FernUniversität in Hagen
Fakultät für Mathematik und Informatik

AIG
Artificial Intelligence Group

Reinforcement Learning für Brettspiele

Masterarbeit

zur Erlangung des Grades *Master of Science (M.Sc.)*
im Studiengang Praktische Informatik

vorgelegt von
Thorsten Clausing

Erstgutachter: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Betreuer: Prof. Dr. Matthias Thimm
Artificial Intelligence Group

Erklärung

Ich erkläre, dass ich die Masterarbeit selbstständig und ohne unzulässige Inanspruchnahme Dritter verfasst habe. Ich habe dabei nur die angegebenen Quellen und Hilfsmittel verwendet und die aus diesen wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht. Die Versicherung selbstständiger Arbeit gilt auch für enthaltene Zeichnungen, Skizzen oder graphische Darstellungen. Die Masterarbeit wurde bisher in gleicher oder ähnlicher Form weder derselben noch einer anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht. Mit der Abgabe der elektronischen Fassung der endgültigen Version der Masterarbeit nehme ich zur Kenntnis, dass diese mit Hilfe eines Plagiatserkennungsdienstes auf enthaltene Plagiate geprüft werden kann und ausschließlich für Prüfungszwecke gespeichert wird.

Der Veröffentlichung dieser Arbeit auf der Webseite des Lehrgebiets Künstliche Intelligenz und damit dem freien Zugang zu dieser Arbeit stimme ich ausdrücklich zu.

Für diese Arbeit erstellte Software wurde quelloffen verfügbar gemacht, ein entsprechender Link zu den Quellen ist in dieser Arbeit enthalten. Gleiches gilt für angefallene Forschungsdaten.

Berlin, 19.12.2025

. . . .

(Ort, Datum) (Unterschrift)

Zusammenfassung

Die vorliegende Arbeit geht der Frage nach, wie man einem Computer mit Methoden des *Reinforcement Learning* (RL) ein Brettspiel beibringen kann. Dies wird am Beispiel des Spiels Reversi untersucht. Nach einer einführenden Darstellung der Anwendung von RL auf Brettspiele wird gezeigt, wie Reversi für diesen Kontext formal modelliert werden kann. Dann wird ein einfacher Algorithmus entwickelt und implementiert, der auf Grundlage bisher beobachteter Ergebnisse, die in einer Tabelle festgehalten werden, abschätzt, welcher Zug in einer gegebenen Stellung am erfolgversprechendsten ist. Dieser geradlinige „tabulare" Algorithmus erweist sich jedoch aufgrund der großen Anzahl möglicher Reversi-Stellungen als nicht zielführend und erreicht in der experimentelle Überprüfung nur eine geringe Spielstärke. Untersucht wird, ob diese Probleme überwunden werden können, wenn die Tabelle des Algorithmus durch ein neuronales Netzwerk ersetzt wird. Als einfachste Variante wird dieses Netzwerk mit bereits vorliegenden Stellungsabschätzungen trainiert. Experimente mit diesem Ansatz belegen eine sehr erhebliche Steigerung der Spielstärke. Als zweite Variante wird ein „tiefer" RL-Algorithmus konstruiert, bei dem das Netzwerk kontinuierlich mit neuen Beobachtungen (weiter-)trainiert wird, während der Computer Spielerfahrung sammelt. Es gelingt jedoch in der experimentellen Überprüfung nicht, mit diesem komplexeren Vorgehen die Spielstärke des einfacheren Netzalgorithmus zu übertreffen.

Abstract

This paper explores the question of how to teach a computer to play a board game using reinforcement learning (RL) methods. This is examined using the example of the game Reversi. After an introductory presentation of the application of RL to board games, we show how Reversi can be formally modeled for this context. Then, a simple algorithm is developed and implemented which, based on previously observed results recorded in a table, estimates which move is most promising in a given position. However, this straightforward tabular algorithm proves to be ineffective due to the large number of possible Reversi positions and achieves only a low playing strength in experimental testing. We investigate whether these problems can be overcome by replacing the algorithm's table with a neural network. As the simplest variant, this network is trained with existing position estimates. Experiments with this approach demonstrate a very significant increase in playing strength. As a second variant, a deep RL algorithm is constructed in which the network is continuously (further) trained with new observations while the computer gains playing experience. However, in experimental testing, it has not been possible to surpass the playing strength of the simpler network algorithm with this more complex approach.

Inhaltsverzeichnis

1 Überblick **1**

2 Reinforcement Learning **2**
2.1 Grundmodell . . . 3
2.2 Lösungsalgorithmen . . . 4
2.3 Brettspiele als Anwendungsgebiet für RL . . . 6
2.3.1 Forschungsarbeiten zu Brettspiel-Lernalgorithmen . . . 6
2.3.2 Brettspiele im Grundmodell . . . 7

3 Das Spiel Reversi **9**
3.1 Spielregeln . . . 9
3.2 Strategische Eigenschaften von Reversi . . . 11
3.3 Reversi-Programme . . . 12

4 Geradlinige „tabulare“ Implementierung eines RL-Reversi-Spielers **14**
4.1 Tabelle und Zugauswahl . . . 14
4.2 Gegner und Vergleichsmaßstäbe . . . 18
4.3 Implementierungsdetails . . . 20
4.4 Experimentelle Ergebnisse des „tabularen“ RL-Reversi-Spielers . . . 23
4.4.1 Training der ϵ-*greedy*-Variante des Lernenden Spielers gegen sich selbst . . . 24
4.4.2 Training der σ-*greedy*-Variante des Lernenden Spielers gegen sich selbst . . . 25
4.4.3 Training des Lernenden Spielers gegen den Stochastischen Spieler . . . 27
4.4.4 Testergebnisse gegen den Minimax-Spieler . . . 30
4.5 Zwischenfazit . . . 33

5 Geradlinige Implementierung eines RL-Reversi-Spielers mit Netzwerk statt Tabelle **34**
5.1 Auswahl einer geeigneten Netzarchitektur . . . 34
5.2 Implementierungsdetails . . . 35
5.3 Experimentelle Ergebnisse des RL-Reversi-Spielers mit Netzwerk . . . 38
5.3.1 Training der Bewertungsnetze . . . 38
5.3.2 Test der Spielstärke . . . 41
5.3.2.1 Testergebnisse gegen den Stochastischen Spieler . . . 41
5.3.2.2 Testergebnisse gegen den Minimax-Spieler mit Tiefe 4 42
5.3.2.3 „Schwarze“ und „weiße“ Stellungen . . . 44
5.3.2.4 Strategisch äquivalente Stellungen . . . 45
5.3.2.5 Direktvergleich zwischen Version 1 und Version 2 . 47
5.3.2.6 Testergebnisse gegen den Minimax-Spieler mit größerer Tiefe . . . 49

5.4 Zwischenfazit 51

6 Implementierung eines tiefen RL-Reversi-Spielers 52

6.1 Abwechselnde Integration von Erfahrungssammlung und Netztraining in einen Algorithmus 52

6.2 Implementierungsdetails 55

6.3 Experimentelle Ergebnisse des tiefen RL-Reversi-Spielers 57

6.3.1 Grundvariante mit einfacher Trainingsphase 57

6.3.2 Grundvariante mit intensivierter Trainingsphase 60

6.3.3 Training mit angereicherten Daten 62

6.3.4 Hybrides Training mit Netz und Tabelle 66

6.4 Zwischenfazit 68

7 Zusammenfassung der Ergebnisse 69

1 Überblick

Ziel der vorliegenden Arbeit ist es, auf möglichst einfache Weise rein auf Basis von elementaren Reinforcement-Learning-Methoden ein Brettspiel-Computerprogramm zu entwickeln. Dies wird am konkreten Beispiel des Spiels Reversi versucht. Spiele zeichnen sich durch klar definierte Regeln und überschaubare Problemstellungen aus und sind daher traditionell ein wichtiges Erprobungsgebiet für Methoden der Künstlichen Intelligenz. Auch bei der Entwicklung fortgeschrittener Reinforcement-Learning-Verfahren haben Anwendungen auf (Brett- und Video-)Spiele eine wichtige Rolle gespielt. Beispiele dafür bieten [12] und [22]. Reinforcement Learning beruht auf der recht einfachen Idee, in einer Entscheidungssituation diejenige Aktion zu wählen, mit der man in der Vergangenheit in der gleichen oder in ähnlichen Situationen die besten Erfahrungen gemacht hat. Mit daraus abgeleiteten Methoden wird heute eine Vielzahl unterschiedlichster Probleme angegangen, von der Robotersteuerung (siehe [11]) bis zur Feineinstellung von generativen Sprachmodellen (siehe z. B. [14]).

Meine Untersuchung ist wie folgt aufgebaut: Zuerst wird das mathematisch auf Markov-Ketten beruhende Grundmodell des Reinforcement Learning mitsamt der dafür entwickelten theoretischen Lösungsansätze vorgestellt und gezeigt, wie Brettspiele in diesem Grundmodell behandelt werden können (Abschnitt 2). Dann werden die Regeln von Reversi, seine strategischen Besonderheiten und existierende Reversi-Programme dargestellt (Abschnitt 3).

Danach wird eine geradlinige (oder „naive“) Umsetzung der Grundidee des Reinforcement Learning in einem einfachen Algorithmus, der die beobachteten Ergebnisse verschiedener Züge in einer Tabelle festhält, entwickelt und als Python-Programm implementiert (Abschnitt 4). Ein solcher Algorithmus muss sicherstellen, dass hinreichend viele Erfahrungen mit unterschiedlichen Zügen gesammelt werden, bevor er festlegt, welcher Zug in einer bestimmten Situation der beste ist. Es werden zwei verschiedene Strategien für die Generation derartiger Erfahrungen ausprobiert. Zudem wird ausprobiert, ob der Algorithmus besser lernt, wenn er gegen sich selbst spielt oder wenn er gegen einen nicht-lernenden Gegenspieler spielt. Die experimentelle Überprüfung der verschiedenen Versionen dieses Algorithmus ergibt, dass er auch nach einer Million Lernpartien nur eine sehr niedrige Spielstärke erreicht. Nur mit Mühe kann nachgewiesen werden, dass der Algorithmus überhaupt etwas gelernt hat.

Auf Grundlage der mit dem „tabularen“ Spieler gewonnen Ergebnisse wird der Algorithmus weiterentwickelt, indem die Tabelle durch ein neuronales Netzwerk ersetzt wird. Anders als die Tabelle kann ein Netzwerk auch Züge bewerten, für die bislang keine Erfahrungen vorliegen. Ein erster Ansatz sieht vor, das Netzwerk mit den aus dem Training des tabularen Algorithmus bereits vorliegenden Stellungsbewertungen zu trainieren. Dafür werden Methoden des überwachten maschinellen Lernens verwendet. Dieses Vorgehen wird mit verschiedenen Netzwerkarchitekturen programmiertechnisch implementiert und experimentell überprüft (Abschnitt

5). Dabei zeigt sich, dass die Nutzung eines Netzwerks zu einer erheblich höheren Spielstärke führt. Zudem ergibt die Überprüfung, dass eine komplexere Netzwerkarchitektur nicht durchweg zu besseren Ergebnissen führt.

Als zweiter Ansatz für einen netzwerkbasierten RL-Reversi-Spieler wird ein Algorithmus konstruiert, dessen Netzwerk kontinuierlich an neue Beobachtungen angepasst wird (Abschnitt 6). Dafür wechseln sich über viele Zyklen Erfahrungssammlung durch neue Lernpartien, Training der Netzwerkgewichte und Tests der erreichten Spielstärke ab. Auch dieser Algorithmus wird programmiertechnisch implementiert. In der experimentellen Überprüfung gelingt es allerdings nicht, die mit dem einfacheren ersten Ansatz erreichte Spielstärke zu überbieten.

Im abschließenden Abschnitt 7 werden die Ergebnisse der verschiedenen Verfahren nochmals verglichen.

Der Quelltext der für diese Arbeit entwickelten Programme kann auf github.com/ThorstenClausing/reversi eingesehen werden. Dort sind auch die Gewichte verfügbar, die bei den in den Abschnitten 5 und 6 beschriebenen Trainingsprozessen generiert wurden. Die Tabellendaten aus Abschnitt 4 sind auf https://fernuni-hagen.sciebo.de/s/cnTZNJmx4zazJnj verfügbar.

2 Reinforcement Learning

Reinforcement Learning (RL) ist neben überwachtem und unüberwachtem Lernen einer der Teilbereiche des maschinellen Lernens. Gelernt wird dabei die beste Aktionswahl in einer sich wiederholenden Entscheidungssituation. Einerseits verändert sich die Entscheidungssituation, insbesondere auch aufgrund der in der Vergangenheit ausgewählten Aktionen, von Entscheidungszeitpunkt zu Entscheidungszeitpunkt, andererseits bleiben aber die grundsätzlichen Zusammenhänge zwischen Aktionswahl und Situationsänderungen gleich. Diese Zusammenhänge sind dem Lerner anfangs unbekannt. Er versucht, sie aus den Erfahrungen abzuleiten, die er mit dem Ausprobieren verschiedener Aktionen macht, und lernt dabei, welche Aktion in welcher Situation am besten ist (vergleiche beispielsweise [23], S. 1).

Diese Art des Lernens erscheint sehr attraktiv, wenn man einem Computer ein Spiel beibringen will. Statt für jede mögliche Spielsituation zu definieren, was jeweils der beste Spielzug ist, reicht es nach diesem Ansatz aus, die Spielregeln zu definieren und den Computer mit einem RL-Algorithmus auszustatten. Die Regeln legen fest, welches in jeder Spielsituation die zulässigen Spielzüge sind und wann das Spiel vorbei ist und wer dann gewonnen hat. Welcher der zulässigen Züge der beste ist, soll der Computer dann durch RL selber ermitteln.

Im folgenden Abschnitt wird eine formalere allgemeine Darstellung von RL und RL-Algorithmen gegeben, die dann auf den speziellen Fall von Brettspielen angewendet wird.

2.1 Grundmodell

In der RL-Literatur wird der Lerner in der Regel als *Agent* bezeichnet, und die Problemsituation, für die er das beste Verhalten erlernen soll, als *Umgebung*. Die Umgebung kann als Markov-Entscheidungsproblem (MEP) modelliert werden. Ich folge in diesem Abschnitt der entsprechenden Darstellung in [23], S. 47ff.

Der *Zustand* der Umgebung zum Zeitpunkt t wird durch eine Zustandsvariable $S_t \in \mathcal{S}$ modelliert. Der Agent wählt zum Zeitpunkt t eine *Aktion* $A_t \in \mathcal{A}$. Die Zustandsvariable S_{t+1} hängt stochastisch von S_t und der gewählten Aktion A_t ab. Zudem erhält der Agent zum Zeitpunkt $t+1$ eine *Belohnung* $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$, die ebenfalls stochastisch von S_t und A_t abhängt. Die stochastischen Abhängigkeiten werden durch eine Transitionsfunktion $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ beschrieben, wobei

$$p(s', r|s, a)$$

die bedingte Wahrscheinlichkeit dafür angibt, dass $S_{t+1} = s'$ und $R_{t+1} = r$ gelten, wenn $S_t = s$ und $A_t = a$ gelten. Bei dieser Modellierung sind die Wahrscheinlichkeiten von t unabhängig, sie ändern sich also im Zeitablauf nicht. Es gilt $\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1$ für alle $s \in \mathcal{S}, a \in \mathcal{A}$.

Vorerst sei nur der Fall endlicher Mengen $\mathcal{A}, \mathcal{S}$ und $\mathcal{R}$ und diskreter Zeit $t \in \mathbb{N}_0$ betrachtet. Dabei mag es Zustände $s \in \mathcal{S}$ geben, in denen keine Aktionen und keine weiteren Zustandsübergänge möglich sind (= finale Zustände).

Unterstellt ist, dass der Agent nicht nur an der Belohnung zum jeweils nächsten Zeitpunkt interessiert ist, sondern an allen zukünftigen Belohnungen. Da dies unendlich viele Belohnungen sein können, wird als Zielvariable G_t des Agenten die diskontierte Summe aller zukünftigen Belohnungen verwendet

$$G_t = \sum_{i=0}^{\infty} \gamma^i R_{t+1+i} \tag{1}$$

$\gamma \in [0, 1]$ ist dabei ein Diskontfaktor, der zum einen technisch gesehen auch bei einer unendlichen Summe in (1) bei begrenztem $\mathcal{R}$ Konvergenz garantiert und zum anderen inhaltlich gesehen eine Präferenz des Agenten für Belohnungen in der näheren Zukunft gegenüber Belohnungen in der weiter entfernten Zukunft ausdrückt. Je kleiner γ, desto größer ist diese Präferenz.

Unterstellt ist ferner, dass der Agent die Zustandsvariable S_t beobachten kann. Daher kann er seine Aktionswahl von ihr abhängig machen. Ein vollständiger Handlungsplan des Agenten könnte folglich als eine Funktion aufgefasst werden, die jedem möglichen Zustand eine Aktion zuordnet. Dabei sei zugelassen, dass die Aktionswahl stochastisch erfolgt. Eine *Politik* des Agenten ist somit eine Funktion $\pi : \mathcal{A} \times \mathcal{S} \to [0, 1]$, die jedem Zustand eine Wahrscheinlichkeitsverteilung über der Menge der Aktionen zuordnet. $\pi(a|s)$ steht folglich für die Wahrscheinlichkeit, mit der der Agent in Zustand $S_t = s$ die Aktion $a \in \mathcal{A}$ ausführt. Es gilt $\sum_{a \in \mathcal{A}} \pi(a|s) = 1$ für alle $s \in \mathcal{S}$.

2.2 Lösungsalgorithmen

Es gibt eine große Anzahl unterschiedlicher Ansätze, wie in dem im vorigen Abschnitt beschriebenen Rahmen die optimale Aktionsauswahl des Agenten bestimmt werden kann. Der Darstellung in [15] folgend, können diese in wertbasierte, politikbasierte und modellbasierte Algorithmen eingeteilt werden.

Der erste Ansatz ordnet jedem Zustand oder jedem Zustands-Aktions-Paar einen *Wert* zu. Dieser Wert entspricht dem Erwartungswert der kumulierten diskontierten zukünftigen Belohnungen, der sich für den Agenten unter einer gegebenen Politik ausgehend vom jeweiligen Zustand beziehungsweise ausgehend vom jeweiligen Zustand bei Wahl der jeweiligen Aktion ergibt. In der Notation von [23] (S. 58ff) bezeichne $v_\pi(s)$ den Wert des Zustands $s \in \mathcal{S}$ unter der Politik π. Es gilt

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi[\sum_{i=0} \gamma^i R_{t+1+i} | S_t = s] \\ &= \sum_{a \in \mathcal{A}} \pi(a|s) \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r, s, a)[r + \gamma v_\pi(s')] \end{aligned}$$

Unter einer optimalen Politik π^* – also einer Politik, die $v_\pi(s)$ für alle Zustände maximiert – gilt für den Wert aller Zustände $s \in \mathcal{S}$

$$v_*(s) = \max_\pi v_\pi(s) = \max_{a \in \mathcal{A}} \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r, s, a)[r + \gamma v_*(s')] \quad (2)$$

Alternativ lässt sich auch jedem Paar aus einer Aktion $a \in \mathcal{A}$ und einem Zustand $s \in \mathcal{S}$ ein Wert $q_\pi(s, a)$ zuordnen. Unter einer optimalen Politik gilt dann für alle Paare

$$q_*(s, a) = \sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r, s, a)[r + \gamma \max_{a' \in \mathcal{A}} q_*(s', a')] \quad (3)$$

Die Gleichungen (2) beziehungsweise (3) werden Bellmann-Gleichungen genannt. Sie bilden jeweils ein Gleichungssystem, das im endlichen Fall eine eindeutige Lösung hat. Aus dieser Lösung lässt sich wiederum für jeden Zustand die optimale Aktion und damit eine optimale Politik ableiten. Wenn die Transitionsfunktion p bekannt ist und $\mathcal{A}$ und $\mathcal{S}$ nicht zu groß sind, lässt sich ein MEP auf diese Weise analytisch lösen (siehe [23], S. 64).

RL-Algorithmen beschäftigen sich mit Situationen, in denen die Transitionsfunktion nicht bekannt ist, sondern aus Erfahrungswerten abgeschätzt werden muss. Wenn die Anzahl der Zustände beziehungsweise der Zustand-Aktions-Paare klein genug ist, kann für jeden einzelnen Zustand oder jedes einzelne Paar eine separate, aus den beobachteten Werten hergeleitete Abschätzung von $v_*(s)$ oder $q_*(s, a)$ (gewissermaßen in einer Tabelle) gespeichert werden. Ein derartiger Algorithmus wird in [15] als „tabularer" wertbasierter Algorithmus bezeichnet. Ist die Zustandsmenge

für ein solches Vorgehen zu groß, wird ein Algorithmus benötigt, der die Beobachtungen für mehrere Zustände aggregiert verarbeitet. Dies kann insbesondere mit Hilfe eines (tiefen) neuronalen Netzwerks erreicht werden. In diesem Fall spricht man von einem tiefen wertbasierten Algorithmus.

