplomSchwenck08.pdf

[19] W. W. Bledsoe, I. Browning (1959): Pattern Recognition and Reading by Machine. In: *1959 PROCEEDINGS OF THE EASTERN JOINT COMPUTER CONFERENCE*; S 225-232.

[20] S. M. Lucas (1997): Face Recognition with the continuous n-tuple classifier. In: *Proceedings of the British Machine Vision Conference*, S. 222 -231).
http://algoval.essex.ac.uk/rep/imagerec/lucas97face.pdf

[21] M. Morciniec und R. Rohwer (1995): The n-tuple Classifier: Too good to ignore. *Technical Report*. Department of Computer Science an Applied Mathematics, Aston University, Birmingham.
http://eprints.aston.ac.uk/515/

[22] W. Konen und T. Bartz-Beielstein (2008): Reinforcement Learning: Insights from Interesting Failures in Parameter Selection. In: G. Rudolph (Hrsg.), *Proc. Parallel Problem Solving From Nature (PPSN' 2008)*. Springer, Berlin; S. 478-487.
http://www.gm.fh-koeln.de/~konen/Publikationen/ppsn2008.pdf

[23] M. Thill et al. (2012): Reinforcement Learning with N-tuples on the Game Connect-4. In: Cutello, V. and Pavone, M. (Hrsg.),*International Conference on Parallel Problem Solving from Nature 2012*, Taormina, Italien, Sep. 2012).

[24] M. Thill (2012): Einsatz von N-Tupel-Systemen mit TD-Learning für strategische Brettspiele am Beispiel von Vier Gewinnt. *Praxisprojekt*, Institut für Informatik, Fachhochschule Köln, Campus Gummersbach.

[25] Albert L. Zobrist (1970): A new hashing method with application for game playing. *Technical Report 88*. Computer Sciences Department, The University of Wisconsin, Madison.
research.cs.wisc.edu/techreports/1970/TR88.pdf

[26] Aske Plaat et al. (1994): Nearly Optimal Minimax Tree Search? *Technical Report 94-19*. Department of Computing Science, University of Alberta, Edmonton, Alberta; S. 9-13.
http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.46.7740

[27] L. Bremer (2010): Download-Seite des Vier-Gewinnt-Programmes Mustrum (zuletzt abgerufen am: 13.06.2012).
http://www.mustrum.de/mustrum.html

[28] S. Edelkamp und P. Kissmann (2008): Symbolic Classification of General Two-Player Games. *German Conference on Artificial Intelligence (KI). Kaiserslautern*; S. 185-192.
http://www.tzi.de/~edelkamp/publications/conf/ki/EdelkampK08-1.pdf

A. Anhang

A.1. Hinweise zur beigefügten CD

Inhalt der CD

- Software-Framework (Eclipse-Projekt in Java) mit einer Trainings- und Testumgebung für das Spiel *Tic Tac Toe*, das lediglich um das N-Tupel-System erweitert wurde (siehe hierzu auch [12] und [22]).
- Software-Framework (Eclipse-Projekt in Java) mit einer Trainings- und Testumgebung inkl. eine Hilfe-Datei für das Spiel *Vier Gewinnt*, basierend auf der *Tic-Tac-Toe*-Umgebung.
- Diverse Hilfsprogramme:
  - Programm zum Zählen von Spielstellungen mit x Spielsteinen in *Tic Tac Toe* und *Vier Gewinnt*
  - Programm zum Zählen der realisierbaren N-Tupel-Zustände (siehe Abschnitt 4.4.2); Java-Programm sowie ein Maple-Skript.
  - Generierungs-Programm für gewisse Code-Abschnitte des Minimax-Agenten.
  - Programme zur Erzeugung, Konvertierung und Sortierung der Eröffnungsdatenbanken – mit 8,10 und 12 Spielsteinen – für *Vier Gewinnt* (in C bzw. C++).
  - Weitere kleinere Programme
- Trainings- und Testergebnisse für *Vier Gewinnt*: Rohdaten, Excel-Sheets und R-Skripte zur Erzeugung der Diagramme. Weitere, in dieser Arbeit nicht erwähnten Ergebnisse.
- Eine ganze Reihe von Ergebnissen zum Spiel *Tic Tac Toe*, die in dieser Arbeit keine Rolle spielten.
- Verwendete Literatur

A.2. Details zur Berechnung der realisierbaren Zustände

In diesem Abschnitt soll kurz gezeigt werden, wie die in Abschnitt 4.4.2 genannten Funktionen zustande kommen. Hierfür ist zunächst die Betrachtung eines Beispiels vorteilhaft.