Für „tabulare" wertbasierte Algorithmen lässt sich unter recht allgemeinen Umständen beweisen, dass die mit ihnen hergeleitete Lösung gegen die optimale Lösung konvergiert (siehe [23], S. 78ff). Für tiefe wertbasierte Algorithmen lassen sich keine vergleichbaren Konvergenzergebnisse beweisen (ebenda, S. 254). Auch im „tabularen" Fall ist aber nicht gesagt, dass die Konvergenz zur optimalen Lösung hinreichend schnell erfolgt, um von praktischem Nutzen zu sein.

Bei wertbasierten Algorithmen besteht die Gefahr, dass einer Aktion in einem Zustand aufgrund eines am Anfang des Lernprozess zufällig beobachteten hohen Belohnung ein hoher Wert zugeordnet wird, wodurch alle alternativen Aktionen unattraktiv erscheinen und auch zukünftige nicht gewählt werden, selbst wenn sie tatsächlich im Durchschnitt zu (noch) höheren Belohnungen führen könnten. Es ist daher notwendig, nicht immer die nach aktuellem Kenntnisstand beste Aktion zu wählen, sondern auch hin und wieder Aktionen zu wählen, mit denen noch wenig Erfahrungen vorliegen. Dies wird als Zielkonflikt zwischen Exploitation (= Ausnutzung des bereits erworbenen Wissens) und Exploration (= Ausprobieren bislang selten oder noch gar nicht beobachteter Aktionen zum Erwerb weiteren Wissens) bezeichnet, siehe [23], S. 3.

Eine einfache Strategie, mit diesem Problem umzugehen, besteht darin, bei jeder Aktionswahl mit einer geringen Wahrscheinlichkeit ϵ eine zufällige Aktion zu wählen und entsprechend (nur) mit Wahrscheinlichkeit $1 - \epsilon$ die aktuell am besten bewertete. Dieses Vorgehen wird ϵ-*greedy*-Strategie genannt und lässt sich grundsätzlich in jeden (nicht nur wertbasierten) Algorithmus einbauen ([15], S. 52).

Eine andere einfache Möglichkeit, den verfrühten Ausschluss von Aktionen zu vermeiden, besteht darin, jede Aktion mit umso größerer Wahrscheinlichkeit zu wählen, je besser sie aktuell bewertet wird, dabei aber alle möglichen Aktionen mit positiver Wahrscheinlichkeit zu wählen. In Ermangelung eines bessere Ausdrucks werde ich ein solches Verfahren als σ-*greedy*-Strategie bezeichnen, da die Aktionswahl mit positiver (Standard-)Abweichung σ um die aktuell am besten bewertete Aktion streut.

Politikbasierte Algorithmen sind für RL-Probleme mit sehr großen, potentiell stetigen Aktionsmengen entwickelt worden. Ihnen liegt der Begriff einer *parametrisierten Politik* zugrunde, die dem Agenten direkt, also ohne Rückgriff auf Zustandswerte, die zu wählende Aktion vorgibt. Anders, als es die Unterteilung in wertbasierte und politikbasierte Algorithmen vermuten lässt, verwenden die meisten politikbasierten Algorithmen gleichwohl eine Wertfunktion, um die Politikparameter zu lernen, aber eben nicht, um die beste Aktion abzuleiten. Mit der Notation aus [23] bezeichne $\pi(a|s, \theta)$ die Wahrscheinlichkeit, mit der die Politik π die Aktion $A_t = a$ vorgibt, wenn der Agent Zustand $S_t = s$ beobachtet und der Politikparameter $\theta_t = \theta$ ist. Unterstellt wird, dass π nach θ ableitbar ist.

Der Grundablauf politikbasierter Algorithmen besteht darin, zunächst den Politikparameter θ beliebig zu initialisieren und damit eine Episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ zu generieren. Wenn die Episode „erfolgreich“ erscheint, wird θ so angepasst, dass sich die Wahrscheinlichkeit der beobachteten Episode erhöht. Andernfalls wird θ in die entgegengesetzte Richtung angepasst. Dies wird so lange mit neu generierten Episoden fortgesetzt, bis Konvergenz erreicht ist ([15], S. 100).

Bei modellbasierten Algorithmen verfügt der Agent über ein Modell, das den Zusammenhang zwischen Aktionen und Zustandsübergängen sowie Belohnungen abbildet. Er nutzt die beobachteten Belohnungen und Zustandsübergänge, um die Parameter dieses Modells abzuschätzen. Optimale Aktionen werden dann aus dem Modell abgeleitet. Dafür können dann beispielsweise wieder wert- oder politikbasierte RL-Algorithmen genutzt werden (siehe [15], S. 123ff).

2.3 Brettspiele als Anwendungsgebiet für RL

Brettspiele bieten ein naheliegendes Anwendungsfeld für Lernalgorithmen. Im folgenden Abschnitt wird ein knapper Überblick über die bekanntesten einschlägigen Forschungsarbeiten gegeben, danach wird erläutert, wie sich die Terminologie des Grundmodells auf Brettspiele anwenden lässt.

2.3.1 Forschungsarbeiten zu Brettspiel-Lernalgorithmen

Ein frühes Beispiel für Brettspiel-Lernalgorithmen sind die Dame-Programme, die in den 1950er Jahren von Arthur Samuel entwickelt wurden. Wie in [23], S. 426ff, dargestellt, berechneten diese Programme Werte für Stellungen unter anderem auf Grundlage der Differenz der Anzahl der Spielsteine beider Spieler, die sich bei der wahrscheinlichsten Fortsetzung des Spiels aus der zu bewertenden Stellung heraus ergeben wird. Diese Werte wurden dann im Lernprozess angepasst. Die Programme erreichten die Spielstärke überdurchschnittlich guter Amateure.

In den 1990er Jahren hat Gerald Tesauro das Backgammon-Programm TD-Gammon entwickelt (siehe [23], S. 421ff). Das Programm nutzt ein neuronales Netzwerk, um einer eingegebenen Stellung einen Wert zuzuordnen. Wenn es am Zug ist, berechnet es so die Werte aller mit den aktuell gewürfelten Zahlen erreichbaren Stellungen und wählt den Zug, der zu der am besten bewerteten dieser Stellungen führt. Zum Training des Netzwerks spielt das Programm so lange gegen sich selbst und passt die Netzwerkparameter an, bis die ausgegebenen Werte für eine Stellung in etwa der jeweiligen Gewinnwahrscheinlichkeit entsprechen. Dieses Programm erreichte annähernd die Spielstärke menschlicher Großmeister.

Ein weiteres Beispiel bieten die bei DeepMind entwickelten Go-Programme AlphaGo und AlphaGo Zero (siehe [23], S. 441ff). AlphaGo nutzt überwachtes Lernen, um auf der Grundlage einer Sammlung von Partien menschlicher Meister ein neuronales Netzwerk zu trainieren, das für eine eingegebene Stellung den mutmaßlich besten Zug ermittelt. Dieses Netzwerk wird dann mit RL durch Spiele gegen sich

selbst (beziehungsweise ältere Versionen von sich selbst) weiter optimiert. AlphaGo Zero beruht hingegen ausschließlich auf RL (siehe [22]). AlphaGo konnte einen menschlichen Go-Weltmeister besiegen, verliert aber seinerseits regelmäßig gegen AlphaGo Zero.

In [21] wird AlphaGo Zero weiterentwickelt zu AlphaZero, einem Algorithmus, der nicht mehr auf ein einzelnes Brettspiel spezialisiert ist, sondern neben Go auch Schach, Shogi und verschiedene Videospiele gelernt hat. Der in [20] vorgestellte Algorithmus MuZero geht wiederum einen Schritt weiter, indem er alle diese Spiele ohne Kenntnis der Spielregeln erlernt. Er „weiß“ lediglich, was der Raum grundsätzlich möglicher Aktionen (= die Menge der stellungsunabhängig möglichen Züge) ist. Weitere Verfeinerungen von RL-basierten Verfahren nach dem Vorbild von AlphaGo Zero, AlphaZero und MuZero bleiben ein aktuelles Forschungsthema. Ein im Kontext dieser Masterarbeit besonders relevanter Beitrag ist [13], in dem eine Variante von AlphaGo Zero vorgestellt wird, die mit wesentlich geringerem Rechenaufwand auskommt als das Original und statt auf Go auf Reversi spezialisiert ist.

Zudem werden auch maschinelle Lernverfahren ohne RL-Komponente in aktuellen Forschungsarbeiten auf Brettspiele angewendet. So wird in [17] ein neuronales Netzwerk mit der aus generativen großen Sprachmodellen bekannte Transformer-Architektur verwendet, um Schachzüge zu bewerten und entsprechend den besten Zug auszuwählen. Eine Netzvariante mit 270 Millionen Parametern erreicht dabei eine Spielstärke auf Großmeisterniveau. In [2] wird ein generatives großes Sprachmodell in einen Algorithmus für das Spiel Diplomacy eingebaut. Dieses Spiel zeichnet sich dadurch aus, dass zwischen den eigentlichen Zügen informelle Verhandlungen zwischen einzelnen Spielern stattfinden, bei denen unverbindliche Absprachen über koordinierte Züge getroffen werden können. In diesen Verhandlungsphasen wird das Sprachmodell eingesetzt. Der Algorithmus erreicht eine mit der von menschlichen Spielern vergleichbare Spielstärke. Ein generatives großes Sprachmodell, das auch die Eingabe von Bildern erlaubt, wird in [4] auf Reversi angewendet. Es wird allerdings nicht genutzt, um optimale Züge zu ermitteln, sondern um Züge aus zum Training verwendeten Partien zu reproduzieren und zulässige Züge zu finden.

2.3.2 Brettspiele im Grundmodell

Wenn man die Begriffe des Grundmodells aus Abschnitt 2.1 auf den Fall eines Brettspiels überträgt, so ist der Agent der Spieler, und sein(e) Gegenspieler ist (sind) Teil der Umgebung. Der Zustand der Umgebung ist die Stellung der Spielsteine auf dem Spielbrett. Wenn es in dem Spiel wie bei Backgammon Zufallselemente gibt, so sind diese ebenfalls Teil der Umgebung. So sind bei Backgammon die vom Spieler vor dem Setzen seiner Spielsteine erwürfelten Augenzahlen Teil des Zustands der Umgebung. Die dem Spieler zur Verfügung stehenden Aktionen sind die in der gegebenen Spielstellung zulässigen Züge. Bei manchen Spielen gibt es auch Teile des Zustands, die dem am Zug befindlichen Spieler nicht bekannt sind. So gibt es beim

Spiel Scotland Yard einen Spieler, der die Position seines Spielsteins auf dem Brett geheim aufschreibt und nur nach jeweils fünf Zügen bekannt gibt. In diesem Fall ist der tatsächliche Zustand der Umgebung für die anderen Spieler anders als im Grundmodell vorgesehen nur teilweise beobachtbar. Hier spricht man von Spielen mit unvollkommener Information (siehe z. B. [8], S. 107). Bei Spielen mit vollkommener Information wie Dame, Backgammon, Go oder Reversi kennt der Spieler den Zustand jedoch vollständig.

Auf den ersten Blick könnte man meinen, dass der Spieler in einem Spiel ohne Zufallselement mit seinem Zug die Stellung auf dem Spielbrett und damit auch den Zustand der Umgebung in deterministischer Weise verändert. Dadurch, dass der (die) Gegenspieler ebenfalls zieht (ziehen), bevor der Spieler wieder am Zug ist, ist die Veränderung des Zustands zwischen den Entscheidungszeitpunkten des Spielers jedoch nicht ausschließlich durch seine vorhergehende Entscheidung determiniert, sondern aus seiner Sicht (in der Regel) nicht mit Sicherheit vorhersehbar, also stochastisch. Der deterministische Zusammenhang zwischen dem Zug eines Spielers und der sich durch diesen Zug ergebenden Stellung auf dem Spielbrett erlaubt es jedoch, statt der möglichen Züge selbst alternativ die mit einem möglichen Zug erreichbaren Stellungen als das Auswahlobjekt, also die Aktion des Spielers aufzufassen.

Die Belohnung, die der Spieler nach seinem Zug erhält, ist Null, solange das Spiel nicht beendet ist. Anders gesagt erhält der Spieler erst am Ende des Spiels – also wenn ein finaler Zustand erreicht ist, in dem keine weiteren Züge möglich sind – eine Belohnung. Diese kann durch eine Auszahlung von 1 bei einem Sieg, 0 bei einem Unentschieden und -1 bei einer Niederlage modelliert werden. Einige Spiele erlauben jedoch auch eine Aussage darüber, wie hoch ein Spieler gewonnen hat. Die Höhe eines Sieges beziehungsweise einer Niederlage kann dann als Belohnung aufgefasst werden.

Bemerkenswert erscheint, dass die Belohnung des Gegenspielers in der RL-Modellierung eines Spiels irrelevant ist. Es macht insbesondere keinen Unterschied, ob die Spieler nur gemeinsam gewinnen können oder ob ein Spieler (eine Gruppe von Spielern) genau dann gewinnt, wenn sein Gegenspieler (die Gruppe aller Gegenspieler) verliert. Im Folgenden werden allerdings nur Zwei-Personen-Spiele betrachtet, bei denen der eine Spieler genau dann (und gegebenenfalls genau so hoch) gewinnt, wenn (und wie) der andere verliert. In der Spieltheorie werden diese Spiele als Nullsummenspiele bezeichnet (siehe z. B. [8], S. 13).

Festzuhalten ist zudem, dass die in Abschnitt 2.2 dargestellten Lösungsalgorithmen darauf beruhen, dass der Agent immer wieder dem gleichen MEP gegenüber steht. Aus der Modellierung des Gegenspielers als Teil der Umgebung ergibt sich folglich die Einschränkung, dass der Gegenspieler immer gleich spielt. Das bedeutet natürlich nicht, dass der Gegenspieler immer die gleichen Züge wählt, da er seine Züge in jeder Spielsituation zufällig auswählen kann und dabei beispielsweise auch unterschiedliche Spielstile stochastisch mischen kann. Die Auswahlwahrscheinlichkeiten bleiben dabei aber unverändert. Anders als der Agent kann der Gegenspieler

also nicht dazulernen. Zudem lernt der Agent genau genommen auch nur, gegen einen bestimmten Gegenspieler gut zu spielen. Die Erwartung an einen erfolgreich trainierten Spieler wäre hingegen wohl, dass er ganz allgemein, also gegen einen beliebigen Gegenspieler, erfolgreich spielt.

3 Das Spiel Reversi

Das Spiel Reversi ist auch unter dem Namen Othello bekannt und wird in verschiedenen Varianten gespielt. Ich folge hier den Spielregeln der World Othello Federation (WOF), siehe [26], die im folgenden Abschnitt dargestellt werden.

3.1 Spielregeln

Reversi wird von zwei Spielern auf einem quadratischen Spielbrett mit 64 Feldern gespielt. Es gibt 64 scheibenförmige Spielsteine, die jeweils eine schwarze und eine weiße Seite haben. Die Spieler legen abwechselnd jeweils einen Spielstein auf ein noch leeres Feld. Der Spieler, der als erster zieht, legt die Steine mit der schwarzen Seite nach oben, der andere mit der weißen. Einen Spielstein, der mit der weißen Seite nach oben auf dem Spielbrett liegt, werde ich als weißen Stein bezeichnen, einen Spielstein, der mit der schwarzen Seite nach oben liegt, dementsprechend als schwarzen. Zu Spielbeginn liegen bereits vier Steine auf den vier zentralen Feldern des Spielbretts, und zwar auf dem linken unteren und dem rechten oberen der vier Felder jeweils ein schwarzer Stein und auf dem linken oberen und dem rechten unteren der vier Felder jeweils ein weißer Stein. Diese Grundstellung ist in Abb. 1 dargestellt.

Die Spieler ziehen abwechselnd. Ein Stein darf nur so gelegt werden, dass dadurch mindestens ein Stein der anderen Farbe eingeschlossen wird. Steine einer Farbe sind eingeschlossen, wenn sie in einer horizontalen, vertikalen oder diagonalen Reihe liegen und auf beiden Seiten ein Stein der anderen Farbe neben ihnen liegt. Dabei darf es keine leeren Felder zwischen den beteiligten Steinen geben. Alle Steine, die durch einen neu gelegten Stein eingeschlossen werden, werden umgedreht und wechseln damit die Farbe. Steine, die nach dem Umdrehen der eingeschlossenen Steine durch einen der umgedrehten Stein eingeschlossen werden, bleiben unverändert liegen.

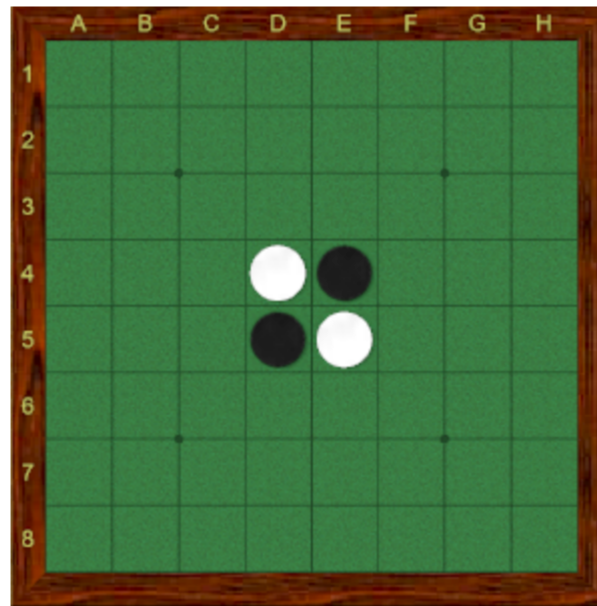

Abbildung 1: Die Grundstellung, mit der jede Reversi-Partie beginnt (Quelle: [26])

Zur Illustration sei die in Abb. 2 dargestellte Stellung betrachtet. Der Spieler mit den weißen Steinen ist am Zug. Wenn er nun einen Stein auf das Feld C6 legt, werden dadurch aufgrund des weißen Steines auf C3 die schwarzen Steine auf den Feldern C5 und C4 vertikal und aufgrund des weißen Steines auf F3 die schwarzen Steine auf D5 und E4 diagonal eingeschlossen. Alle genannten schwarzen Steine müssen dann also umgedreht werden. Alternativ könnte der Spieler mit den weißen Steinen auch einen Stein auf B4, B5, B6, D3, D6, E2, F2 oder F4 legen, er hat hier also insgesamt neun mögliche Züge.

Wenn der Spieler, der am Zug ist, keinen zulässigen Zug hat, muss er passen, und sein Gegenspieler darf erneut ziehen. Das Spiel endet, wenn kein Spieler mehr ziehen kann. Dies ist typischerweise der Fall, wenn alle 64 Felder belegt sind, kann aber auch schon früher passieren. Gewonnen hat der Spieler, dessen Farbe die meisten Steine auf dem Spielbrett haben. Bei gleich vielen weißen und schwarzen Steinen ist das Spiel unentschieden.

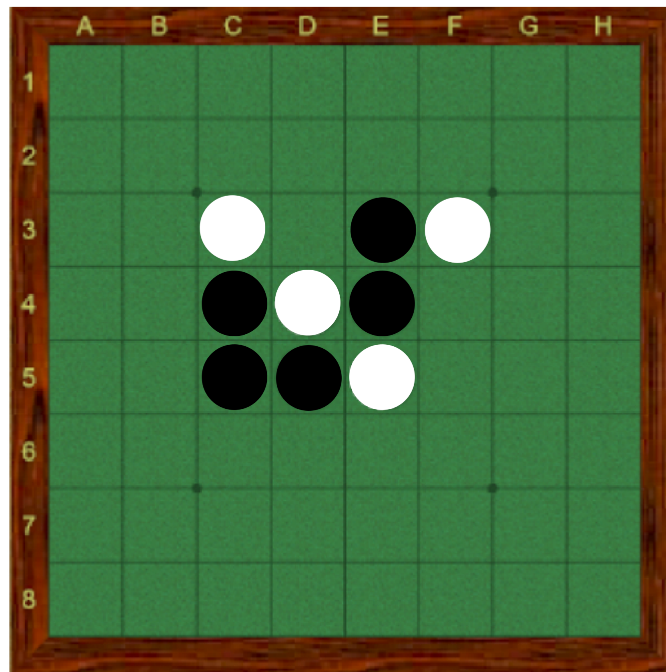

Abbildung 2: Eine Reversi-Stellung nach fünf Zügen

Aus der Anzahl der am Spielende auf dem Spielbrett liegenden Steine der beiden Farben lässt sich für Turnierwertungen oder ähnliche Zwecke leicht ein Spielergebnis ableiten: Der Verlierer erhält so viele Punkte, wie er Steine hat, der Gewinner erhält 64 Punkte minus die Punkte des Gegenspielers. Bei einem Unentschieden erhalten beide 32 Punkte (siehe [25], S. 11).

3.2 Strategische Eigenschaften von Reversi

Aus den Spielregeln von Reversi ergeben sich einige besondere Eigenschaften, die bei der Konstruktion eines Algorithmus für dieses Spiel zu berücksichtigen sind.