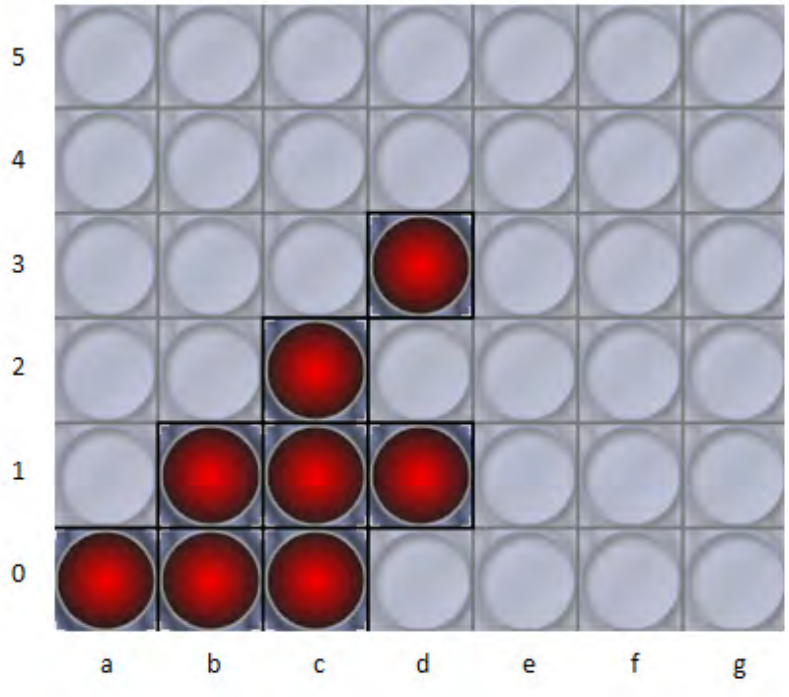


Abbildung A.1. Das N-Tupel, dass im Beispiel betrachtet wird.

Beispiel

Beispielhaft soll die Berechnung für das N-Tupel in Abb. A.1 durchgeführt werden. Spaltenweises Vorgehen ist am sinnvollsten (Kombinationen werden für jede Spalte bestimmt und im Anschluss multipliziert):

$$L_{\mathrm{T}} = \prod_{i=0}^{6} S_i \tag{A.1}$$

Die einzelnen Spalten werden von oben nach unten durchlaufen. Für jeden auftretenden Abtastpunkt vergrößert sich die Zahl realisierbarer Zustände der Spalte. Unten angekommen, hat man die Gesamtzahl für die entsprechende Spalte (n_0).

Kombination für die a-Spalte

$\boldsymbol{m = 3}$	$\boldsymbol{m = 4}$
$n_0 = 3$	$n_0 = 3$

Kombination für die b-Spalte

$\boldsymbol{m = 3}$	$\boldsymbol{m = 4}$
$n_1 = 3$	$n_1 = 4$
$n_0 = 2n_1 + 1 \cdot 1 = 7$ **Anmerkung:** $2n_1$ deshalb, weil nur für einen gelben bzw. roten Stein oberhalb noch die 3 Kombinationen möglich sind. Ist b_0 (unterste Zelle der b-Spalte) unbelegt, dann kann es für die darüber liegenden Zellen nur eine Möglichkeit geben (nämlich leer).	$n_0 = 2(n_1 - 1) + 1 \cdot 1 = 7$ **Anmerkung:** Sollte die unterste Zelle b_0 der b-Spalte belegt sein (Gelb oder Rot), so sind in der Zelle oberhalb (b_1) noch drei Belegungen möglich (Gelb, Rot, Leer aber erreichbar), daher: $2(n_1 - 1)$. Ist b_0 (unterste Zelle der b-Spalte) unbelegt, dann kann es für den darüber liegenden Abtastpunkt nur eine Möglichkeit geben (nämlich leer).

Kombination für die c-Spalte

$\boldsymbol{m = 3}$	$\boldsymbol{m = 4}$
$n_2 = 3$	$n_2 = 4$
$n_1 = 2n_2 + 1 = 7$	$n_1 = 2(n_1 - 1) + 2 = 8$ **Anmerkung**: Ist die Zelle c_1 belegt, sind oberhalb noch drei Belegungen möglich; daher: $2(n_1 - 1)$. Ist c_1 unbelegt, so kann c_2 nur den Zustand "Leer" annehmen, c_1 aber "Leer" und "Leer und erreichbar"; daher noch die Addition von 2.
$n_0 = 2n_1 + 1 = 15$ **Anmerkung:** Analog zur b-Spalte.	$n_0 = 2(n_1 - 1) + 1 = 15$ **Anmerkung:** Analog zur b-Spalte.

Kombination für die d-Spalte

$\boldsymbol{m = 3}$	$\boldsymbol{m = 4}$
$n_3 = 3$	$n_3 = 4$
$n_2 = n_3 = 3$ **Anmerkung**: Da die Zelle d_2 kein Abtastpunkt des N-Tupels ist, ändert sich nichts weiter.	$n_2 = n_3 = 4$ **Anmerkung**: Da die Zelle d_2 kein Abtastpunkt des N-Tupels ist, ändert sich nichts weiter.
$n_1 = 2n_2 + 1 = 7$	$n_1 = 2n_2 + 2 = 10$ **Anmerkung:** Sollte die Zelle d_1 belegt sein (Rot oder Gelb), dann sind in Abtastpunkt d_3 nach wie vor alle 4 Kombinationen möglich (daher: $2n_2$). Dies liegt daran, dass zwischen beiden Abtastpunkten eine Zelle liegt, die nicht am N-Tupel beteiligt ist.
$n_0 = n_1 = 7$ **Anmerkung:** Die unterste Zelle d_0 stellt keinen Abtastpunkt des Tupels dar, daher ändert sich am Ergebnis nichts.	$n_0 = n_1 = 10$ **Anmerkung:** Die unterste Zelle d_0 stellt keinen Abtastpunkt des Tupels dar, daher ändert sich am Ergebnis nichts.

Da die nachfolgenden Spalten keine Abtastpunkte enthalten, ändert sich am Ergebnis ansonsten nichts weiter.

Gesamtergebnis für das Beispiel

$\boldsymbol{m = 3}$	$\boldsymbol{m = 4}$
$L_\mathrm{T} = 3 \cdot 7 \cdot 15 \cdot 7 = 2205$	$L_\mathrm{T} = 3 \cdot 7 \cdot 15 \cdot 10 = 3150$

Schlussfolgerungen

Beim Durchlaufen der Spalten von oben nach unten gilt:

- Allgemein gilt $n_i = n_{i+1}$, wenn die aktuelle Zelle KEIN Element (Abtastpunkt) des N-Tupels ist.
- Ist die die aktuelle Zelle ein Abtastpunkt des N-Tupels
  - und weiterhin $m = 3$, dann gilt: $n_i = 2n_{i+1} + 1$.
  - und $m = 4$:
    - Ist die Zelle direkt oberhalb auch ein Abtastpunkt, dann gilt: $n_i = 2(n_{i+1} - 1) + z$.
    - Ist die Zelle direkt oberhalb KEIN Abtastpunkt, dann gilt: $n_i = 2n_{i+1} + z$.
    - Hierbei ist zu beachten, dass $z = 1$, wenn die aktuelle Zelle die Unterste der Spalte ist; ansonsten ist $z = 2$

Im Folgenden werden Funktionen $S_m(x, y)$ gesucht, die für eine Spalte $x \in \mathbb{N}_0$ die Anzahl der Kombinationen berechnet. Der Zeilen-Parameter $y \in \mathbb{N}_0$ dient hierbei der Rekursion. Die Herleitung soll zunächst für die Fälle $m = 3$ und $m = 4$ getrennt geschehen. Beide Funktionen sollen jedoch in

$$L_\mathrm{T} = \prod_{i=0}^{6} S_m(i, 0) \tag{A.2}$$

einsetzbar sein.