Reversi bietet eine hohe Anzahl möglicher Stellungen. Geschätzt wird, dass es bis zu 10^{58} mögliche Spielverläufe und bis zu 10^{28} mögliche Stellungen gibt ([1], S. 167). Offensichtlich können in jedem Spielverlauf höchstens 60 Steine gesetzt werden. Da es Stellungen gibt, in denen ein Spieler passen muss, und ein solches Passen auch als Zug zu zählen ist, kann eine Partie mehr als 60 Züge umfassen. Da eine Partie beendet ist, wenn beide Spieler unmittelbar nacheinander passen müssen, ergibt sich eine theoretische Höchstlänge von 120 Zügen. Da aber im ersten, zweiten, dritten, vierten, ... Zug nie gepasst werden muss, ist die tatsächliche Höchstlänge deutlich niedriger. (Ich zähle hier jeden einzelnen Zug eines Spielers als einen ganzen Zug und nicht als einen Halbzug, wie es beispielsweise im Schach üblich ist, wo ein ganzer Zug dementsprechend aus einem Halbzug von Weiß und einem Halbzug von Schwarz besteht.) Um eine Vorstellung zu haben, welche durchschnittlichen und

extremen Partielängen zu erwarten sind, habe ich 4 Millionen Partien mit zufälliger Zugauswahl simuliert. Die kürzestes Partie war 10 Züge lang, die längste 70. Die Durchschnittslänge betrug 60,42 Züge. Gepasst wurde frühestens im neunten Zug.

Grundsätzlich kann ein und dieselbe Stellung über verschiedene Zugfolgen erreicht werden. Für den weiteren Spielverlauf ist die Zugfolge irrelevant, und damit auch für die Einschätzung, welcher der Spieler in dieser Stellung besser steht, also größere Gewinnchancen hat.

Aufgrund der Passen-Regel kann man einer Stellung nicht notwendigerweise leicht ansehen, welcher Spieler am Zug ist. Theoretisch kann es Stellungen geben, die über Zugfolgen mit und ohne Passen erreicht werden können, so dass bei gleicher Verteilung der Steine auf dem Spielbrett einmal schwarz und einmal weiß am Zug ist. Solche Stellungen sind als zwei verschiedenen Stellungen zu betrachten.

Das Spielfeld hat vier Symmetrieachsen. Da die Spielregeln nicht zwischen oben und unten oder rechts und links unterscheiden, macht es für die möglichen Fortsetzungen und die Gewinnaussichten der beiden Spieler keinen Unterschied, wenn eine Stellung an einer dieser Achsen gespiegelt wird. Ist ein Zug in einer Stellung optimal, so ist nach der Spiegelung der entsprechend gespiegelte Zug optimal. Das Gleiche gilt, wenn eine Stellung um 90, 180 oder 270 Grad gedreht wird. Durch Spiegelung oder Drehung auseinander hervorgegangene Stellungen bezeichne ich im Folgenden als „strategisch äquivalent".

Eine naheliegende Idee zur Bewertung einer Stellung besteht darin, sie nach der Differenz in der Anzahl der Spielsteine zu bewerten. Für die Endstellung ist dies offensichtlich korrekt, für frühere Stellungen wäre dies jedoch irreführend, wie in [19], S. 46, angemerkt wird:

> Although the goal is to finish with the most pieces of your color up, the best strategy, paradoxically, is usually to limit your opponent's options by flipping over as few of his discs as possible during the first two-thirds of the game.

Ein Ergebnis eines Nullsummenspiels, von dem beide Spieler durch geeignete Zugwahl sicherstellen können, dass das Spiel für sie nie mit einem schlechteren als diesem Ergebnis ausgehen wird, bezeichnet man als den spieltheoretischen Wert eines Spiels. Dabei wird der Wert aus Sicht des zuerst ziehenden Spielers angegeben, für den als Zweiten ziehenden Spieler ist er entsprechend mit -1 zu multiplizieren. Jedes endliche Zwei-Personen-Nullsummenspiel mit vollkommener Information hat einen eindeutigen Wert (siehe z. B. [8], S. 103). In [24] wird ein Beweis dafür präsentiert, dass der Wert von Reversi, ausgedrückt als die Differenz der von Schwarz und Weiß erhaltenen Punkte, Null ist. Bei bestem Spiel beider Spieler sollte also jede Partie unentschieden enden.

3.3 Reversi-Programme

Es gibt zahlreiche Computerprogramme für Reversi. Diese beruhen in der Regel auf einem Baumsuchalgorithmus, der von der aktuellen Stellung aus mögliche Zugfol-

gen betrachtet und die aus ihnen resultierenden zukünftigen Stellungen bewertet (siehe z. B. [1]). Für endliche Zwei-Personen-Nullsummenspiele ist der *Minimax-Algorithmus* die Grundform eines derartigen Baumsuchalgorithmus. Alle von der aktuellen Situation aus möglichen Spielverläufe werden bis zum Spielende als Baum aufgezeichnet, wobei die Kanten Züge sind und der Knoten am Ende der Kante für die durch diesen Zug entstandene Stellung steht. Von diesem Knoten gehen wieder Kanten für alle in dieser Stellung möglichen Züge des Gegenspielers aus. Endstellungen (= Endknoten) werden mit dem Spielergebnis aus Sicht des Anziehenden bewertet. Knoten, von denen aus nur Endknoten erreicht werden können und an denen der Anziehende am Zug ist, werden mit dem maximalen Wert dieser Endknoten bewertet. Knoten, von denen aus nur Endknoten erreicht werden können und an denen der Nachziehende am Zug ist, werden mit dem minimalen Wert dieser Endknoten bewertet. Entsprechend wird ein Knoten, dessen direkte Nachfolger bereits alle bewertet sind, mit dem maximalen Wert der Nachfolger bewertet, wenn der Anziehende an diesem Knoten am Zug ist, und mit dem minimalen Wert der Nachfolger, wenn der Nachziehende am Zug ist. Geht der Suchbaum von der Grundstellung aus, entspricht der Wert des Wurzelknotens dem am Ende von Abschnitt 3.2 erwähnten Wert des Spiels. Am aktuellen Knoten wählt der Anziehende den Zug, der zu dem am höchsten bewerteten Folgeknoten führt, oder der Nachziehende den Zug, der zu dem am niedrigsten bewerteten Folgeknoten führt (vergleiche z. B. [18], S. 195ff).

Bei den meisten Brettspielen und insbesondere bei Reversi ist der für dieses Verfahren aufzustellende Suchbaum viel zu groß, um ihn rechentechnisch handhaben zu können, falls nicht bereits eine Stellung kurz vor Partieende erreicht wurde. Deshalb wurden eine große Anzahl von Abwandlungen des Algorithmus entwickelt, die den Rechenaufwand verringern. Eine davon ist die *α-β-Suche*. Betrachtet sei dafür ein Knoten k, an dem der Anziehende (= der maximierende Spieler) am Zug ist. Sei β das beste Ergebnis, dass der Nachziehende (= der minimierende Spieler) in einem bereits untersuchten, aus Minimax-Zügen bestehenden Spielverlauf außerhalb des bei k beginnenden Teilspielbaums erreichen kann. Wenn der Anziehende nun in diesem Teilspielbaum einen Minimax-Spielverlauf findet, der ihm ein Ergebnis $\alpha > \beta$ bietet, so kann geschlussfolgert werden, dass der Nachziehende diesen Teilspielbaum meiden, also an einem der Vorgängerknoten von k einen anderen Zug als den zu k führenden wählen wird. Daher braucht dieser Teilspielbaum nicht weiter untersucht zu werden. Analoges gilt für Knoten, an denen der Nachziehende am Zug ist. Die α-β-Suche führt zum gleichen Ergebnis wie der Minimax-Algorithmus, jedoch häufig mit erheblich geringerem, d. h. tatsächlich handhabbarem Rechenaufwand (vergleiche z. B. [18], S. 198ff).

Eine weitere Alternative besteht darin, die möglichen Zugfolgen nicht bis zum Spielende, sondern nur bis zu einer bestimmten maximalen Anzahl von Zügen (= Tiefe) zu betrachten. Die Stellungen der Endknoten sind dann keine finalen Stellungen der Partie, so dass man sie nicht mit dem Partieergebnis bewerten kann, sondern irgend eine andere Bewertungsmethode finden muss. Von diesen Bewer-

tungen aus kann man dann analog wie bei der α-β-Suche vorgehen. Ein derartiges Verfahren wird beispielsweise in den Reversi-Programm Edax und Egaroucid, die aktuell als die spielstärksten Reversi-Programme gelten (siehe [5] und [27]) verwendet. Edax wird in [24] als Grundlage für den Beweis, dass Reversi den Wert Null hat, genutzt. Dabei wird auch gezeigt, wie auf Basis von Edax ein Mechanismus kreiert werden kann, der immer optimale Züge entsprechend der Minimax-Lösung spielt, also nie verliert.

In [23], S. 159ff, werden Suchbaumverfahren als Methoden des „Planens" beschrieben und Lernmethoden wir RL gegenübergestellt. Dabei wird jedoch die formale Ähnlichkeit beider Methodenfamilien betont. Diese Ähnlichkeit wird noch größer, wenn bei einem Suchbaumverfahren eine Bewertungsfunktion für Stellungen verwendet wird, die auf in der Vergangenheit von der bewerteten Stellung aus erzielten Spielergebnissen beruht.

4 Geradlinige „tabulare" Implementierung eines RL-Reversi-Spielers

Für eine geradlinige („naive") Implementierung eines „tabularen" wertbasierten RL-Algorithmus, wie er in Abschnitt 2.2 abstrakt dargestellt ist, definiere ich in diesem Abschnitt einen Zugauswahlalgorithmus für einen lernenden Spieler, aus dem sich nach Abschluss des RL-Trainingsprozesses der für den „Produktivbetrieb" gedachte optimierende Spieler konstruieren lässt. Dieser *Lernende Spieler* merkt sich für alle Stellungen, die ihm schon einmal begegnet sind, welchen Zug er gespielt hat und zu welchem Endergebnis dies geführt hat, und wählt dann einen Zug entsprechend den in vorhergehenden Partien mit diesem Zug erreichten Ergebnissen. Aufgrund des Zielkonflikts zwischen Exploration und Exploitation ist die Wahl jedoch in geeigneter Weise randomisiert. Der *Optimierende Spieler* wählt hingegen jeweils deterministisch denjenigen Zug, der sich in vorhergehenden Partien in der aktuellen Stellung als am erfolgreichsten erwiesen hat. Er ist gewissermaßen das Endergebnis des Lernprozesses eines Lernenden Spielers, wenn eine hinreichend hohe Spielstärke erreicht wurde.

4.1 Tabelle und Zugauswahl

Wie die Bezeichnung nahe legt, ist das zentrale Element eines „tabularen" Algorithmus eine Tabelle, in der die mit den verschiedenen beobachteten, d. h. bislang gespielten Zügen gemachten Erfahrungen gespeichert werden. In diesem Abschnitt werden die Eigenschaften dieser Tabelle und der aus den gespeicherten Werten abgeleiteten Zugauswahl beschrieben.

Die Tabelle eines wertbasierten Algorithmus speichert im Allgemeinen für jedes Paar aus möglichem Zustand und in diesem Zustand möglicher Aktion eine Bewertung. Bei einem Brettspiel entspricht dies den Paaren aus möglicher Stellung und in dieser Stellung möglichem Zug. Wie in Abschnitt 2.3.2 bereits dargelegt, kann

ein Zug mit der durch diesen Zug erreichten Stellung identifiziert werden. Wie in Abschnitt 3.2 angemerkt, kann bei Reversi ein und dieselbe Stellung durch verschiedene Zugfolgen erreicht werden. Die Bewertung der erreichten Stellung sollte nicht davon abhängen, über welche Zugfolge sie erreicht wird. Es ist also nicht nötig, in der Wertetabelle für jedes Paar aus einer Ausgangsstellung und einer von ihr aus durch einen möglichen Zug zu erreichenden Stellung eine eigene Zeile vorzusehen, sondern es reicht jeweils eine Zeile mit einer Bewertung für jede erreichbare Stellung. Die Bewertung erfolgt dabei aus Sicht des Spielers, der diese Stellung herbeigeführt hat - also aus Sicht des Spielers, der in der Stellung nicht am Zug ist (siehe [23], S. 137, wo die durch einen Zug erreichbare Stellung als *afterposition* bezeichnet wird; ich werde dafür auch den Begriff Folgestellung verwenden).

Als nächstes ist zu klären, was genau als Bewertung gespeichert werden soll. Wie in Abschnitt 2.3.2 dargelegt, kann die Bewertung entweder nur zwischen Sieg, Niederlage und Unentschieden unterscheiden, oder auch die Höhe des Sieges berücksichtigen. Die Vermutung liegt nahe, dass Letzteres einen höheren Informationsgehalt hat und damit zu einem schnelleren Lernfortschritt führen sollte. Als Bewertung werden daher hier die vom jeweiligen Spieler nach den WOF-Regeln erreichten Punkte verwendet. Bei einem Ergebnis von 34:30 sind dies also für Schwarz 34 Punkte und für Weiß 30. Jedes Mal, wenn eine Stellung in einer Partie auftritt, wird nach der Partie die Anzahl der Punkte, die der Spieler, der in der Stellung nicht am Zug ist, erreicht hat, zu den bisher für diese Stellung gespeicherten Punkten hinzuaddiert. Zugleich wird gespeichert, wie oft die Stellung aufgetreten ist. Die Tabelle hat also drei Spalten: Stellung, Punktsumme und Auftretenshäufigkeit. Der Quotient aus der Punktsumme und der Häufigkeit ergibt dann die aktuelle Bewertung der Stellung.

In Abschnitt 3.2 wurde darauf hingewiesen, dass es strategisch äquivalenten Stellungen gibt. Zwei strategisch äquivalente Stellungen sollten immer gleich bewertet werden. Um dies sicherzustellen, wird aus jeder Gruppe strategisch äquivalenter Stellungen (also aus jeder Äquivalenzklasse) eine bestimmte Stellung ausgewählt und als die „kanonische“ Stellung dieser Gruppe definiert. In der Tabelle werden nur kanonische Stellungen gespeichert, und bei jeder Stellung, die im Trainingsprozess auftritt, wird das beobachtete Ergebnis der zugehörigen kanonischen Stellung zugeschrieben.

Für die Zugauswahl betrachte ich zwei Varianten. In der ϵ-*greedy*-Variante des Lernenden Spielers prüft der Algorithmus bei Eingabe einer aktuellen nicht-finalen Stellung, ob in dieser Stellung mehr als ein Zug möglich ist. Falls nicht, wird der einzige mögliche Zug gewählt oder, falls gar kein Zug möglich ist, gepasst. Falls mehr als ein Zug möglich ist, werden mit Wahrscheinlichkeit ϵ die in der Tabelle vorliegenden Bewertungen möglicher Züge ignoriert und stattdessen mit gleichverteilter Wahrscheinlichkeit ein beliebiger dieser Züge ausgewählt. Mit der Gegenwahrscheinlichkeit $1 - \epsilon$ wird ermittelt, welche Stellungen von der aktuellen Stellung aus in einem Zug erreichbar sind. Wenn zu all diesen Stellungen (in deren kanonischer Variante) noch keine Bewertung gespeichert ist, wird jeder mögliche Zug mit glei-

cher Wahrscheinlichkeit ausgewählt. Wenn zu allen Stellungen bereits Bewertungen gespeichert sind, wird der Zug mit der besten Bewertung gespielt. Sollten mehrere Züge die gleiche beste Bewertung haben, wird zwischen ihnen mit gleichverteilter Wahrscheinlichkeit gewählt. Wenn zu einigen, aber nicht zu allen Stellungen Bewertungen vorliegen, ist für die bislang nicht bewerteten Stellungen eine hypothetische Bewertung („Standardbewertung“) zu verwenden. Im letzten Fall ist zu beachten, dass umso eher einer der noch nicht erprobten Züge gewählt wird, je höher die Standardbewertung ist. Insofern hängt die Explorationsfreude des Algorithmus nicht nur von ϵ ab, sondern auch von der Standardbewertung. Daher wähle ich 40 als Standardbewertung, also einen relativ hohen Wert oberhalb des „neutralen“ Mittelwerts 32.

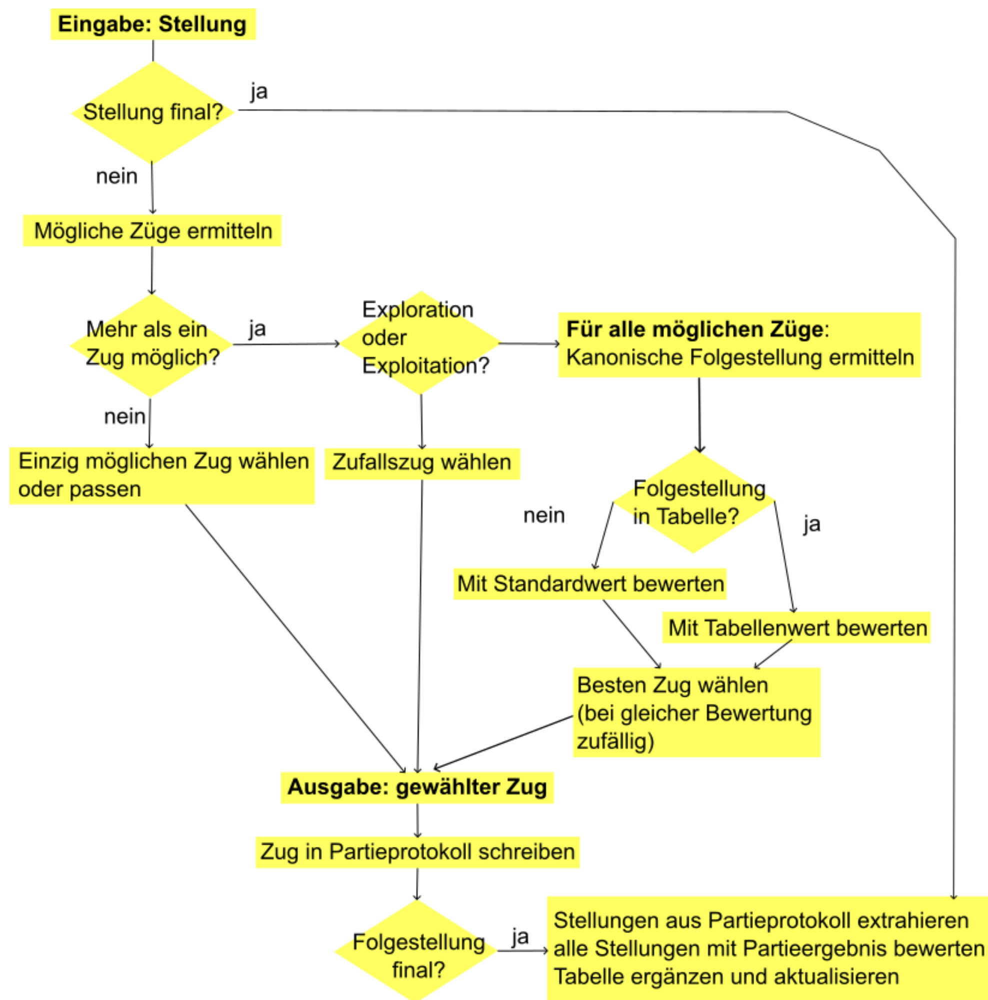

Abbildung 3: ϵ-*greedy*-Variante des Lernenden Spielers

Der gewählte Zug wird in einem Spielprotokoll gespeichert. Wenn eine finale

Stellung erreicht wird, wird zudem das Spielergebnis ermittelt und im Protokoll vermerkt. Daraufhin werden die Bewertungen der in der abgeschlossenen Partie aufgetretenen Stellungen in der Tabelle auf Grundlage der Daten im Protokoll wie oben beschrieben aktualisiert oder ergänzt.

Der Algorithmus für den „tabularen" Lernenden Spieler in der ϵ-*greedy*-Variante wird schematisch in Abb. 3 dargestellt.