Herleitung einer Funktion $S_3(x, y)$ für $m = 3$

Es müssen lediglich die zwei betreffenden Punkte aus den Schlussfolgerungen berücksichtigt werden. Daher ist schnell ersichtlich, dass sich die Funktion folgendermaßen beschreiben lässt:

$$S_3(x,y) = \begin{cases} S(x,y+1), & y < 6 \ \wedge \ (x,y) \notin \mathrm{T} \\ 2S(x,y+1)+1, & y < 6 \ \wedge \ (x,y) \in \mathrm{T} \\ 1, & y \geq 6 \end{cases} \tag{A.3}$$

Durch die Rekursion, ergibt sich für jede Spalte eine Form, die dem Horner-Schema entspricht (abhängig von der Zahl der Abtastpunkte innerhalb einer Spalte):

$$S_3(x) = 2 \cdot (\cdots 2 \cdot (2 \cdot 1 + 1) + \cdots) + 1 \tag{A.4}$$

bzw.:

$$S_3(x,y) = 1 + 2^1 + \cdots + 2^k = \frac{1 - 2^{k+1}}{1 - 2} = 2^{k+1} - 1 \tag{A.5}$$

mit der Anzahl k der Abtastpunkte je Spalte:

$$k = \sum_{j=5}^{y} p(x,j) \tag{A.6}$$

Herleitung einer Funktion $S_4(x, y)$ für $m = 4$

Für den Fall $m = 4$ sind einige Punkte mehr zu beachten. Zum einen muss der – in den Schlussfolgerungen beschriebene – Parameter z den Wert 1 liefern, wenn die aktuelle Zelle die Unterste der Spalte ist und ansonsten den Wert 2. Da die Spalte hierfür keine Rolle spielt ,lässt sich folgende Aussage treffen:

$$z_4(y) = \begin{cases} 2, & 1 \leq \ y < 6 \\ 1, & y = 0 \\ 0, & sonst \end{cases} \tag{A.7}$$

Weiterhin muss – wenn die aktuelle Zelle ein Element des N-Tupels darstellt – geprüft werden, ob die Zelle oberhalb auch ein Abtastpunkt ist (dadurch resultiert: $n_i = 2(n_{i+1} - 1) + z$) oder aber nicht ($n_i = 2n_{i+1} + z$). Eine Funktion, die hierfür geeignet scheint, lautet:

$$p(x,y) = \begin{cases} 1, & (x,y) \in \mathrm{T} \\ 0, & sonst \end{cases} \qquad mit \quad x,y \in \mathbb{N}_0 \tag{A.8}$$

Setzt man diese Teilergebnisse zusammen, erhält man die Funktion für die Variante $m = 4$:

$$S_4(x,y) = \begin{cases} S(x,y+1), & y < 6 \ \wedge \ (x,y) \notin \mathrm{T} \\ 2\big(S(x,y+1) - \ p(x,y+1)\big) + z_4(y), & y < 6 \ \wedge \ (x,y) \in \mathrm{T} \\ 1, & y \geq 6 \end{cases} \tag{A.9}$$