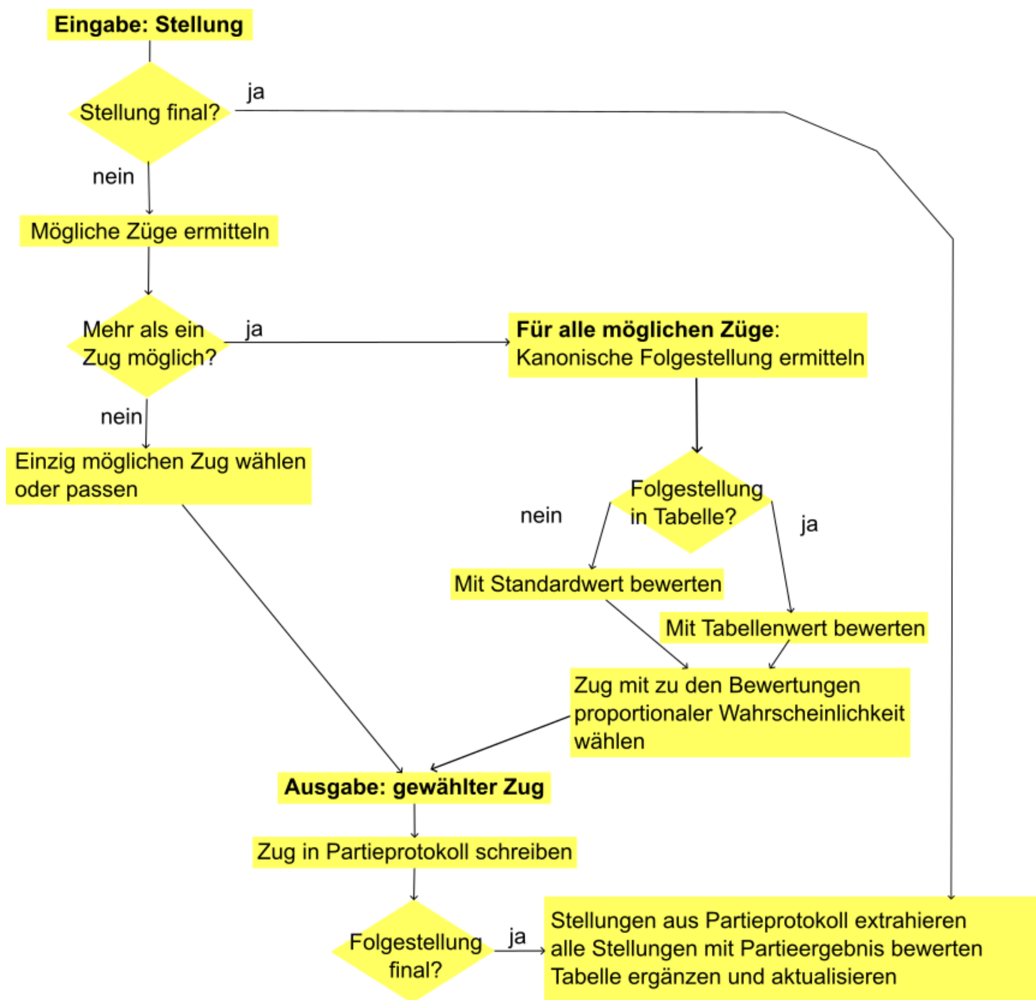

Abbildung 4: σ-*greedy*-Variante des Lernenden Spielers

Bei der σ-*greedy*-Variante des Lernenden Spielers wird ebenfalls bei Eingabe einer aktuellen nicht-finalen Stellung zunächst geprüft, ob in dieser Stellung mehr als ein Zug möglich ist. Falls nicht, wird der einzige mögliche Zug gewählt oder, falls gar kein Zug möglich ist, gepasst. Falls mehr als ein Zug möglich ist, wird ermittelt, welche Folgestellungen von der aktuellen Stellung aus erreichbar sind. Für bereits beobachtete Stellungen wird die Bewertung (ihrer kanonischen Variante) der

Tabelle entnommen, noch nicht beobachteten Stellungen wird ein Standardwert zugewiesen. Nun wird einer der möglichen Züge ausgewählt, wobei jeder Zug mit der Wahrscheinlichkeit gewählt wird, die dem Verhältnis der Bewertung dieses Zuges zur Summe der Bewertungen aller möglichen Züge entspricht. Hat ein Zug also eine doppelt so hohe Bewertung wie ein alternativer, wird er mit doppelt so hoher Wahrscheinlichkeit gewählt.

Auch bei dieser Algorithmusvariante beeinflusst die Standardbewertung die Explorationsfreude. Ich wähle daher auch hier den Wert 40. Zudem ist zu vermeiden, dass eine Stellung mit Null bewertet wird, da der entsprechende Zug dann nie mehr ausprobiert würde. Wenn eine bislang nicht in der Tabelle verzeichnete Stellung in einer Partie auftritt, in der der Spieler, der in dieser Stellung nicht am Zug ist, Null Punkte erhält, wird daher abweichend vom oben geschilderten Vorgehen in der Tabelle nicht eine Bewertung von Null, sondern von zwei verzeichnet.

Wie in der ϵ-*greedy*-Variante wird der gewählte Zug in einem Spielprotokoll gespeichert, auf dessen Grundlage nach Abschluss einer Partie die Bewertungen in der Tabelle aktualisiert oder ergänzt werden.

Der Algorithmus für den „tabularen" Lernenden Spieler in der σ-*greedy*-Variante wird schematisch in Abb. 4 dargestellt.

Der Algorithmus des Optimierenden Spielers prüft in einer nicht-finalen Stellung, in der mehr als ein Zug möglich ist, für welche Folgestellungen Bewertungen vorliegen. Unter diesen wählt er die mit der besten Bewertung aus. Weisen mehrere Folgestellungen die beste Bewertung auf, wird eine von ihnen mit gleichverteilter Wahrscheinlichkeit ausgewählt. Enthält die Tabelle für keine der möglichen Folgestellungen eine Bewertung, wird mit gleichverteilter Wahrscheinlichkeit unter allen möglichen Zügen ausgewählt.

4.2 Gegner und Vergleichsmaßstäbe

Tabelle und Zugauswahl beschreiben die Modellierung des Agenten (= Spielers). Als Nächstes ist die Umgebung zu modellieren. Wie in Abschnitt 2.3.2 dargestellt, ist der Gegenspieler der zentrale Teil der Umgebung. Dabei ist es naheliegend, diesen Teil der Umgebung genau wie den Spieler selbst als Zugauswahlalgorithmus zu implementieren. Grundsätzlich könnte dafür genau der gleiche Algorithmus wie für den Lernenden Spieler zum Einsatz kommen. Allerdings ergibt sich dann das am Ende von Abschnitt 2.3.2 erwähnte Problem, dass der Gegenspieler nicht in allen Trainingspartien gleich spielt. Um den in Abschnitt 2.2 dargestellten Verfahren formal zu entsprechen, bräuchte man also einen „konstant" spielenden Gegenspieler. Ferner sind Spieleralgorithmen mit konstanter Spielstärke auch erforderlich, um einen Vergleichsmaßstab zu haben, an dem man die (hoffentlich im Lernprozess wachsende) Spielstärke des Lernenden Spielers messen kann. Dafür bieten sich insbesondere folgende zwei einfache Varianten an:

Der *Stochastische Spieler* wählt jeweils einen der möglichen Züge gleichverteilt zufällig aus. Er spielt damit genauso wie ein Lernender Spieler mit leerer Tabelle (oder

ein Optimierender Spieler mit leerer Tabelle) und kann insofern als dessen Ausgangsvariante aufgefasst werden. Das mag ihn als Vergleichsmaßstab, an dem man (eventuell sehr geringe) Lernfortschritte nachweisen kann, grundsätzlich geeignet erscheinen lassen. Gegen den Stochastischen Spieler spricht natürlich dessen lächerliche Spielstärke. Für seien Einsatz als Gegenspieler beim Training spricht, dass er seine Züge mit sehr geringem Rechenaufwand (= Zeitaufwand) wählt.

Zu Test- und späteren Vergleichszwecken habe ich den Stochastischen Spieler 10 mal je 1.000 Partien gegen sich selbst spielen lassen. Die Ergebnisse sind in Tabelle 1 wiedergegeben. Es zeigt sich – wie bei reiner Zufallsauswahl wohl zu erwarten – eine nicht unerhebliche Streuung der Ergebnisse. Der Mittelwert für die Erfolgsquote des Stochastischen Spielers mit den schwarzen Steinen liegt bei $\mu = 47,92\%$ mit einer (Stichproben-)Standardabweichung von $\sigma \approx 1,812947$ Prozentpunkten (zur Berechnung vergleiche [9], S. 67). Auch wenn der Wert von Reversi Null ist, es bei optimalem Spiel beider Spieler also zu einem Remis kommt, scheint es bei rein zufälliger Wahl der Züge einen leichten Vorteil für den Nachziehenden zu geben.

Wenn man unterstellt, dass die Erfolgsquote bei einer Partie zwischen zwei Stochastischen Spielern – als Summe einer hinreichend großen Zahl von identisch verteilten Zufallsvariablen – annähernd normalverteilt ist, ergibt sich ein 95%-Intervall für die Erfolgsquote des schwarzen Spielers von $\mu - 1,96 * \sigma \approx 44,366$ bis $\mu + 1,96 * \sigma \approx 51,473$. Für den Spieler mit den weißen Steinen ergibt sich $\mu - 1,96 * \sigma \approx 48,527$ bis $\mu + 1,96 * \sigma \approx 55,633$ (vergleiche [9], S. 73f). Wenn ein Algorithmus in 1.000 Testpartien mit Schwarz gegen den Stochastischen Spieler eine Erfolgsquote von über 51,473% oder mit Weiß von über 55,633% erzielt, kann man somit mit einer Verlässlichkeit von 95% schließen, dass der Algorithmus besser ist als eine rein zufällige Zugauswahl.

Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgsquote Schwarz
32553:31447	497	39	464	51,65%
31860:32140	468	46	486	49,10%
31745:32255	448	47	505	47,15%
31380:32620	447	47	506	47,05%
31545:32455	462	40	498	48,20%
31269:32731	436	34	530	45,30%
31615:32385	458	46	496	48,10%
31247:32753	436	41	523	45,65%
31467:32533	462	41	497	48,25%
31975:32025	472	31	497	48,75%

Tabelle 1: Ergebnisse von je 1.000 Partien Stochastischer Spieler gegen Stochastischen Spieler

Die zweite Variante des Gegen- und Vergleichsspielers ist der *Minimax-Spieler*.

Er berechnet - mit einer Tiefe von m Zügen - alle von der aktuellen Stellung aus erreichbaren Stellungen, bewertet diese mit der Differenz der Anzahl der weißen und schwarzen Steine und ermittelt von dort aus rückwärts entsprechend dem in Abschnitt 3.3 beschriebenen Minimax-Algorithmus den besten Zug für die aktuelle Stellung. Ist m größer gleich der maximalen Anzahl Züge von der gegebenen Stellung bis zu einer Endstellung, so spielt der Minimax-Spieler ab dieser Stellung bis zum Spielende optimal. Für $m > 60$ spielt er demnach fast ab Partiebeginn optimal. Wegen des hohen Rechenaufwands sind jedoch nur erheblich kleiner Werte praktisch implementierbar. Wie in Abschnitt 3.2 angemerkt, ist die Differenz der Steinanzahl für Stellungen insbesondere in frühen Partiestadien kein guter Bewertungsmaßstab. Ist m deutlich kleiner als die Anzahl der voraussichtlich bis zum Partieende noch verbleibenden Züge, kann man vom Minimax-Spieler in der hier konstruierten Variante keine sehr hohe Spielstärke erwarten.

In der experimentellen Überprüfung zeigt sich schon bei $m = 4$ deutlich der hohe (und mit m exponentiell steigende) Rechenaufwand des Minimax-Spielers. Eine Partie zwischen einem Minimax-Spieler und einem Stochastischen Spieler dauert mit diesem Wert von m auf der mir zur Verfügung stehenden Rechenanlage etwas 10 mal so lange wie eine Partie zwischen zwei Stochastischen Spielern. Dabei erreicht der Minimax-Spieler gegen den Stochastischen Spieler mit schwarz wie mit weiß eine Erfolgsquote von rund 86%, ist also – wenig überraschend – schon bei geringer Vorausschautiefe der rein zufälligen Zugauswahl weit überlegen. Damit sollte er auch der interessantere Vergleichsmaßstab für den Lernenden Spieler sein.

In Tabelle 2 sind die Ergebnisse von je 1.000 Testpartien zwischen dem Minimax-Spieler mit Tiefe 4 und dem Stochastischen Spieler wiedergegeben.

Minimax spielt mit	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgsquote Minimax
Schwarz	46.359 : 17.641	853	15	132	86,05%
Weiß	18.457 : 45.543	138	12	850	85,60%

Tabelle 2: Ergebnisse von je 1.000 Partien Minimax-Spieler mit Tiefe 4 gegen Stochastischen Spieler

4.3 Implementierungsdetails

Die oben dargestellten Algorithmen sind in Python implementiert. Der Quelltext ist wie folgt gegliedert:

In der Datei spiellogik.py wird die Klasse `Stellung` definiert, mit der die Spielregeln operationalisiert werden. `Stellung` ist von der numpy-Klasse `ndarray` abgeleitet und wird als 8×8-Matrix initialisiert. Die Matrix entspricht dem Spielbrett, und jedes freie Feld wird mit 0 markiert, jedes Feld mit einem Stein des Spielers, der am Zug ist, mit 1, und jedes Feld mit einem Stein des Spielers, der nicht am Zug ist,

mit -1. Durch Aufruf der Methode grundstellung wird die Matrix entsprechend der Grundstellung mit den Werten -1, 0 und 1 befüllt.

Weitere Methoden der Klasse sind moegliche_zuege und zug_spielen. Die erste Methode gibt eine Liste der in der jeweiligen Stellung möglichen Züge zurück. Ein möglicher Zug wird dabei durch die Koordinaten des Matrixfeldes, auf den ein Stein gesetzt werden soll, beschrieben und zusätzlich durch die Richtungen, in die von diesem Feld aus die umzudrehenden Steine des Gegenspielers liegen (UNTEN_RECHTS, RECHTS und so weiter). Letztere Information, die bei der Ermittlung der möglichen Züge ohnehin anfällt, erleichtert die Ausführung der Methode zug_spielen. Dieser wird ein Zug als Argument übergeben, woraufhin die Eintragungen in der Matrix so umgewandelt werden, dass sie der Folgestellung entspricht (einschließlich des Wechsels des Spielers, der nun am Zug ist).

Die Datei spiellogik.py enthält zudem die klassenfreie Funktion als_kanonische_stellung, die die kanonische Variante einer als Argument übergebenen Stellung zurückgibt. Dafür werden die Matrizen der an den vier Symmetrieachsen gespiegelten Stellung sowie der um 90, 180 und 270 Grad gedrehten Stellung ermittelt. Diese werden jeweils mit der ndarray-Methode tobytes in einen Byte-String umgewandelt. Auf Byte-Strings ist eine eindeutige Größer-als-Relation definiert. Die Stellung mit dem kleinsten Byte-String wird als die kanonische ausgewählt.

Die Datei spieler.py enthält die Zugauswahlmechanismen. Sie sind alle von einer abstrakten Grundklasse Spieler abgeleitet, die die Methode zug_waehlen vorgibt. Dieser wird als Argument eine Stellung übergeben, und sie gibt den vom jeweiligen Spieler in dieser Stellung gewählten Zug zurück.

Die fünf Unterklassen Lernender_Spieler_epsilon, Lernender_Spieler_sigma, Optimierender_Spieler, Stochastischer_Spieler und Minimax_ Spieler implementieren die jeweiligen Zugauswahlmechanismen so, wie in den Abschnitten 4.1 beziehungsweise 4.2 beschrieben. Der Minimax-Spieler implementiert die in Abschnitt 4.2 beschriebene α-β-Suche über eine rekursive Methode _minimax, die für eine an sie übergebene Stellung den Wert des mit dieser Stellung beginnenden Spielteilbaums berechnet oder die Berechnung abbricht, falls die ebenfalls an sie übergebenen α- oder β-Werte über- beziehungsweise unterschritten werden. Bei der Instantiierung wird dem Minimax-Spieler als Parameter die Tiefe übergeben, bis zu der er die Methode _minimax rekursiv einsetzen soll (= Anzahl der vorauszuberechnenden Züge).

Die drei Spielerklassen Lernender_Spieler_epsilon, Lernender_Spieler_sigma und Optimierender_Spieler verfügen über ein Attribut erfahrungsspeicher, das auf ein Objekt verweisen muss, das über eine Methode bewertung_geben eine Bewertung für eine als Argument übergebene Stellung zurückgibt. Klassen für derartige Objekte werden in der Datei bewertungsgeber.py definiert. Für einen „tabularen“ Algorithmus ist die Klasse Bewertungstabelle bestimmt. Sie hat ein Attribut bewertung, das auf ein Python-*dictionary* verweist, welches die Bewertungstabelle abbildet. Als Schlüssel wird in diesem *dictionary* der Byte-String der zu speichernden Stellung verwendet. Der Eintrag zu diesem Schlüs-

sel besteht entsprechend der Beschreibung in Abschnitt 4.1 aus einem Zweiertupel ganzer Zahlen, nämlich der Punktsumme und der Häufigkeit der jeweiligen Stellung.

Zudem gibt es die Bool'schen Attribute `schwarz` und `weiss`, die festlegen, ob in der Tabelle Bewertungen für einen Spieler mit den schwarzen oder den weißen Steinen (oder beides) gespeichert werden. Hat `schwarz` den Wert `True`, werden also Stellungen mit Bewertungen gespeichert, in denen weiß am Zug ist. Entsprechendes gilt für `weiss`.

Die Methode `bewertung_geben` wandelt eine als Argument übergebene Stellung in deren kanonische Variante um und gibt den Quotienten aus Punktsumme und Häufigkeit zurück, wenn die kanonische Stellung in der Tabelle gespeichert ist. Ansonsten gibt sie `None` zurück. (`None` wird dann gegebenenfalls in der Methode `zug_waehlen` eines Spielerobjekts durch einen Standardwert ersetzt.)

Die Methode `bewertung_aktualisieren` aktualisiert und ergänzt die Einträge im *dictionary* `bewertung` auf der Grundlage eines als Argument übergebenen Protokolls. Ein Protokoll ist eine Liste aller Züge, die in einer Partie gespielt wurden, mit dem Ergebnis der Partie als letztem Listeneintrag. In der Methode `bewertung_aktualisieren` werden ausgehend von der Grundstellung alle in der Partie aufgetretenen Stellungen nachvollzogen. Hat `schwarz` den Wert `True`, wird jede durch einen Zug von schwarz erreichte Stellung in ihre kanonische Variante umgewandelt und geprüft, ob diese bereits im *dictionary* enthalten ist. Wenn nicht, wird die kanonische Stellung mit dem Partieergebnis von schwarz und der Häufigkeit 1 im *dictionary* ergänzt, ansonsten wird die bestehende Bewertung durch Hinzuaddieren des Partieergebnisses und Inkrementieren der Häufigkeit aktualisiert. (Ist das Partieergebnis bei einer noch nicht im *dictionary* gespeicherten Stellung 0, so wird aus den in Abschnitt 4.1 dargestellten Gründen 2 als Punktsumme gespeichert.) Entsprechend wird vorgegangen, wenn `weiss` den Wert `True` hat.

Mit den Methoden `bewertung_speichern` und `bewertung_laden` wird das *dictionary* `bewertung` persistent gespeichert beziehungsweise ein persistent gespeichertes *dictionary* in ein `Bewertungstabelle`-Objekt eingelesen.

Um zu Trainings- oder Testzwecken eine Partie spielen zu lassen, ist in der Datei partieumgebung.py die Klasse `Partieumgebung` definiert. Deren Attribute `spieler_schwarz` und `spieler_weiss` verweisen jeweils auf ein `Spieler`-Objekt. Aus Sicht eines (Lernenden) Spielers bilden die Partieumgebung und der Spieler mit der anderen Farbe die Umgebung im Sinne des Grundmodells aus Abschnitt 2.1. Das Attribut `erfahrungsspeicher` verweist auf ein Objekt, das über eine Methode `bewertung_aktualisieren` verfügt. Das für Testzwecke zu nutzende Attribut `testprotokoll` verweist (wenn es nicht den Wert `None` annimmt) auf eine Liste aus einem Zweiertupel ganzer Zahlen und drei ganze Zahlen.

Mit der Methode `partie_starten` wird eine Partie durchgeführt, indem ausgehend von der Grundstellung jeweils abwechseln die Methode `zug_waehlen` von `spieler_schwarz` und `spieler_weiss` aufgerufen wird. Als Argument übergeben wird dabei jeweils die Stellung, die sich aus dem vorhergehenden Zug des je-

weiligen Gegenspielers ergibt. Jeder Zug wird in die Liste `protokoll` eingetragen. Vor Aufruf von `zug_waehlen` wird jeweils geprüft, ob eine finale Stellung erreicht wurde, also ob zweimal hintereinander gepasst wurde oder alle Felder belegt sind. Ist dies der Fall, so wird das Partieergebnis ermittelt und zur Liste `protokoll` hinzugefügt. Falls `erfahrungsspeicher` nicht den Wert `None` hat, wird die Methode `bewertung_ aktualisieren` von `erfahrungsspeicher` mit dem Argument `protokoll` aufgerufen.

Mit der Methode `test_starten` wird ebenfalls eine Partie durchgeführt. Sie ist aber für Testzwecke gedacht. Ihr Ablauf entspricht dem der Methode `partie_starten` mit dem Unterschied, dass kein Zugprotokoll geführt und am Ende einer Partie nicht auf eine Methode von `erfahrungsspeicher` zugegriffen wird. Dafür wird die Liste `testprotokoll` aktualisiert, indem das Partieergebnis zum Zweiertupel an Position 0 der Liste hinzuaddiert und bei einem Sieg von schwarz die Zahl an Position 1 der Liste, bei einem Remis die Zahl an Position 2 und bei einem Sieg von weiß die Zahl an Position 3 um eins erhöht wird.