Herleitung einer allgemeingültigen Funktion $S_m(x,y)$ für $m = 3$ und $m = 4$

Da $S_3(x,y)$ und $S_4(x,y)$ (A.3 bzw. A.9) sich bereits stark ähneln, soll eine allgemeingültige Funktion, die beide Varianten vereint, ermittelt werden. Ein Vergleich zwischen $S_3(x,y)$ und $S_4(x,y)$ liefert bereits zu Beginn:

$$S_m(x,y) = \begin{cases} S(x,y+1), & y < 6 \ \wedge \ (x,y) \notin \mathrm{T} \\ A_m(x,y), & y < 6 \ \wedge \ (x,y) \in \mathrm{T} \\ 1, & y \geq 6 \end{cases} \quad ,mit\ m \in \{3,4\} \tag{A.10}$$

mit

$$A_3(x,y) = 2S(x,y+1) + 1 = 2S(x,y+1) + z_3(y) \tag{A.11}$$

und

$$A_4(x,y) = 2\big(S(x,y+1) - \ p(x,y+1)\big) + z_4(y) \tag{A.12}$$

Da der Term $p(x,y+1)$ in A.12 nur für den Fall $m = 4$ berücksichtigt werden soll, kann man diesem den Faktor $(m-3)$ voranstellen. Weiterhin ist es möglich, eine Funktion $z(y)$ zu finden, die $z_3(y) = 1$ und $z_4(y)$ vereint:

$$z(y) = z_m(y) = \begin{cases} m-2, & 1 \leq y < 6 \\ 1, & y = 0 \\ 0, & sonst \end{cases} \quad mit \quad y \in \mathbb{N}_0 \tag{A.13}$$

Dadurch ergibt sich

$$A_m(x,y) = 2\big(S(x,y+1) - \ (m-3)p(x,y+1)\big) + z(y) \tag{A.14}$$

und letztendlich:

$$S_m(x,y) = \begin{cases} S(x,y+1), & y < 6 \ \wedge \ (x,y) \notin \mathrm{T} \\ 2\big(S(x,y+1) - \ (m-3)p(x,y+1)\big) + z(y), & y < 6 \ \wedge \ (x,y) \in \mathrm{T} \\ 1, & y \geq 6 \end{cases} \tag{A.15}$$

bzw.:

$$\begin{aligned} S_m(x,y) \quad &= \begin{cases} [1-p(x,y)] \cdot S(x,y+1) + p(x,y) \cdot A_m(x,y), & y < 6 \\ 1, & y \geq 6 \end{cases} \\ &= \begin{cases} S(x,y+1)(p(x,y)+1) + p(x,y)\big(z(y) - 2(m-3)p(x,y+1)\big), & y < 6 \\ 1, & y \geq 6 \end{cases} \end{aligned} \tag{A.16}$$

A.3. Anzahl der non-terminalen Stellungen in *Tic Tac Toe*

Terminale Stellung: Stellung, bei der unmittelbar eine Aussage über Sieg, Niederlage oder Unentschieden getroffen werden kann und kein weiterer Zug mehr möglich ist.

Anzahl der Stellungen mit 0 Spielsteinen

Es gibt nur eine solche Stellung, daher:

$$n_0 = 1 \tag{A.17}$$

Anzahl der Stellungen mit einem Spielstein

Der anziehende Spieler hat neun Möglichkeiten, um seinen Stein zu setzen, da noch kein Feld belegt ist:

$$n_1 = 9 \tag{A.18}$$

Anzahl der Stellungen mit zwei Spielsteinen

Jeder Spieler setzt jeweils einen Stein, die beide unterscheidbar sind:

$$n_2 = 9 \cdot 8 = 72 \tag{A.19}$$

Anzahl der Stellungen mit drei Spielsteinen

Der Anziehende platziert zwei Spielsteine auf dem Feld, der Nachziehende nur einen. Zu beachten ist, dass die beiden Spielsteine des Anziehenden nicht unterscheidbar sind (Objekte einer Klasse). Durch die Vertauschung dieser beiden Steine entsteht wieder dieselbe Stellung.

$$n_3 = \frac{9 \cdot 8 \cdot 7}{2} = 252 \tag{A.20}$$

Anzahl der Stellungen mit vier Spielsteinen

Jeweils zwei Felder des Spielfeldes werden durch die beiden Spieler belegt. Hierbei handelt es sich um vier Objekte zweier Klassen.

$$n_4 = \frac{1}{2 \cdot 2} \cdot \frac{9!}{5!} = 756 \tag{A.21}$$

Im Weiteren gilt:

- S_{ges} sei die Menge aller möglichen Stellungen mit k Spielsteinen.
- S_{ter} sei die Menge der terminalen Stellungen resultierend aus k Spielsteinen (Stellungen mit einem Sieg des Anziehenden).
- Die Menge der relevanten Stellungen ist daher: $S \coloneqq S_{ges} \setminus S_{ter}$, wobei $S_{ter} \subset S_{ges}$. Weiterhin beschreiben $S_{ter(W)}$ und $S_{ter(S)}$ die Menge der terminalen Stellungen für Weiß (Anziehender) bzw. Schwarz.

Anzahl der Stellungen mit fünf Spielsteinen

Die Gesamtzahl aller Stellungen mit fünf Spielsteinen lautet:

$$\left|S_{ges}\right| = \frac{1}{3! \cdot 2!} \cdot \frac{9!}{4!} = 1260 \tag{A.22}$$

Der Anziehende setzt genau drei Spielsteine. Er kann also maximal über eine von *acht* möglichen Dreier-Reihen verfügen. Kombiniert mit den beiden anderen Spielsteinen ergibt sich eine Zahl von

$$|S_{ter}| = 8 \cdot \frac{6 \cdot 5}{2!} = 120 \tag{A.23}$$

terminalen Stellungen. Somit erhält man letztendlich:

$$n_5 = |S| = \left|S_{ges} \setminus S_{ter}\right| = \left|S_{ges}\right| - |\,S_{ter}| = 1140 \tag{A.24}$$

Anzahl der Stellungen mit sechs Spielsteinen

Die Berechnung erfolgt ähnlich wie für n_5. Es muss jedoch beachtet werden, dass jetzt beide Spieler terminale Stellungen erzeugen können. Es ist daher möglich, dass der Anziehende (Weiß) oder der Nachziehende (Schwarz) oder sogar beide gleichzeitig über eine Dreierreihe verfügen. Lediglich bei einer diagonalen Dreier-Reihe kann der jeweils andere keine eigene Dreier-Reihe besitzen.

$$\left|S_{ges}\right| = \frac{1}{3! \cdot 3!} \cdot \frac{9!}{3!} = 1680 \tag{A.25}$$

$$\left|S_{ter(W)}\right| = \left|S_{ter(S)}\right| = 8 \cdot \frac{6 \cdot 5 \cdot 4}{3!} = 160 \tag{A.26}$$

Es muss lediglich beachtet werden, dass in gewissen Stellungen beide Spieler jeweils eine Dreier-Reihe besitzen. Diese Stellungen dürfen nicht doppelt gezählt werden. Daher wird noch $\left|S_{ter(W)} \cap S_{ter(S)}\right|$ von der Anzahl der terminalen Stellungen subtrahiert.

Folgende Punkte sind hierbei zu beachten, die auch im Weiteren gelten:

1. Diagonale Dreier-Reihen können in diesen Stellungen nicht vorhanden sein.
2. Das Spielfeld lässt jeweils *drei* vertikale und horizontale Dreier-Reihen zu. (Faktor: 3)
3. Eine horizontale / vertikale Dreier-Reihe kann mit *zwei* weiteren horizontalen / vertikalen Reihen des Gegners kombiniert werden. (Faktor: 2)
4. Horizontale und vertikale Reihen müssen nicht separat betrachtet werden. Es reicht aus, dies für eine Variante zu berechnen und die Zahl anschließend mit *zwei* zu multiplizieren.