Der Ablauf der Lernexperimente aus Abschnitt 4.4 ist in den Skripten lernskript_ epsilon.py (Abschnitt 4.4.1), lernskript_sigma.py (Abschnitt 4.4.2) sowie lernskript_ schwarz.py und lernskript_weiss.py (beide Abschnitt 4.4.3) festgelegt, in denen jeweils die benötigten Spieler-, Erfahrungsspeicher- und Partieumgebungsobjekte erzeugt und entsprechend den in den jeweiligen Abschnitten beschriebenen Parametern Lern- und Testpartien gestartet werden.

Als Hilfsmittel habe ich bei der Erstellung von Docstring-Kommentaren in den Dateien spiellogik.py, spieler.py, bewertungsgeber.py und partieumgebung.py das KI-Modell Gemini 2.5 Flash genutzt (siehe [7]).

4.4 Experimentelle Ergebnisse des „tabularen" RL-Reversi-Spielers

Für das Training des „tabularen" Algorithmus habe ich folgende Experimente durchgeführt: Zunächst habe ich beide Varianten des Lernenden Spielers jeweils eine Million Partien gegen sich selbst spielen lassen. Alle 100.000 Partien habe ich dabei durch Testpartien gegen den Stochastischen Spieler überprüft, ob eine Zunahme der Spielstärke nachgewiesen werden kann.

Wie schon mehrfach ausgeführt, ist es theoretisch problematisch, als Gegenspieler einen Lernenden Spieler zu verwenden. Insbesondere sind dann die in Abschnitt 2.2 referierten Konvergenzergebnisse für „tabulare" Algorithmen nicht gültig. Gleichwohl wird dieses Vorgehen in der Literatur häufig angewandt (siehe z. B. [23], S. 182, S. 187). Ein Vorteil bei diesem Verfahren ist, dass dabei gleichzeitig das Spiel mit den schwarzen und den weißen Steinen trainiert wird.

Alternativ habe ich den Lernenden Spieler gegen den Stochastischen Spieler trainiert. Auch hierbei wurden jeweils nach 100.000 Trainingspartien Testpartien gegen den Stochastischen Spieler durchgeführt.

Schließlich habe ich die mit den verschiedenen Verfahren trainierten Spieler gegen den Minimax-Spieler spielen lassen, um die erreichte Spielstärke nochmals an

einem zweiten, möglicherweise aussagekräftigeren Maßstab zu überprüfen.

4.4.1 Training der ϵ-*greedy*-Variante des Lernenden Spielers gegen sich selbst

Bei der ϵ-*greedy*-Variante ist der Parameter ϵ vorab festzulegen. Typischerweise wird die Wahrscheinlichkeit für zufällige explorative Züge im Verlauf des Trainings reduziert (vergleiche z. B. [23], S. 30). Deshalb habe ich das Training mit einem Wert von $\epsilon = 0{,}5$ begonnen. Nach 100.000 Partien wurde ϵ auf ⅓ abgesenkt, nach jeweils weiteren 100.000 Partien auf ¼, ⅕, ⅙, ⅐, ⅛ und ⅑. Die letzten 200.000 Partien wurden mit $\epsilon = 0{,}1$ gespielt.

Zur Messung des Lernfortschritts habe ich vor Beginn des Trainings und dann alle 100.000 Partien den Optimierenden Spieler je 1.000 Testpartien mit Schwarz und mit Weiß gegen den Stochastischen Spieler spielen lassen. Dabei griff der Optimierende Spieler auf die bis zu diesem Zeitpunkt vom Lernenden Spieler ermittelten Stellungsbewertungen zurück. Die Ergebnisse der Testpartien sind in Tabelle 3 dargestellt. Eine Runde bezeichnet dabei jeweils den Ablauf von 100.000 weiteren Trainingspartien. Angegeben ist die Ergebnissumme über die jeweils 1.000 Partien, die Anzahl der von Schwarz gewonnen Partien, die Anzahl von Unentschieden, die Anzahl von Weiß gewonnener Partien und die Erfolgsquote des Optimierenden Spielers sowie die Anzahl an bewerteten Stellungen, die der Optimierende Spieler in der jeweiligen Runde nutzen konnte.

Runde (ϵ)	Optimierender Spieler	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgsquote	Anzahl bewerteter Stellungen
0	Schwarz	31540:32460	456	41	503	47,65%	0
	Weiß	31462:32538	431	47	522	54,55%	
1	Schwarz	32188:31812	483	53	464	50,95%	5.085.215
(0,5)	Weiß	30438:33562	402	48	550	57,4%	
2	Schwarz	32618:31382	506	40	454	52,60%	9.730.051
(⅓)	Weiß	30397:33603	406	35	559	57,65%	
3	Schwarz	32177:31823	499	38	463	51,80%	13.863.559
(0,25)	Weiß	30249:33751	390	54	556	58,30%	
4	Schwarz	32227:31773	475	48	477	49,90%	17.402.689
(0,2)	Weiß	30710:33290	415	39	546	56,55%	
5	Schwarz	32309:31691	472	48	480	49,6%	20.359.455
(⅙)	Weiß	30297:33703	398	33	569	58,55%	
6	Schwarz	32189:31811	478	49	473	50,25%	22.778.992
(⅐)	Weiß	30462:33538	403	39	558	57,75%	
7	Schwarz	32306:31694	486	38	476	50,50%	24.750.628
(0,125)	Weiß	30401:33599	403	42	555	57,60%	
8	Schwarz	31984:32016	483	45	472	50,55%	26.355.191
(⅑)	Weiß	30653:33347	411	34	555	57,20%	
9	Schwarz	32392:31608	490	51	459	51,55%	27.666.536
(0,1)	Weiß	30870:33130	427	41	532	55,25%	
10	Schwarz	32514:31486	496	31	473	51,15%	28.921.926
(0,1)	Weiß	30767:33233	426	45	529	55,15%	

Tabelle 3: Ergebnisse des „tabularen“ Spielers in der ϵ-*greedy*-Variante vor Trainingsbeginn und nach jeweils weiteren 100.000 Trainingspartien

Die Entwicklung der Erfolgsquoten ist als blaue Linie (Version 1) für das Spiel mit schwarz in Abb. 5 und mit weiß in Abb. 6 abgebildet. Nach dem in Abschnitt 4.2 hergeleiteten Minimalkriterium ist festzustellen, dass ein Lernfortschritt gegenüber einer reinen Zufallsauswahl mit schwarz (nur) in den Runden 2, 3 und 9 auftritt, mit weiß (immerhin) in den Runden von 1 bis 8. Eine kontinuierliche Verbesserung der Spielstärke über den Trainingsverlauf lässt sich nicht feststellen. Nach einer kurzen anfänglichen Steigerung fällt die Spielstärke wieder, mit weiß sogar bis auf das Niveau vor dem Training.

4.4.2 Training der σ-*greedy*-Variante des Lernenden Spielers gegen sich selbst

Auch die σ-*greedy*-Variante wurde über eine Million Partien gegen sich selbst trainiert, wobei der Lernfortschritt wiederum vor Beginn des Trainings und dann al-

le 100.000 Partien durch jeweils 1.000 Testpartien des Optimierenden Spielers mit Schwarz und mit Weiß gegen den Stochastischen Spieler überprüft wurde. Die Ergebnisse der Testpartien sind in Tabelle 4 dargestellt.

Runde	Optimierender Spieler	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgsquote	Anzahl bewerteter Stellungen
0	Schwarz	31625:32375	454	41	505	47,45%	0
	Weiß	31504:32496	466	31	503	51,85%	
1	Schwarz	32305:31695	491	49	460	51,55%	5.275.665
	Weiß	31520:32480	440	40	520	54,00%	
2	Schwarz	32161:31839	474	46	480	49,70%	10.472.616
	Weiß	31101:32899	444	35	521	53,85%	
3	Schwarz	32679:31321	496	44	460	51,80%	15.639.704
	Weiß	31300:32700	436	45	519	54,15%	
4	Schwarz	32946:31054	515	37	448	53,35%	20.787.332
	Weiß	31105:32895	445	39	516	53,55%	
5	Schwarz	32838:31162	509	43	448	53,05%	25.920.773
	Weiß	30607:33393	418	32	550	56,60%	
6	Schwarz	33001:30999	529	46	425	55,20%	31.042.952
	Weiß	31091:32909	426	33	541	55,75%	
7	Schwarz	32950:31050	503	42	455	52,40%	36.156.371
	Weiß	30851:33149	428	40	532	55,20%	
8	Schwarz	32924:31076	524	44	432	54,60%	41.260.330
	Weiß	30896:33104	439	32	529	54,50%	
9	Schwarz	33311:30689	513	34	453	53,00%	46.357.806
	Weiß	30537:33463	396	51	553	57,85%	
10	Schwarz	33223:30777	527	38	435	54,60%	51.450.022
	Weiß	30259:33741	405	40	555	57,50%	

Tabelle 4: Ergebnisse des „tabularen" Spielers in der σ-*greedy*-Variante vor Trainingsbeginn und nach jeweils weiteren 100.000 Trainingspartien

Die Entwicklung der Erfolgsquoten ist als grüne Linie (Version 2) für das Spiel mit schwarz in Abb. 5 und mit weiß in Abb. 6 abgebildet. Ein Lernfortschritt gegenüber einer reinen Zufallsauswahl ist mit schwarz in den Runden 1 sowie 3 bis 10 zu verzeichnen, mit weiß in den Runden 5, 6, 9 und 10. Anders als bei der ϵ-*greedy*-Variante lässt sich bei beiden Farben über den Trainingsverlauf hinweg eine tendenzielle Zunahme der Spielstärke beobachten. Allerdings steigt die Spielstärke mit weiß erst gegen Ende des Trainingsverlaufs über die mit der ϵ-*greedy*-Variante erreichten Werte, und erreicht in keinem Fall ein Niveau, das weit über dem der reinen Zufallsauswahl läge.

Auffällig ist, dass die Anzahl der bewerteten Stellungen hier erheblich höher ist

als bei der ϵ-*greedy*-Variante. Dies mag teilweise erklären, warum es bei der σ-*greedy*-Variante einen konsistenteren Aufwärtstrend bei der Spielstärke gibt. Die Anzahl bewerteter Stellungen hängt bei der ϵ-*greedy*-Variante natürlich insbesondere von den gewählten ϵ-Werten ab. Die Vermutung liegt nahe, dass bei höheren Werten ein besserer Lernerfolg hätte erreicht werden können.

4.4.3 Training des Lernenden Spielers gegen den Stochastischen Spieler

Insbesondere bei der ϵ-*greedy*-Variante des Lernenden Spielers wurde beobachtet, dass sich die Spielstärke im Training erst etwas erhöht hat, danach aber wieder zurückging. Theoretisch könnte dies auch daran liegen, dass sich die Umwelt im Trainingsprozess verändert hat, also der Gegenspieler auch etwas gelernt und seine Spielweise verändert hat. Das mag angesichts des insgesamt geringen Lernfortschritts im Training wenig wahrscheinlich erscheinen. Gleichwohl legt es nahe, auch ein Trainingsexperiment mit einem unveränderten Gegenspieler durchzuführen.

Runde	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote	Anzahl bewerteter Stellungen
0	31604:32396	445	40	515	46,50%	0
1	32981:31019	510	49	441	53,45%	2.625.648
2	32838:31162	519	29	452	53,35%	5.215.209
3	32677:31323	506	44	450	52,80%	7.788.804
4	33377:30623	546	29	425	56,05%	10.351.096
5	32935:31065	510	41	449	53,05%	12.905.140
6	32595:31405	499	41	460	51,95%	15.453.336
7	32565:31435	492	42	466	51,30%	17.996.599
8	33243:30757	540	47	413	56,35%	20.534.894
9	33046:30954	490	54	456	51,70%	23.069.909
10	32715:31285	511	31	458	52,65%	25.601.654

Tabelle 5: Ergebnisse des gegen den Stochastischen Spieler trainierten „tabularen“ Spielers mit schwarz vor Trainingsbeginn und nach jeweils weiteren 100.000 Trainingspartien

In diesem Fall ist das Training mit schwarz und weiß getrennt durchzuführen. Ich habe den Lernenden Spieler daher mit jeder Farbe jeweils eine Million Trainingspartien gegen den Stochastischen Spieler spielen lassen. Dafür habe ich die ϵ-*greedy*-Variante des Lernenden Spielers verwendet, da diese in den vorhergehenden Experimenten etwas besser abgeschnitten hat. Wiederum wurden anfangs und nach jeweils 100.000 Trainingspartien 1.000 Testpartien mit dem Optimierenden Spieler gegen den Stochastischen Spieler durchgeführt, um den Lernfortschritt zu messen.

Die Ergebnisse der Testpartien aus dem Training, in dem der Lernende Spieler mit schwarz spielte, sind in Tabelle 5 wiedergegeben.

Die Entwicklung der Erfolgsquoten ist als rote Linie (Version 3) in Abb. 5 dargestellt. Das Minimalkriterium, von der einen Zufallsauswahl unterscheidbar zu sein, wird hier in allen Runden außer Runde 7 erfüllt. Ein Maximum der Spielstärke wird nach 800.000 Partien erreicht. Gleichwohl lässt sich auch hier kein durchweg steigender Verlauf der Spielstärke erkennen.

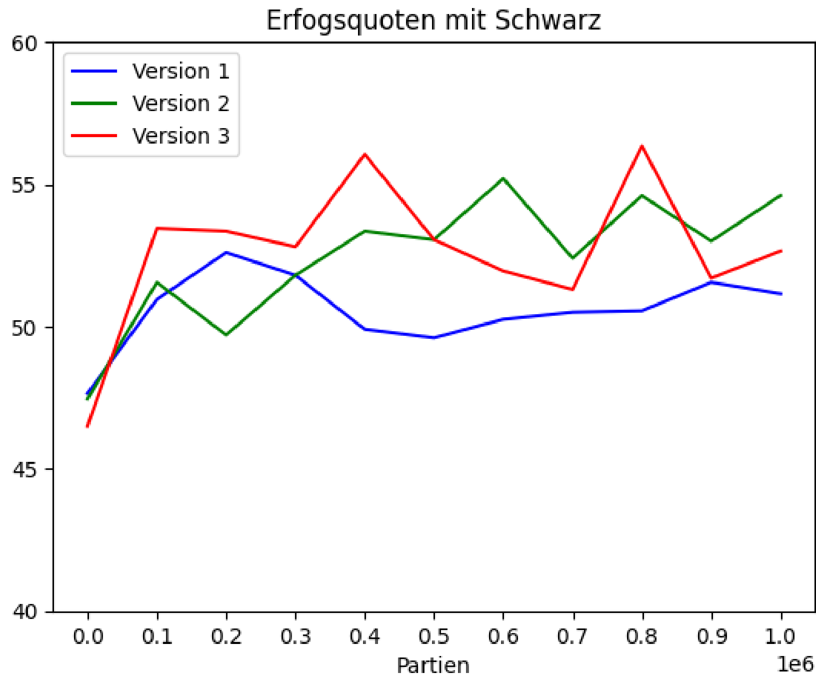

Abbildung 5: Erfolgsquoten mit schwarz im Trainingsverlauf. Version 1: ϵ-*greedy*-Variante gegen sich selbst trainiert; Version 2: σ-*greedy*-Variante gegen sich selbst trainiert; Version 3: σ-*greedy*-Variante gegen Stochastischen Spieler trainiert

Die Ergebnisse der Testpartien aus dem Training, in dem der Lernende Spieler mit schwarz spielte, sind in Tabelle 6 wiedergegeben.

Runde	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote	Anzahl bewerteter Stellungen
0	31309:32691	450	34	516	53,30%	0
1	31222:32778	447	30	523	53,80%	2.649.062
2	31015:32985	437	39	524	54,35%	5.256.148
3	30953:33047	418	39	543	56,25%	7.849.198
4	30526:33474	407	44	549	57,10%	10.434.021
5	31037:32963	431	35	534	55,15%	13.012.785
6	30493:33507	407	37	556	57,45%	15.586.300
7	30254:33746	398	37	565	58,35%	18.155.511
8	30421:33579	410	54	536	56,30%	20.721.283
9	30339:33661	409	29	562	57,65%	23.282.862
10	29991:34009	379	39	582	60,15%	25.841.581

Tabelle 6: Ergebnisse des gegen den Stochastischen Spieler trainierten „tabularen“ Spielers mit schwarz vor Trainingsbeginn und nach jeweils weiteren 100.000 Trainingspartien

Die Entwicklung der Erfolgsquoten ist als rote Linie (Version 3) in Abb. 6 dargestellt. Das Minimalkriterium, von der einen Zufallsauswahl unterscheidbar zu sein, wird hier in den Runden 3, 4 sowie 6 bis 10 erfüllt. Der Verlauf der Spielstärke erscheint wellig, nimmt aber tendenziell über den gesamten Trainigsverlauf zu. Von allen Trainingskurven sieht diese noch am ehesten so aus, wie man es sich für ein erfolgreiches Training erhoffen würde. Jedoch bleibt auch hier die maximal erreichte Erfolgsquote deutlich unter der, die der Minimax-Spieler gegen den Stochastischen Spieler erreicht.

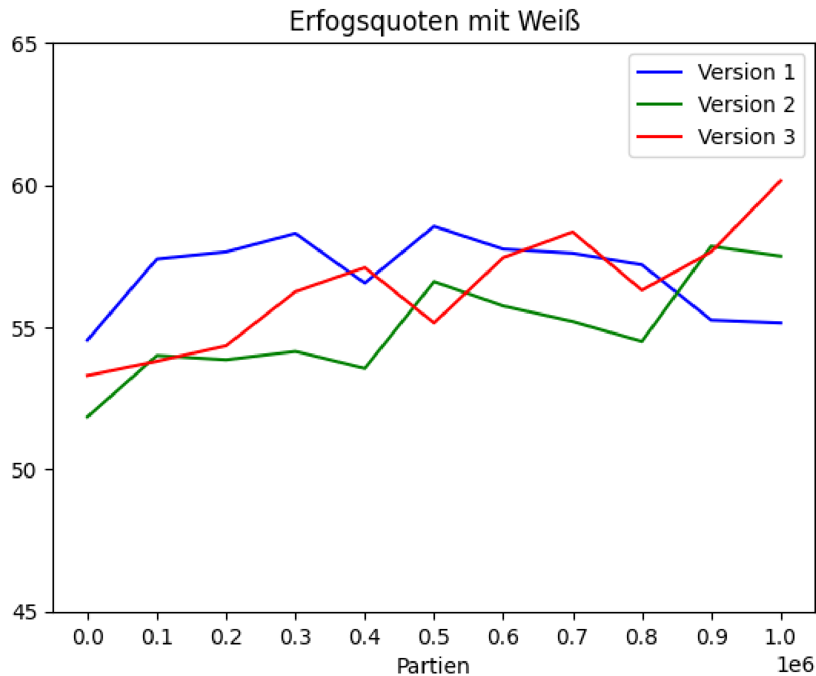

Abbildung 6: Erfolgsquoten mit weiß im Trainingsverlauf. Version 1: ϵ-*greedy*-Variante gegen sich selbst trainiert; Version 2: σ-*greedy*-Variante gegen sich selbst trainiert; Version 3: σ-*greedy*-Variante gegen Stochastischen Spieler trainiert

4.4.4 Testergebnisse gegen den Minimax-Spieler

Zur alternativen Messung des erreichten Lernfortschritts habe ich den Optimierenden Spieler mit den in den verschiedenen Trainingsexperimenten gewonnenen Bewertungen mit schwarz und weiß jeweils 1.000 Testpartien gegen den Minimax-Spieler (mit $m = 4$) spielen lassen. Tabelle 7 verzeichnet die jeweiligen Ergebnisse.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	17885:46115	139	13	848	14,55%
Version 1 (weiß)	41036:22964	767	19	214	22,35%
Version 2 (schwarz)	22131: 41869	198	21	781	20,85%
Version 2 (weiß)	46565:17435	895	15	90	9,75%
Version 3 (schwarz)	22285:41715	225	38	737	24,40%
Version 3 (weiß)	35216:28784	617	32	351	36,70%

Tabelle 7: Testergebnisse gegen den Minimax-Spieler. Version 1: ϵ-*greedy*-Variante gegen sich selbst trainiert; Version 2: σ-*greedy*-Variante gegen sich selbst trainiert; Version 3: σ-*greedy*-Variante gegen Stochastischen Spieler trainiert.

Es bestätigt sich, dass die erreichte Spielstärke in allen Varianten deutlich unter der des Minimax-Spielers bleibt. Überraschend scheint hingegen, dass die Reihenfolge der Spielstärken der Varianten beim Test gegen den Minimax-Spieler nicht mit der Reihenfolge übereinstimmt, die sich beim Test gegen den Stochastischen Spieler ergeben hat. Mit den schwarzen Steinen hatte die σ-*greedy*-Variante des Lernenden Spielers am Ende des Trainingsverlaufs gegen den Stochastischen Spieler eine höhere Erfolgsquote als der gegen den Stochastischen Spieler trainierte Lernende Spieler und dieser wiederum eine höhere Erfolgsquote als die ϵ-*greedy*-Variante des Lernenden Spielers erreicht. Beim Test gegen den Minimax-Spieler schneidet die σ-*greedy*-Variante hingegen etwas schlechter ab als der gegen den Stochastischen Spieler trainierte Lernende Spieler. Dies ist in Abb. 7 graphisch dargestellt.

Mit den weißen Steinen ist die Erfolgsquote der σ-*greedy*-Variante im Test gegen den Minimax-Spieler schlechter als die aller anderen Varianten (sogar deutlich schlechter als die der reinen Zufallsauswahl), während sie im Test gegen den Stochastischen Spieler höher war als die der ϵ-*greedy*-Variante. Abb. 8 stellt dies graphisch dar. Mit beiden Farben schneidet der gegen den Stochastischen Spieler trainierte Lernende Spieler im Test gegen den Minimax-Spieler unter allen trainierten Varianten jeweils am besten ab.

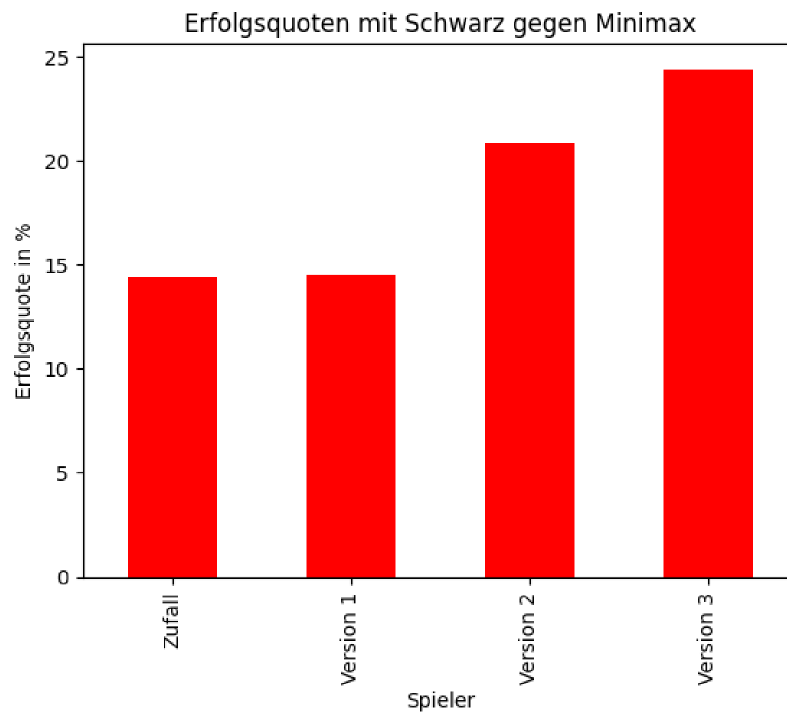

Abbildung 7: Erfolgsquoten mit schwarz gegen den Minimax-Spieler. Version 1: ϵ-*greedy*-Variante gegen sich selbst trainiert; Version 2: σ-*greedy*-Variante gegen sich selbst trainiert; Version 3: σ-*greedy*-Variante gegen Stochastischen Spieler trainiert

Hinzuweisen ist noch einmal darauf, dass bei diesen Tests jeweils die Tabelle mit der maximalen Anzahl an Bewertungen verwendet wurde, also die am Ende des Trainingsverlaufs vorliegende Tabelle. Wie in Abb. 5 und 6 zu sehen, wurden teilweise im Trainingsverlauf mit kleineren Tabellen bessere Testwerte gegen den Stochastischen Spieler erreicht. Es läge daher nahe, den Optimierenden Spieler auch mit diesen kleineren Tabellen, mit denen jeweils die besten Testwerte erreicht wurden, gegen den Minimax-Spieler spielen zu lassen. Wegen des damit verbundenen Zeit- und Speicherplatzaufwandes wurden diese Tabellenvarianten jedoch während des Trainings nicht dauerhaft gespeichert und stehen daher im Nachhinein nicht für Tests zur Verfügung.

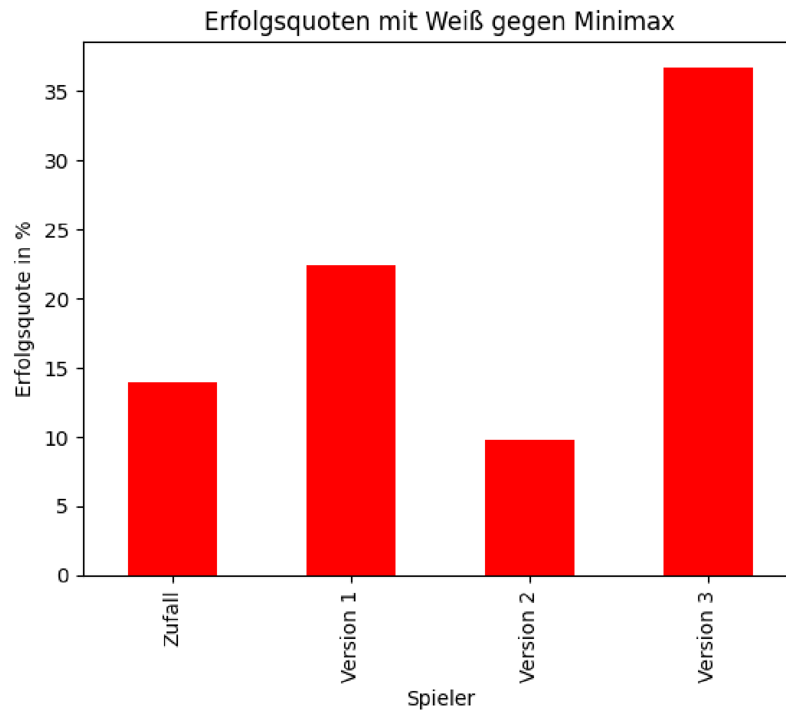

Abbildung 8: Erfolgsquoten mit weiß gegen den Minimax-Spieler. Version 1: ϵ-*greedy*-Variante gegen sich selbst trainiert; Version 2: σ-*greedy*-Variante gegen sich selbst trainiert; Version 3: σ-*greedy*-Variante gegen Stochastischen Spieler trainiert.

4.5 Zwischenfazit

Mit dem „tabularen" RL-Algorithmus lässt sich ein minimaler Lernfortschritt erkennen, der Algorithmus lernt jedoch sehr langsam und erreicht nur eine sehr geringe Spielstärke.

Es drängen sich zahlreiche Möglichkeiten auf, wie die durchgeführten Trainingsexperimente abgewandelt werden können, um eventuell bessere Ergebnisse zu erreichen. So könnte die ϵ-*greedy*-Variante mit höheren (insbesosndere weniger schnell absinkenden) ϵ-Werten getestet werden. Im Test gegen den Minimax-Spieler könnten die jeweils optimalen statt der maximalen Tabellen verwendet werden. Es könnten höhere oder niedrigere Standardwerte für bislang nicht bewertete Stellungen verwendet werden. Die Trainingsdauer könnte verlängert werden, beispielsweise bis zwei Millionen Partien. Statt gegen sich selbst oder den Stochastischen Spieler könnte der Lernende Spieler gegen den Minimax-Spieler trainiert werden.

Das grundsätzliche Problem des „tabularen" Ansatzes, dass nur ein sehr kleiner Anteil der möglichen Stellungen in einer Tabelle gespeichert werden kann, würde allerdings von keiner dieser möglichen Verfeinerungen gelöst. Es scheint daher

angemessen, die Untersuchung „tabularer“ Algorithmen hier zu beenden und stattdessen Ansätze zu betrachten, die auch noch nicht beobachtete Stellungen bewerten können.

5 Geradlinige Implementierung eines RL-Reversi-Spielers mit Netzwerk statt Tabelle

Ein Problem des in Abschnitt 4 entwickelten Algorithmus besteht darin, dass der Optimierende Spieler die vorliegenden Erfahrungen nur dann nutzt, wenn er auf eine Stellung trifft, die schon im Training aufgetreten ist. Bei bislang unbekannten Stellungen wählt er einen Zufallszug. Eine Idee, um dieses Problem zu überwinden, besteht darin, die vorliegenden Erfahrungen dafür zu nutzen, eine allgemeine Bewertungsfunktion herzuleiten, mit der jeder beliebigen Stellung ein Wert zugewiesen werden kann. Die Bewertung einer bislang unbekannten Stellung wird dabei aus den Bewertungen ähnlicher bekannter Stellungen abgeleitet.

Eine derartige Bewertungsfunktion lässt sich mithilfe eines neuronalen Netzwerks konstruieren, das mit den vom Lernenden Spieler in seiner Tabelle gesammelten Daten trainiert wird. Dabei können die Standardmethoden des überwachten Lernens verwendet werden. Die Eingabe in das Netzwerk ist eine Stellungsbeschreibung, also eine 8×8-Matrix mit Werten aus der Menge $\{-1, 0, 1\}$, die Ausgabe das zu erwartende Spielergebnis aus Sicht des Spielers, der nicht am Zug ist, also eine rationale Zahl im Intervall [0, 64]. Formal ist die Bewertungsfunktion somit als $B : \{-1, 0, 1\}^{8 \times 8} \to [0, 64]$ darstellbar.

5.1 Auswahl einer geeigneten Netzarchitektur

Eine einfache Architektur für das Bewertungsnetzwerk besteht aus mehreren hintereinander angeordneten Schichten von Neuronen (Netzwerkknoten), wobei von jedem Neuron je eine Verbindung zu allen Neuronen der nächstfolgenden Schicht ausgeht und es ansonsten keine weiteren Verbindungen zwischen Neuronen gibt. Ein solches Netzwerk wird als vollverbundenes vorwärtsgerichtetes Netzwerk oder als *Multilayer Perceptron* (MLP) bezeichnet, siehe z. B. [6], S. 163.

Bei einem MLP ergibt sich die Anzahl der Neuronen in der ersten Schicht, der Eingabeschicht, aus dem Format der Eingabe. Im gegebenen Fall werden 64 Neuronen benötigt. Die Anzahl der Neuronen in der letzten Schicht, der Ausgabeschicht, ergibt sich entsprechend aus dem Format der Ausgabe und beträgt hier eins. Die Anzahl der zwischen Eingabe- und Ausgabeschicht liegenden, sogenannten „verborgenen“ Schichten und die Anzahl der Neuronen in diesen verborgenen Schichten sind Architekturparameter, für die möglichst günstige Werte gefunden werden müssen. Wie in [6], S. 191ff, dargelegt, ist die optimale Anzahl und Größe verborgener Schichten für eine gegebene Aufgabe in der Regel experimentell zu ermitteln. Zwar besagt das Repräsentationstheorem, dass eine große Klasse von Funktionen durch ein Netzwerk mit nur einer verborgenen Schicht nachgebildet werden kann.

Gleichwohl ist es häufig sinnvoll, ein MLP mit mehreren verborgenen Schichten zu verwenden, da sich ein solches in vielen Fällen besser trainieren lässt.

Ein MLP behandelt die eingegebene Stellung wie einen 64-stelligen Vektor. Eine alternative Architektur, die unmittelbar die zweidimensionale Matrixstruktur der Eingabe berücksichtigt, bietet ein Faltendes Netzwerk (*Convolutional Neural Netwerk*, CNN), siehe z. B. [6], S. 321ff. Es liegt nahe zu vermuten, dass für die Bewertung einer Reversi-Stellung deren zweidimensionale Struktur, also beispielsweise die Frage, ob Steine auf der gleichen Diagonale liegen, von Bedeutung sein kann. Folglich könnte es für ein CNN leichter sein als für ein MLP, eine Bewertungsfunktion zu erlernen.

Der Darstellung in [6] folgend, lässt sich eine Faltungsoperation in einem CNN so beschreiben, dass dabei mithilfe eines Filters gewichtete Durchschnitte über benachbarte Felder der eingegebenen Matrix berechnet werden. Die Größe eines Filters bestimmt, welche Nachbarfelder dabei berücksichtigt werden. Bei einem 3×3–Filter werden beispielsweise alle direkt angrenzenden Felder in die Durchschnittsberechnung einbezogen. Architekturparameter sind dabei die Anzahl an aufeinander folgenden Faltungsoperationen und die Anzahl und Größe der dabei jeweils verwendeten Filter. Auf die Faltungsoperationen folgt als zweiter Teil eines CNN in der Regel ein MLP, das die einzelnen Werte in den durch die Faltungen entstandenen Durchschnittsmatrizen weiterverarbeitet und in das benötigte Ausgabeformat bringt. Für diesen zweiten Teil eines CNN sind wiederum Anzahl und Größe der verborgenen Schichten die Architekturparameter. Wie bei einem MLP sind auch für ein CNN die günstigsten Parameterwerte experimentell zu bestimmen.

CNN werden häufig zur Verarbeitung von Bildern verwendet. Wenn es sich dabei um Farbbilder handelt, wird jeder Punkt der zweidimensionalen Bildmatrix durch mehrere unterschiedliche Farbwerte beschrieben, und zwar typischerweise durch drei Werte für die Farben rot, blau und grün (siehe [6], S. 337). In Analogie dazu könnte man auch die eingegebenen Stellungen in mehrere Kanäle aufspalten: einen Kanal, in dem nur die Steine des am Zug befindlichen Spielers durch Einsen markiert werden, einen Kanal, in dem die Steine des Gegenspielers durch Einsen markiert werden, und einen dritten Kanal, in dem die freien Felder durch Einsen markiert werden. Die jeweils nicht markierten Felder erhalten dabei jeweils den Wert Null. Eine derartige Darstellung enthält zwar nicht mehr Informationen als die ursprüngliche Stellungsbeschreibung mit den Werten -1, 0 und 1. Die Vermutung, dass ein Netzwerk eine Eingabe leichter verarbeiten kann, wenn es nicht zwischen positiven und negativen Werten unterscheiden muss und die verschiedenen Aspekte der Stellung getrennt vorliegen, scheint aber zumindest eine Überprüfung wert zu sein.

5.2 Implementierungsdetails

Zur programmiertechnischen Implementierung der verschiedenen Netzwerke in Python wird die Programmbibliothek PyTorch verwendet (siehe [10]). Für die Nut-

zung eines MLP wird in der Datei bewertungsgeber.py die Klasse `Bewertungsnetz` definiert, für ein CNN die Klasse `Faltendes_Bewertungsnetz`. Beide Klassen sind von der PyTorch-Klasse `nn.Module` abgeleitet. Sie verfügen genau wie die Klasse `Bewertungstabelle` über eine Methode `bewertung_geben`, die für eine übergebene Stellung eine Bewertung zurückgibt. Objekte der beiden Klassen können daher in jedem Spieler-Objekt, das ein Attribut `erfahrungsspeicher` hat, für dieses Attribut verwendet werden. Die im „tabularen" Algorithmus eingesetzten `Spieler`-Klassen können daher unverändert weiterverwendet werden, wenn die Tabelle durch ein Netzwerk ersetzt wird.

Das verwendete MLP verfügt über zwei verborgene Schichten. Die erste umfasst 96 Neuronen, die zweite 34 Neuronen. Als Aktivierungsfunktion wird in allen Neuronen der verborgenen Schichten der *tangens hyperbolicus* verwendet. Bei Proberechnungen mit einer kleinen Stichprobe von Bewertungen lieferte diese Funktion deutlich bessere Anpassungswerte als die alternativ ausprobierte ReLU-Funktion $max(x, 0)$. Die Gesamtarchitektur des MLP ist in Abb. 9 schematisch dargestellt.

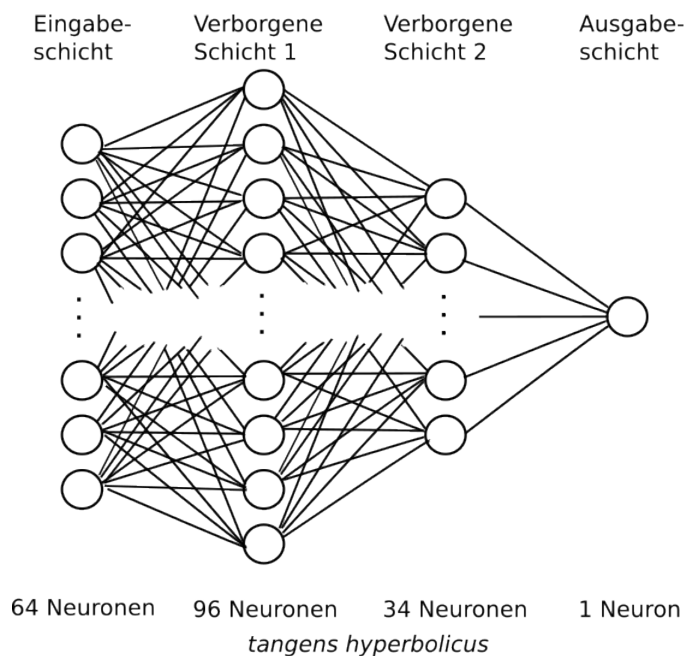

Abbildung 9: MLP-Architektur

Der Ablauf des Netztrainings mit den Tabellendaten ist im Skript netztraining.py festgelegt. Die Daten werden im Skript eingelesen und als PyTorch-Tensoren zu zwei Dritteln in eine Trainingsliste und zu einem Drittel in eine Testliste eingefügt. Wie in [10] als Standardverfahren beschrieben, werden beide Listen jeweils in ein Objekt einer von der PyTorch-Klasse `utils.data.Dataset` abgeleiteten Klasse eingefügt. Dafür wird in bewertungsgeber.py von `utils.data.Dataset` die Klasse `Bewertungsdatensatz` abgeleitet. Die beiden Bewertungsdatensatz-Objekte

werden wiederum jeweils in ein Objekt der PyTorch-Klasse `utils.data.DataLoader` für Trainings- und Testdaten eingefügt.

Weiterhin dem in [10] empfohlenen Standardverfahren folgend, werden eine Funktion `training_loop` und eine Funktion `test_loop` definiert. In `training_loop` werden Trainingsdaten aus dem DataLoader-Objekt entnommen, damit eine Netzausgabe berechnet, mit einer Verlustfunktion die Abweichung von Netzausgabe und Zielbewertung bewertet und auf dieser Grundlage mit *backpropagation* (siehe z. B. [6], S. 197ff) die Netzgewichte angepasst. Als Verlustfunktion wird dabei die mittlere quadrierte Abweichung verwendet.

In `test_loop` werden Testdaten aus dem DataLoader-Objekt entnommen, damit eine Netzausgabe und der sich daraus ergebende Verlust sowie zusätzlich das Bestimmtheitsmaß R^2 berechnet. Erreicht das Bestimmtheitsmaß einen besseren Wert als das bisherige Maximum, werden die aktuellen Netzgewichte gespeichert. Die Funktion `test_loop` erhält den besten bislang erreichten Bestimmtheitswert als Parameter übergeben und gibt ihrerseits den besten Bestimmtheitswert – also entweder den erhaltenen oder den selbst berechneten, falls dieser besser ist – zurück. Im Skript werden `training_loop` und `test_loop` für eine festgelegte Anzahl von Epochen (= vollständige Durchgänge durch die Trainingsdaten) abwechselnd aufgerufen.

Beim Bewertungsnetz mit CNN habe ich mich auf der Grundlage von Proberechnungen mit einer kleinen Stichprobe von Bewertungen für die Variante entschieden, bei der die eingegebenen Stellungen in drei Kanäle aufgespalten werden. Die erste Faltungsschicht des in der Klasse `Faltendes_Bewertungsnetz` definierten CNN erwartet daher eine Eingabe im Format $3 \times 8 \times 8$. Auf jeden der Eingabekanäle werden jeweils drei Filter der Größe 3×3 mit halbem Padding und einer Schrittweite von eins angewendet. Für jeden Kanal ergeben sich damit drei *feature maps* im Format 8×8 (siehe [6], S. 323), auf die in der zweiten Faltungsschicht erneut jeweils drei Filter der Größe 3×3 mit halbem Padding und einer Schrittweite von eins angewendet werden. Aus dieser Schicht gehen damit neun *feature maps* im Format 8×8 hervor. Proberechnungen haben gezeigt, dass es günstiger ist, die *feature maps* für die drei Kanäle in beiden Faltungsschichten separat zu halten und in den Faltungsschichten keine Aktivierungsfunktion zu verwenden (= die Identitätsfunktion als Aktivierungsfunktion zu verwenden). Zudem wird auf ein die *feature maps* verkleinerndes Pooling nach den Faltungen verzichtet, da schon die Eingabematrizen mit dem Format 8×8 vergleichsweise klein, sodass eine weitere Formatverkleinerung nicht notwendig erscheint.

Auf die zweite Faltungsschicht folgt eine MLP-Schicht mit 300 Neuronen, bei denen wiederum *tangens hyperbolicus* als Aktivierungsfunktion verwendet wird. Darauf folgt die Ausgabeschicht mit einem Neuron.