Beachtet man diese vier Punkte so erhält man:

$$\left|S_{ter(W)} \cap S_{ter(S)}\right| = \overset{Pkt.\ 2}{\overbrace{3}} \cdot \overset{Pkt.\ 3}{\overbrace{2}} \cdot \overset{Pkt.\ 4}{\overbrace{2}} = 12 \tag{A.27}$$

$$|S_{ter}| = \left|S_{ter(W)}\right| + \left|S_{ter(S)}\right| - \left|S_{ter(W)} \cap S_{ter(S)}\right| = 308 \tag{A.28}$$

$$n_6 = |S| = \left|S_{ges} \setminus S_{ter}\right| = 1372 \tag{A.29}$$

Anzahl der Stellungen mit sieben Spielsteinen

Die Berechnung kann analog zum vorherigen Abschnitt durchgeführt werden:

$$\left|S_{ges}\right| = \frac{1}{4! \cdot 3!} \cdot \frac{9!}{2!} = 1260 \tag{A.30}$$

Die Anzahl an terminalen Stellungen, die beide Spieler erzeugen, lautet:

$$\left|S_{ter(W)}\right| = 8 \cdot \overset{W}{\overbrace{6}} \cdot \frac{\overset{S}{\overbrace{6 \cdot 5 \cdot 4}}}{3!} = 480 \tag{A.31}$$

$$\left|S_{ter(S)}\right| = 8 \cdot \frac{\overset{W}{\overbrace{6 \cdot 5 \cdot 4 \cdot 3}}}{4!} = 120 \tag{A.32}$$

$$\left|S_{ter(W)} \cap S_{ter(S)}\right| = \overset{siehe\ A.27}{\overbrace{3 \cdot 2 \cdot 2}} \cdot \overset{letzter\ Spielstein\ W}{\overbrace{3}} = 36 \tag{A.33}$$

Damit folgt unmittelbar:

$$|S_{ter}| = \left|S_{ter(W)}\right| + \left|S_{ter(S)}\right| - \left|S_{ter(W)} \cap S_{ter(S)}\right| = 564 \tag{A.34}$$

$$n_7 = |S| = \left|S_{ges} \setminus S_{ter}\right| = 696 \tag{A.35}$$

Anzahl der Stellungen mit acht Spielsteinen

Auch bei acht Spielsteinen kann jeder Spieler max. eine Dreier-Reihe erzeugen (bei 9 Steinen könnte der Anziehende theoretisch zwei erhalten).

$$\left|S_{ges}\right| = \frac{9!}{4! \cdot 4!} = 630 \tag{A.36}$$

Die Anzahl an terminalen Stellungen, die beide Spieler erzeugen lautet:

$$\left|S_{ter(W)}\right| = \left|S_{ter(S)}\right| = 8 \cdot \overbrace{6}^{W} \cdot \frac{\overbrace{5 \cdot 4 \cdot 3 \cdot 2}^{S}}{4!} = 240 \tag{A.37}$$

$$\left|S_{ter(W)}\right| = \left|S_{ter(S)}\right| = 8 \cdot \overbrace{6}^{S} \cdot \frac{\overbrace{5 \cdot 4 \cdot 3 \cdot 2}^{W}}{4!} = 240 \tag{A.38}$$

$$\left|S_{ter(W)} \cap S_{ter(S)}\right| = \overbrace{3 \cdot 2 \cdot 2}^{siehe\ A.27} \cdot \overbrace{3 \cdot 2}^{\substack{letzte\ Spielsteine \\ beider\ Spieler}} = 72 \tag{A.39}$$

$$|S_{ter}| = \left|S_{ter(W)}\right| + \left|S_{ter(S)}\right| - \left|S_{ter(W)} \cap S_{ter(S)}\right| = 408 \tag{A.40}$$

Daraus ergibt sich:

$$n_8 = |S| = \left|S_{ges} \setminus S_{ter}\right| = 222 \tag{A.41}$$

Anzahl der Stellungen mit neun Spielsteinen

Alle Stellungen mit 9 Spielsteinen sind zwangsläufig terminale Stellungen (Unentschieden / Sieg für den Anziehenden). Diese sind also in diesem Fall nicht relevant.

Zusammenfassung

Letztendlich ergibt sich eine Zahl von

$$n = \sum_{i=0}^{8} n_i = 4520 \tag{A.42}$$

non-terminalen Stellungen.

Erklärung

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

Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht 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 bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

(Ort, Datum) (Unterschrift)