Wie erwähnt, erwartet die erste Faltungsschicht eine Eingabe im Drei-Kanal-Format. Der Methode `bewertung_geben` wird jedoch weiterhin ein Element aus der Menge $\{-1, 0, 1\}^{8 \times 8}$ als Stellung übergeben. Die Umwandlung in das Drei-Kanal-Format findet dann in der Methode `bewertung_geben` statt, wo die umgewandel-

te Stellung schließlich – durch Aufruf der überschriebenen Methode `forward` von `nn.Module` – an das „eigentliche" CNN weitergeben wird. Beim Training des CNN mit den Tabellendaten, bei dem die Methode `forward` ohne vorherigen Aufruf von `bewertung_geben` genutzt wird, muss die Drei-Kanal-Aufteilung im Trainingsskript faltendes_netztraining.py außerhalb des CNN erfolgen. Ansonsten entspricht der Ablauf in faltendes_netztraining.py dem in netztraining.py.

Die gesamte CNN-Architektur einschließlicher der Drei-Kanal-Aufteilung ist in Abb. 10 schematisch dargestellt.

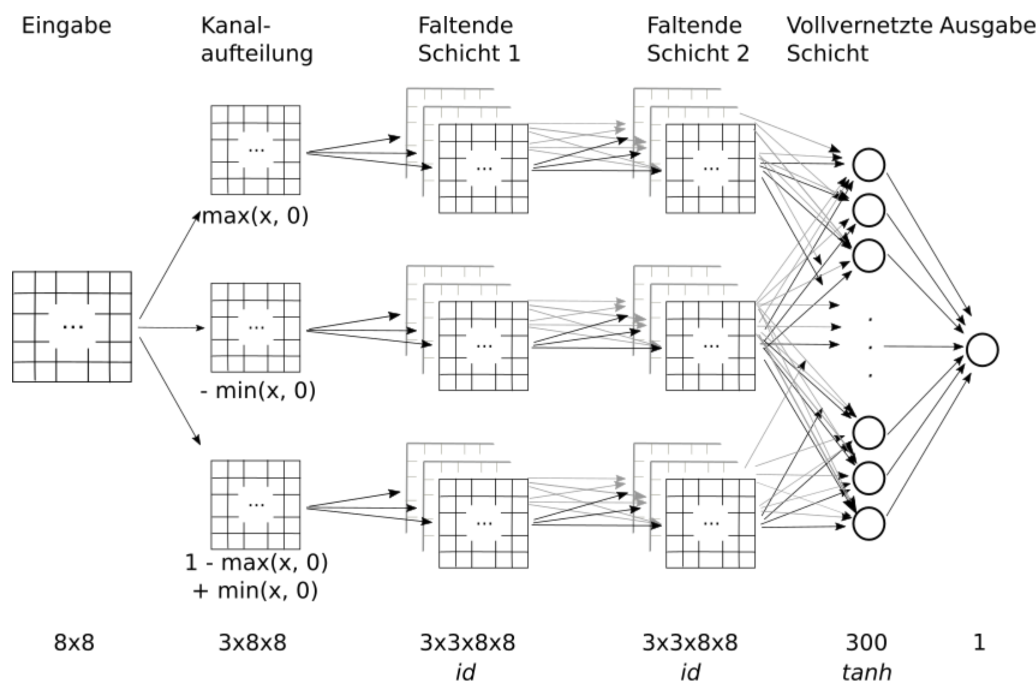

Abbildung 10: CNN-Architektur einschließlich Kanalaufteilung

5.3 Experimentelle Ergebnisse des RL-Reversi-Spielers mit Netzwerk

Für die Anwendung des RL-Reversi-Spielers mit Netzwerk ist zuerst das Netzwerktraining mit den Tabellendaten durchzuführen, bevor mit den dabei generierten Netzgewichten die erreichte Spielstärke experimentell überprüft werden kann.

5.3.1 Training der Bewertungsnetze

Aus den in Abschnitt 4.4 berichteten Experimenten liegen vier Tabellen vor (zwei mit Bewertungen aus Beobachtungen für beide Farben und zwei mit Bewertungen jeweils aus Beobachtungen nur für eine Farbe), die für das Training von Netzwerken genutzt werden können. Mit jedem dieser Datenbestände habe ich ein Bewertungsnetz vom MLP-Typ und ein Bewertungsnetz vom Drei-Kanal-CNN-Typ trainiert,

um zu sehen, welche Kombination aus Tabelle und Netztyp zur besten Anpassung führt (= zu den höchsten Werten für das Bestimmtheitsmaß).

Für die aus Trainingspartien der ϵ-*greedy*-Variante des Lernenden Spielers gegen sich selbst gewonnenen Daten (im Folgenden: „Version 1") ergeben sich sehr gute Anpassungswerte. Mit dem MLP wird ein Bestimmtheitsmaß von 86,11% erreicht, mit dem Drei-Kanal-CNN ein Bestimmtheitsmaß von 86,51%. Der Trainingsverlauf ist in Abb. 11 und Abb. 12 dargestellt.

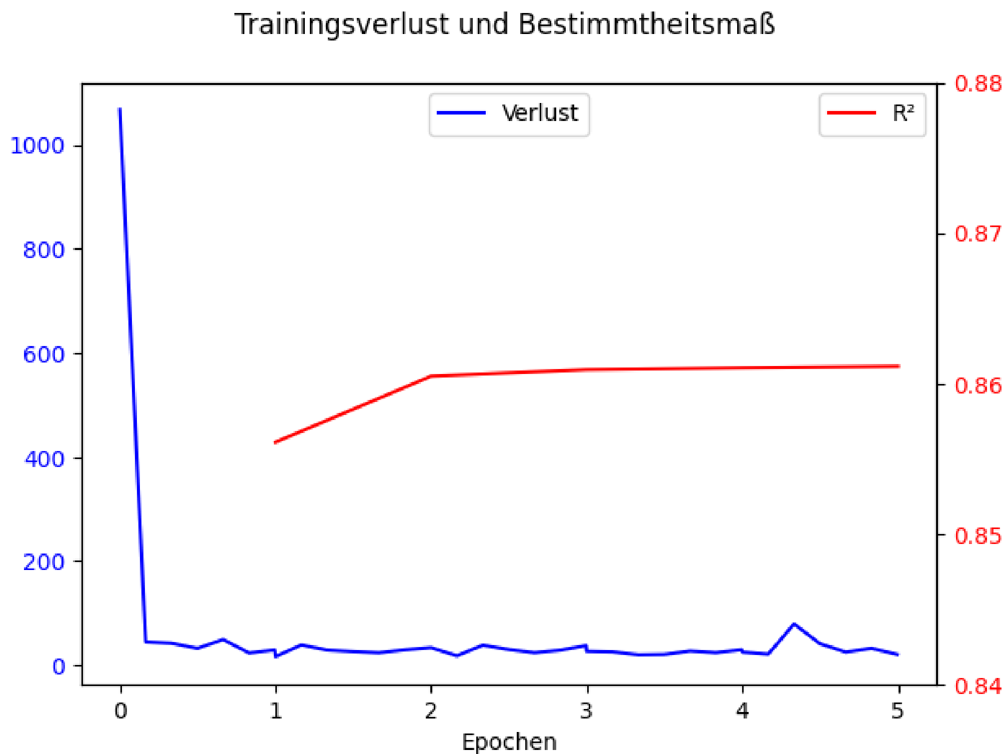

Abbildung 11: MLP-Trainingsverlauf mit Version 1

Das Training ging jeweils über fünf Epochen, nach jeder Epoche fand ein Testlauf mit den Testdaten statt. Beim MLP-Training verbesserte sich der Bestimmtheitswert im Test nach jeder Epoche, zuletzt jedoch so minimal, dass es angemessen erschien, das Training nicht über fünf Epochen hinaus fortzusetzen. Beim CNN-Training wurde der beste Bestimmtheitswert bereits nach zwei Epochen erreicht.

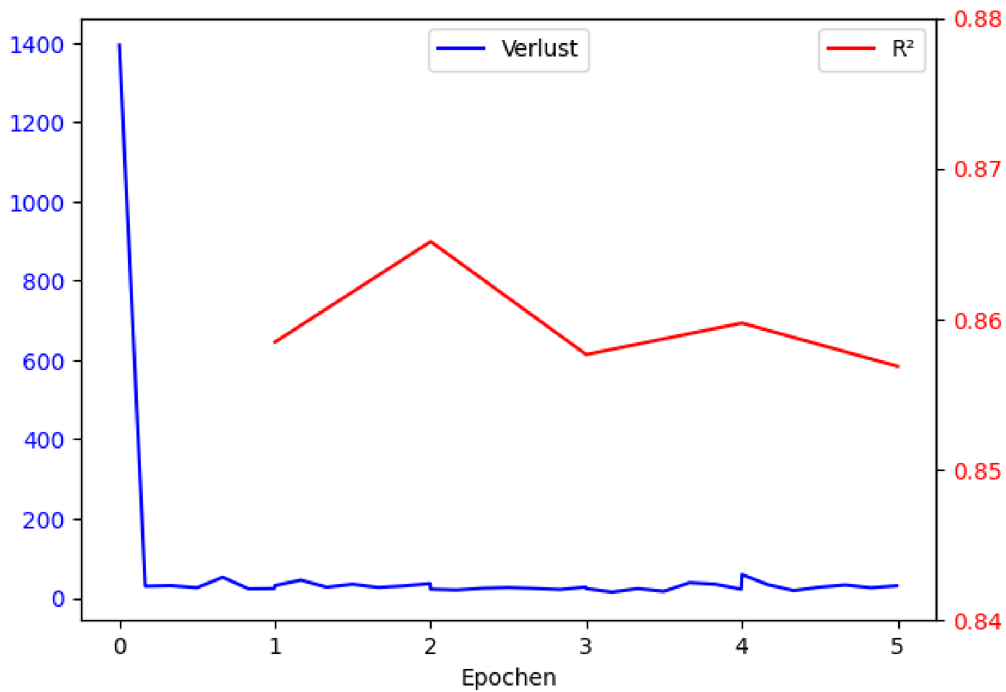

Abbildung 12: CNN-Trainingsverlauf mit Version 1

Für die drei anderen Tabellen ergeben sich erheblich schlechtere Bestimmtheitswerte. Mit einem MLP bleiben sie jeweils unter 30%, mit einem CNN knapp über 30%. Ich habe dabei verschiedene Werte für die Lernrate und verschiedene Optimierungsverfahren (Stochastic Gradient Descent, Adam, Resilient Backpropagation – dabei lieferte Stochastic Gradient Descent durchweg die besten Ergebnisse) und auch eine MLP-Variante mit erheblich mehr Neuronen (nämlich mit drei verborgenen Schichten mit 128, 96 und 64 vollvernetzten Neuronen) verwendet, was jedoch nur zu minimalen Abweichungen bei den Ergebnissen führte. Die besten Werte für die acht Tabelle-Netztyp-Kombinationen sind in Tabelle 8 wiedergegeben. Dabei bezeichnet hier und im Folgenden „Version 2“ die aus Trainingspartien der *σ-greedy*-Variante des Lernenden Spielers gegen sich selbst gewonnenen Daten, „(Version) Schwarz“ die aus Trainingspartien der *σ-greedy*-Variante des Lernenden Spielers mit den schwarzen Steinen gegen den Stochastischen Spieler gewonnenen Daten und „(Version) Weiß“ die aus Trainingspartien der *σ-greedy*-Variante des Lernenden Spielers mit den weißen Steinen gegen den Stochastischen Spieler gewonnenen Daten.

Tabelle	MLP	CNN
Version 1	86,11%	86,51%
Version 2	28,95%	30,10%
Version Schwarz	28,84%	30,04%
Version Weiß	29,04%	30,14%

Tabelle 8: Bestimmtheitsmaße der verschiedenen Tabelle-Netztyp-Kombinationen

Der erhebliche Unterschied zwischen Version 1 einerseits und den anderen drei Versionen, deren Werte sehr dicht beieinander liegen, andererseits mag überraschen. Dabei hat Version 1 in Abschnitt 4.4 tendenziell die schlechteste Spielstärke gezeigt. Insofern bietet die im folgende Abschnitt vorgenommene Überprüfung der Spielstärke der verschiedenen Versionen eines RL-Reversi-Spielers mit Netz die Möglichkeit zu untersuchen, ob für dessen Spielstärke die Anpassung des Netzes an die Tabellendaten (= das Bestimmtheitsmaß des Netztrainings) oder die „Qualität"der Tabellendaten wichtiger ist. Ferner kann festgehalten werden, dass das aufwendigere Drei-Kanal-CNN durchweg etwas, wenn auch nicht wesentlich bessere Werte liefert als das einfachere MLP.

5.3.2 Test der Spielstärke

In diesem Abschnitt werden wie beim „tabularen" RL-Reversi-Spieler der Stochastische Spieler und der Minimax-Spieler mit Tiefe 4 als Vergleichsmaßstab für die Bestimmung der Spielstärke des RL-Reversi-Spielers mit Netzwerk herangezogen. Untersucht wird zudem die Behandlung von „schwarzen" und „weißen" Stellungen (= Stellungen, in denen Schwarz beziehungsweise Weiß am Zug ist) und von strategisch äquivalenten Stellungen durch die verschiedenen Bewertungsnetze. Zudem wird ein Direktvergleich zwischen Spielern mit auf Grundlage verschiedener Tabellen trainierten Bewertungsnetzen versucht. Schließlich wird der RL-Reversi-Spieler mit Netzwerk gegen Minimax-Spieler mit größerer Tiefe getestet.

5.3.2.1 Testergebnisse gegen den Stochastischen Spieler Als ersten Spielstärketest wähle ich den Vergleich mit dem Stochastischen Spieler. Für jede Tabelle-Netztyp-Kombination spielt ein Optimierender Spieler, der seine Stellungsbewertungen aus einem Netz mit den für die jeweilige Kombination optimierten Gewichten erhält, wie in Abschnitt 4.4 jeweils 1000 Partien mit den schwarzen und mit den weißen Steinen gegen den Stochastischen Spieler.

Tabelle 9 zeigt die Ergebnisse für die vier Versionen des RL-Reversi-Spielers mit MLP. Der Unterschied zu den Ergebnissen des „tabularen" RL-Reversi-Spielers ist dramatisch: Während es in Abschnitt 4.4 schwierig war, nachzuweisen, dass der „tabulare" Spieler überhaupt etwas gelernt hat und zumindest minimal besser spielt als die reine Zufallswahl, gewinnt der MLP-Spieler in den Versionen 2, Schwarz und Weiß nahezu jede Partie gegen den Stochastischen Spieler. Die Ergebnisse für

Version 1 fallen demgegenüber etwas ab, bieten aber immer noch sehr hohe Erfolgsquoten von 93,65% und 94,80%.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	45304:18696	931	11	58	93,65%
Version 1 (weiß)	18136:45864	42	20	938	94,80%
Version 2 (schwarz)	51269:12731	999	0	1	99,90%
Version 2 (weiß)	11818:52182	2	2	996	99,70%
Version Schwarz (schwarz)	51333:12667	997	2	1	99,80%
Version Schwarz (weiß)	12381:51619	2	1	997	99,75%
Version Weiß (schwarz)	51021:12979	996	0	4	99,60%
Version Weiß (weiß)	35216:28784	9	0	991	99,10%

Tabelle 9: MLP-Testergebnisse gegen den Stochastischen Spieler

Das gleiche Gesamtbild bietet sich bei den Ergebnissen für die vier Versionen des RL-Reversi-Spielers mit CNN, wie sie in Tabelle 10 dargestellt sind. Die Erfolgsquoten liegen auch hier nahe bei 100%. Bei Version 1 liefert das CNN deutlich bessere Ergebnisse, sodass der Unterschied zwischen Version 1 und den anderen drei Versionen hier weniger ausgeprägt ist als beim Spieler mit MLP. Ansonsten ist das CNN-Ergebnis jedoch nur bei Version Weiß mit den schwarzen Steinen minimal besser als das MLP-Ergebnis, bei den sonstigen Versionen schneidet das MLP (minimal) besser ab.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	48028:15972	969	8	23	97,30%
Version 1 (weiß)	15465:48535	20	6	974	97,70%
Version 2 (schwarz)	53156:10835	998	0	2	99,80%
Version 2 (weiß)	10927:53073	5	3	992	99,35%
Version Schwarz (schwarz)	52177:11823	995	1	4	99,55%
Version Schwarz (weiß)	11867:52133	10	1	989	98,95%
Version Weiß (schwarz)	52700:11300	997	0	3	99,70%
Version Weiß (weiß)	12107:51893	16	6	978	98,10%

Tabelle 10: CNN-Testergebnisse gegen den Stochastischen Spieler

5.3.2.2 Testergebnisse gegen den Minimax-Spieler mit Tiefe 4 Für den Vergleich mit dem Minimax-Spieler mit einer Berechnungstiefe von vier Zügen lasse ich wiederum für jede Tabelle-Netztyp-Kombination einen Optimierender Spieler

jeweils 1000 Partien mit den schwarzen und mit den weißen Steinen gegen diesen Gegenspieler spielen. Die Ergebnisse für Optimierende Spieler mit MLP sind in Tabelle 11 wiedergegeben. Auch hier zeigt sich, dass der RL-Reversi-Spieler dadurch, dass die Bewertungstabelle durch ein Bewertungsnetz ersetzt wird, erheblich an Spielstärke gewinnt. Während der „tabellarische" Spieler in Abschnitt 4.4.4 Erfolgsquoten zwischen 9,76% und 36,70% erzielt und dem Minimax-Spieler somit weit unterlegen erscheint, erreichen Variante 2, Schwarz und Weiß mit den schwarzen Steinen Erfolgsquoten deutlich über 90% und mit den weißen Steinen knapp 90% beziehungsweise (Version Schwarz) knapp 80%. Mit Bewertungsnetz ist der RL-Reversi-Spieler also nun dem Minimax-Spieler erheblich überlegen. Erneut werden mit Version 1 deutlich schlechtere Ergebnisse erzielt als mit den anderen drei Versionen. Auch Version 1 gewinnt jedoch erheblich häufiger gegen den Minimax-Spieler als umgekehrt.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	37174:26826	652	29	319	66,65%
Version 1 (weiß)	31395:32605	391	23	586	59,75%
Version 2 (schwarz)	46757:17243	953	6	41	95,60%
Version 2 (weiß)	19792:44208	102	9	889	89,35%
Version Schwarz (schwarz)	46797:17203	955	5	40	95,75%
Version Schwarz (weiß)	24672:39328	199	5	796	79,85%
Version Weiß (schwarz)	46739:17261	933	8	59	93,70%
Version Weiß (weiß)	19843:44157	102	12	886	89,20%

Tabelle 11: MLP-Testergebnisse gegen den Minimax-Spieler mit Tiefe 4

Die in Tabelle 12 dargestellten Erfolgsquoten für Optimierende Spieler mit Drei-Kanal-CNN gegen den Minimax-Spieler bestätigen nochmals, dass der RL-Reversi-Spieler mit Bewertungsnetz deutlich besser spielt als der Minimax-Spieler. Hinsichtlich der Frage, ob der Spieler mit MLP oder mit CNN stärker spielt, ergibt sich hingegen ein uneinheitliches Bild: Version 1 spielt mit CNN erheblich besser als mit MLP, Version 2 hingegen spielt mit CNN etwas schlechter als mit MLP. Die Versionen Schwarz und Weiß spielen mit CNN mit den schwarzen Steinen leicht (Version Schwarz) beziehungsweise minimal (Version Weiß) schwächer als mit MLP, mit den weißen Steinen hingegen etwas (Version Weiß) beziehungsweise sehr erheblich (Version Schwarz) besser als mit MLP.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	41938:22062	842	26	132	85,50%
Version 1 (weiß)	25139:38861	221	17	762	77,05%
Version 2 (schwarz)	47362:16638	920	15	65	92,75%
Version 2 (weiß)	20630:43370	149	5	846	84,85%
Version Schwarz (schwarz)	47149:16851	926	15	59	93,35%
Version Schwarz (weiß)	16687:47313	72	6	922	92,50%
Version Weiß (schwarz)	47165:16835	919	8	73	92,30%
Version Weiß (weiß)	15589:48411	54	4	942	94,40%

Tabelle 12: CNN-Testergebnisse gegen den Minimax-Spieler mit Tiefe 4

5.3.2.3 „Schwarze" und „weiße" Stellungen Bemerkenswert erscheint, dass die Versionen Schwarz und Weiß auch mit den Steinen der Farbe, die der Lernende Spieler beim Auffüllen ihrer Tabelle nicht gespielt hat, gute Erfolgsquoten erreichen. Bei drei der vier betrachteten Kombinationen von Gegenspieler und Netztyp spielt Version Schwarz mit den weißen Steinen nicht oder nur knapp weniger erfolgreich als mit den schwarzen Steinen, die Version Weiß spielt mit den schwarzen Steinen sogar in drei von vier Fällen erfolgreicher als mit den weißen Steinen.

Die „tabularen" Spieler spielen mit den Steinen der jeweils „falschen" Farbe hingegen genauso schlecht wie der Stochastische Spieler, da sie in ihrer Tabelle nur sehr selten die Stellungen finden, deren Bewertung sie benötigen, und daher ihre Züge meist rein zufällig wählen müssen (siehe die Testergebnisse in Tabelle 13). In der Tabelle der Version Schwarz finden sich beispielsweise keine Stellungen mit sechs, acht oder zehn Steinen, da in allen diesen Stellungen Schwarz am Zug ist und, wie in Abschnitt 4.1 beschrieben, ein mit den schwarzen Steinen spielender Lernender Spieler nur Bewertungen für Stellungen in seine Tabelle einträgt, in denen Weiß am Zug ist. Grundsätzlich gibt es zwar Stellungen, die sowohl mit Weiß am Zug als auch farbenvertauscht mit Schwarz am Zug vorkommen können (sodass beide Varianten in einer Matrixdarstellung, in der die Steine des Spielers, der am Zug ist, mit 1 und die des Gegenspielers mit -1 dargestellt werden, genau gleich aussehen). Solche Stellungen treten aber selten auf: Bei einer von mir durchgeführten Simulation von einer Million Partien mit zufälliger Zugwahl wurde 9.021.506 Mal eine Stellung erreicht, die vorher schon einmal mit dem gleichen Spieler am Zug erreicht worden war, aber nur 255 Mal eine Stellung, deren farbenvertauschte Variante vorher schon einmal erreicht worden war. Von den insgesamt 51.443.089 verschiedenen Stellungen in den Tabellen der Versionen Schwarz und Weiß kommen entsprechend auch nur 146 in beiden Tabellen vor.

Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
„Tabularer“ Spieler mit weißer Tabelle und schwarzen Steinen				
31586:32414	472	32	496	48,80%
„Tabularer“ Spieler mit schwarzer Tabelle und weißen Steinen				
31999:32001	479	37	484	50,25%

Tabelle 13: Ergebnisse des „tabularen“ RL-Reversi-Spielers mit der „falschen“ Farbe gegen den Stochastischen Spieler (Tiefe 4)

Die guten Ergebnisse der Versionen Schwarz und Weiß mit den Steinen der „anderen“ Farbe könnte man dahingehend interpretieren, dass es den Netzwerken im Training mit den Bewertungen aus den Tabellen gelungen ist, bewertungsrelevante Merkmale der Stellungen zu identifizieren, die unabhängig davon sind, ob Schwarz oder Weiß in der jeweiligen Stellung am Zug ist. Damit können beispielsweise auch brauchbare Bewertungen für Stellungen mit sechs, acht oder zehn Steinen ermittelt werden, selbst wenn keine derartigen Stellungen in den Trainingsdaten enthalten waren.

5.3.2.4 Strategisch äquivalente Stellungen Wenn ein Bewertungsnetz in der Lage ist, die wesentlichen bewertungsrelevanten Merkmale einer Stellung zu identifizieren, so sollte man erwarten, dass dieses Netz strategisch äquivalente Stellungen gleich bewertet (vergleiche Abschnitt 3.2). Dann sollte es wiederum nicht nötig sein, eine zu bewertende Stellung zuerst in ihre kanonische Form umzuwandeln, bevor sie in dieses Bewertungsnetz eingegeben wird. Um zu prüfen, ob die in Abschnitt 5.3.1 trainierten Netze strategisch äquivalente Stellungen „erkennen“ (= gleich bewerten), habe ich die Funktion `bewertung_geben` entsprechend so abgeändert, dass eine eingegebene Stellung ohne Kanonisierung bewertet wird. Mit dieser Abwandlung habe ich den Optimierenden Spieler wiederum für jede Tabelle-Netztyp-Kombination jeweils 1000 Partien mit den schwarzen und den weißen Steinen gegen den Minimax-Spieler mit Tiefe 4 spielen lassen. Tabelle 14 zeigt die (enttäuschenden) Ergebnisse für MLP-Netze: Die Erfolgsquoten sind um etwa 20 (Version Weiß mit den weißen Steinen, Version 1, Version 2) beziehungsweise etwa 10 (Version Weiß mit den schwarzen Steinen, Version Schwarz mit den schwarzen Steinen) Prozentpunkte niedriger als mit Kanonisierung. Die einzige Ausnahme bildet Version Schwarz mit den weißen Steinen. Offenbar sind die Netze nur recht unvollkommen in der Lage, strategische Äquivalenz nachzuvollziehen.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	29929:34071	416	34	550	43,30%
Version 1 (weiß)	37116:26884	569	30	401	41,60%
Version 2 (schwarz)	36786:27214	732	13	255	73,85%
Version 2 (weiß)	29416:34584	308	10	682	68,70%
Version Schwarz (schwarz)	43130:20870	861	17	122	86,95%
Version Schwarz (weiß)	22832:41168	160	11	829	83,45%
Version Weiß (schwarz)	42549:21451	837	14	149	84,40%
Version Weiß (weiß)	27623:36377	289	22	689	70,00%

Tabelle 14: MLP-Testergebnisse gegen den Minimax-Spieler mit Tiefe 4 ohne Kanonisierung

Tabelle 15 zeigt, dass der gleiche Befund, wenn auch etwas weniger ausgeprägt, auch für Optimierende Spieler mit CNN gilt.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	35514	634	33	333	65,05%
Version 1 (weiß)	31449:32560	381	29	590	60,45%
Version 2 (schwarz)	43251:20749	834	15	151	84,15%
Version 2 (weiß)	20519:43481	146	10	844	84,90%
Version Schwarz (schwarz)	42493:21507	859	14	127	86,60%
Version Schwarz (weiß)	23159:40841	181	10	809	81,40%
Version Weiß (schwarz)	45314:18686	890	15	95	89,75%
Version Weiß (weiß)	23296:40704	200	14	786	79,30%

Tabelle 15: CNN-Testergebnisse gegen den Minimax-Spieler mit Tiefe 4 ohne Kanonisierung

Eine naheliegende Idee, wie man ein Netzwerk so trainieren kann, dass es strategische Äquivalenz erkennt, ist, die Trainingsdaten – die ja nur kanonische Stellungen enthalten – dadurch anzureichern, dass man für jede im Datenbestand enthaltene kanonische Stellung auch (alle) strategisch äquivalenten nicht-kanonischen Stellungen mit der gleichen Bewertung zum Datenbestand hinzufügt. Ein mit diesem erweiterten Datenbestand trainiertes Netzwerk sollte strategisch äquivalente Stellungen (annähernd) gleich bewerten. Ob sich aus dieser logisch konsistenteren Stellungsbewertung aber auch eine höhere Spielstärke ergibt, bliebe zu prüfen.

5.3.2.5 Direktvergleich zwischen Version 1 und Version 2

Bisher habe ich die Spielstärke der verschiedenen Versionen nur indirekt verglichen, indem sie jeweils am Maßstab des Minimax-Spielers beziehungsweise des Stochastischen Spielers gemessen wurden. Stattdessen könnte man auf die Idee kommen, sie direkt gegeneinander spielen zu lassen, um festzustellen, welche Version stärker spielt. Der Versuch, sie analog zum bisherigen Vorgehen einfach 1000 Partien gegeneinander spielen zu lassen, scheitert jedoch daran, dass ein Bewertungsnetz für unterschiedliche Stellungen fast immer unterschiedliche Bewertungen ausgibt. Während der RL-Reversi-Spieler mit Tabelle und der Minimax-Spieler in vielen Stellungen mehrere mögliche Züge als gleich gut bewerten und dann zufällig einen von ihnen auswählen, ermittelt ein RL-Reversi-Spieler mit Bewertungsnetz in nahezu jeder Stellung einen eindeutig besten Zug und spielt diesen dann deterministisch. Daher ergibt sich zwischen zwei RL-Reversi-Spielern mit Bewertungsnetz aus der Grundstellung heraus immer der gleiche Partieverlauf. Wenn man Version 1 mit den schwarzen Steinen gegen Version 2 mit den weißen Steinen (beide mit MLP) spielen lässt, gewinnt Version 2 immer mit 40:24. Wenn Version 2 mit den schwarzen Steinen spielt, ergibt sich immer ein Unentschieden. Diese zwei Partieverläufe erlauben offensichtlich noch keine zuverlässige Aussage über die relative Spielstärke der beiden Versionen.

Eine Möglichkeit, zu einem aussagekräftigen Vergleich der beiden Versionen zu kommen, besteht darin, nicht nur Partien von der Grundstellung aus zu betrachten, sondern die beiden Versionen ausgehend von einer hinreichend großen Anzahl verschiedener, ausgeglichener Stellungen (d. h. Stellungen, in denen kein Spieler einen Vorteil gegenüber dem anderen hat) gegeneinander spielen zu lassen. Derartige Stellungen lassen sich grundsätzlich beispielsweise mithilfe der in Abschnitt 3.3 erwähnten Reversi-Programme generieren. Wegen des damit verbundenen Zeitaufwands habe ich jedoch hierauf verzichtet.

Eine weitere Möglich besteht darin, den deterministischen Charakter des RL-Reversi-Spielers mit Netz abzuschwächen, indem die vom Netz ausgegebenen Stellungsbewertungen auf wenige Nachkommastellen gerundet werden. Dies erscheint auch inhaltlich angemessen, da minimale Bewertungsunterschiede angesichts der unvollkommenen Anpassung der Netzausgaben an die ursprünglichen Tabellendaten als unerheblich anzusehen sind. Wenn sich die Bewertung zweier Stellungen nur minimal unterscheidet, sollten sie somit als gleichwertig beurteilt werden.

Ohne Rundung				
Version 1 – Version 2	24000:40000	0	0	1000
Version 2 – Version 1	32000:32000	0	1000	0
Auf drei Nachkommastellen gerundet				
Version 1 – Version 2	21353:42647	0	0	1000
Version 2 – Version 1	34982:29018	497	503	0
Auf zwei Nachkommastellen gerundet				
Version 1 – Version 2	24289:39711	189	5	806
Version 2 – Version 1	43392:20608	964	21	15

Tabelle 16: Direktvergleich Version 1 gegen Version 2 mit MLP

Die Direktvergleichsergebnisse für ein MLP mit Rundung sind in Tabelle 16 dargestellt. Bei Rundung auf drei Nachkommastellen ergibt sich weiterhin keine große Vielfalt an Partieverläufen. Mit den weißen Steinen gewinnt Version 2 alle Partien, wobei das Punkteergebnis anzeigt, dass es zumindest zwei verschiedene Partieverläufe geben muss. Wenn Version 2 mit den schwarzen Steinen spielt, lässt das Punkteergebnis die Vermutung zu, dass es neben den 503 Partien, die unentschieden enden, 497 Partien gibt, die Version 2 – mutmaßlich immer mit dem gleichen Verlauf – mit 38:26 gewinnt. Bei Rundung der Netzbewertungen auf zwei Nachkommastellen wirkt das Gesamttestergebnis etwas vielfältiger. Es muss für beide Farbzuweisungen jeweils mindestens drei verschiedene Partieverläufe geben. Ob es aber hinreichend viele Partieverläufe gibt, um tatsächlich zu einer aussagekräftigen Spielstärkebewertung zu kommen, bleibt offen. Erwartungsgemäß spricht das Testergebnis dabei für eine deutliche Überlegenheit von Version 2 gegenüber Version 1.

Ohne Rundung				
Version 1 – Version 2	43000:21000	1000	0	0
Version 2 – Version 1	52000:12000	1000	0	0
Auf drei Nachkommastellen gerundet				
Version 1 – Version 2	37045:26955	1000	0	0
Version 2 – Version 1	58012:5988	1000	0	0
Auf zwei Nachkommastellen gerundet				
Version 1 – Version 2	27021:36979	257	50	693
Version 2 – Version 1	38292:25708	613	0	387

Tabelle 17: Direktvergleich Version 1 gegen Version 2 mit CNN

Tabelle 17 zeigt die Direktvergleichsergebnisse für ein CNN mit Rundung. Hier deutet erst das Testergebnis bei einer Rundung auf zwei Nachkommastellen auf eine Überlegenheit von Version 2 hin, die zudem deutlich weniger ausgeprägt erscheint als bei den Spielern mit MLP. Dabei bestehen die gleichen Zweifel an der Aussagefähigkeit dieser Vergleichsdaten wie im MLP-Fall.

5.3.2.6 Testergebnisse gegen den Minimax-Spieler mit größerer Tiefe Die Erfolgsquoten des RL-Reversi-Spielers mit Netz gegen den Stochastischen Spieler liegen sehr nahe bei 100%, gegen den Minimax-Spieler mit Tiefe 4 liegen sie nicht wesentlich niedriger. Beide Vergleichsmaßstäbe zeigen deutlich die Überlegenheit des RL-Reversi-Spielers mit Netz gegenüber dem „tabularen" RL-Reversi-Spieler. Da Werte über 100% nicht möglich sind, sind sie aber wenig geeignet, um eventuelle weitere Verbesserungen der Spielstärke aufzuzeigen, und auch vergleichende Aussagen über die Vorzüge verschiedener Varianten des RL-Reversi-Spielers mit Netz sind schwierig, wenn alle Erfolgsquoten dicht beieinander liegen. Dafür wird ein spielstärkerer Vergleichsmaßstab als der Minimax-Spieler mit Tiefe 4 benötigt.

Ein naheliegender Kandidat für einen stärkeren Vergleichsmaßstab ist ein Minimax-Spieler, der mehr als vier Züge voraus berechnet. Testergebnisse des RL-Reversi-Spielers mit MLP gegen einen Minimax-Spieler mit Tiefe 6 sind in Tabelle 18 dargestellt. Erwartungsgemäß sind die Erfolgsquoten deutlich niedriger als gegen den Minimax-Spieler mit Tiefe 4, wobei Version 1 nun noch deutlich stärker hinter den anderen drei Versionen zurückbleibt und nur noch Erfolgsquoten von unter 50% erreicht. Versionen 2, Schwarz und Weiß sind deutlich stärker als der Minimax-Spieler mit Tiefe 6. Anders als in Abschnitt 5.3.2.2 gibt es für die anderen drei Versionen nun eine eindeutige Reihenfolge, allerdings mit sehr geringen Abständen: Version 2 erreicht minimal bessere Quoten als Version Schwarz, die ihrerseits leicht besser ist als Version Weiß.

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgsquote
Version 1 (schwarz)	31448:32552	463	33	504	47,95%
Version 1 (weiß)	33909:30091	541	42	417	43,80%
Version 2 (schwarz)	42528:21472	857	22	121	86,80%
Version 2 (weiß)	24026:39974	194	24	782	79,40%
Version Schwarz (schwarz)	42475:21525	843	19	138	85,25%
Version Schwarz (weiß)	24746:39254	209	10	781	78,60%
Version Weiß (schwarz)	41236:22764	820	18	162	82,90%
Version Weiß (weiß)	25880:38120	232	23	745	75,65%

Tabelle 18: MLP-Testergebnisse gegen den Minimax-Spieler (Tiefe 6)

Die Erfolgsquoten für Spieler mit CNN sind in Tabelle 19 wiedergegeben. Wie zu erwarten sind auch hier die Erfolgsquoten deutlich niedriger als gegen den Minimax-Spieler mit Tiefe 4, die qualitativen Ergebnisse entsprechen im Wesentlichen denen aus Abschnitt 5.3.2.2: während Version 1 mit CNN besser abschneidet als mit MLP, gilt bei den drei anderen Versionen meist das Gegenteil. Version 1 spielt am schwächsten, wenn auch knapp besser als der Minimax-Spieler. Zwischen den Versionen 2, Schwarz und Weiß gibt es keine eindeutige Reihenfolge (vor allem, weil Version 2 mit den weißen Steinen auffällig schwach abschneidet).

Optimierender Spieler (mit Farbe)	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Version 1 (schwarz)	32453:31547	554	34	412	57,10%
Version 1 (weiß)	30830:33170	414	36	550	56,80%
Version 2 (schwarz)	41781:22219	820	10	170	82,50%
Version 2 (weiß)	29584:34416	333	16	651	65,90%
Version Schwarz (schwarz)	40891:23109	809	15	176	81,65%
Version Schwarz (weiß)	22400:41600	154	19	827	83,65%
Version Weiß (schwarz)	39984:24016	786	24	190	79,80%
Version Weiß (weiß)	25864:38136	267	25	708	72,05%

Tabelle 19: CNN-Testergebnisse gegen den Minimax-Spieler (Tiefe 6)

Wenn man die Tiefe des Minimax-Spielers auf 8 Züge erweitert, ergeben sich für Version 2 mit MLP die in Tabelle 20 dargestellten Erfolgsquoten. Der RL-Reversi-Spieler mit Netz ist wiederum signifikant besser als der Vergleichsmaßstab, aber erwartungsgemäß in geringerem Ausmaß als in Tabelle 18 oder 11. Auffällig ist, dass die Quoten mit den schwarzen und den weißen Steinen fast identisch sind. Wie schon in Abschnitt 4.2 ausgeführt, steigt der Rechenaufwand des Minimax-Spielers exponentiell mit der Vorschautiefe. Deshalb wurden die Tests nur mit jeweils 100 statt wie in den vorhergehenden Abschnitten mit 1.000 Partien durchgeführt. Zur Berechnung der in Tabelle 20 wiedergegebenen Quoten benötigt die mir zur Verfügung stehende Rechenanlage gleichwohl 16 ½ Stunden. Ich habe daher darauf verzichtet, die Berechnungen auch für die anderen Versionen und für Spieler mit CNN durchzuführen.

Optimierender Spieler mit Farbe	Summe Punkte	Anzahl Siege Schwarz	Anzahl Remis	Anzahl Siege Weiß	Erfolgs-quote
Schwarz	3402:2998	64	3	33	65,50%
Weiß	3034:3366	33	2	65	66,00%

Tabelle 20: MLP-Testergebnisse für Version 2 gegen den Minimax-Spieler (Tiefe 8)

Abb. 13 bietet einen Überblick über die Ergebnisse der verschiedenen Versionen gegen die Minimax-Spieler mit unterschiedlichen Tiefen.

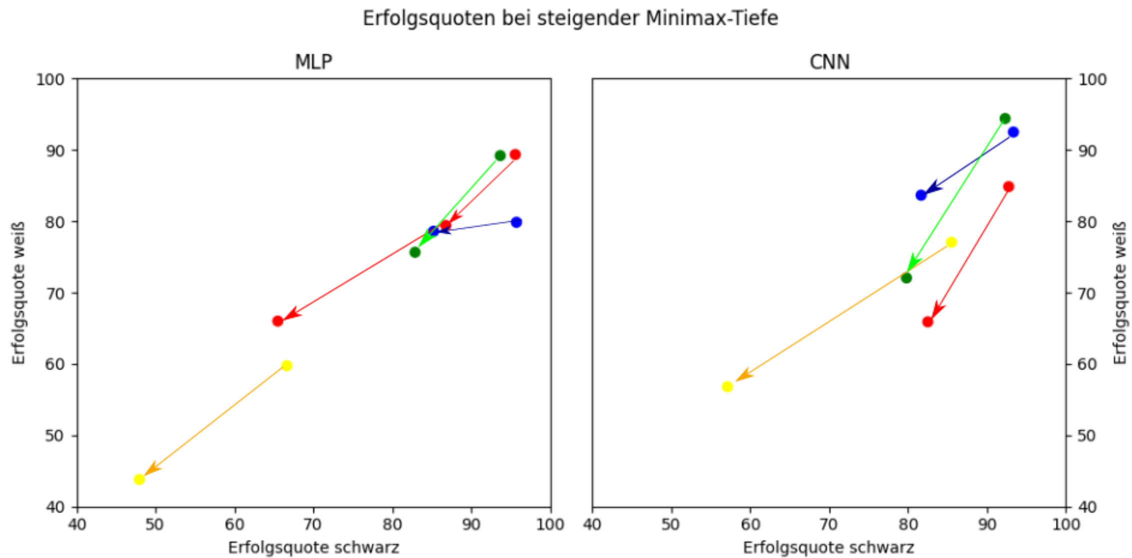

Abbildung 13: Erfolgsquoten der Version 1 (gelb), zwei (rot), Schwarz (blau) und Weiß (grün) gegen den Minimax-Spieler mit Tiefe 4 und 6 sowie (nur Version 2 mit MLP) 8. Die Pfeile zeigen in Richtung steigender Tiefe.

5.4 Zwischenfazit

Der Einsatz eines mit den Tabellenwerten trainierten Netzwerks anstelle einer Tabelle führt zu einer sehr erheblichen Steigerung der Spielstärke des Optimierenden Spielers. Dies bestätigt, dass das Hauptproblem für den Algorithmus aus Abschnitt 4 die große Anzahl möglicher Reversi-Stellungen ist, aufgrund derer die Bewertungstabelle auch nach einer Million Partien noch viel zu klein ist. Das Netzwerk löst dieses Problem, indem es für jede neue Stellung eine Bewertung abschätzt, auch wenn dieses Abschätzung möglicherweise weniger genau ist als die Bewertung aus einer Tabelle. Insgesamt zeigt sich in der experimentellen Überprüfung eine deutlich bessere Spielstärke als beim Minimax-Spieler mit Tiefe 8.

Bemerkenswert ist zudem, dass nicht notwendigerweise das Netzwerk die beste Spielstärke liefert, bei dem sich im Training die besten Anpassungswerte an die Tabellendaten ergeben haben. Tendenziell ergibt sich mit einem Netz, das mit Tabellendaten trainiert ist, die im „tabularen" RL-Reversi-Spieler eine bessere Spielstärke ergeben, auch beim RL-Reversi-Spieler mit Netzwerk die bessere Spielstärke, mit erstaunlich geringer Abhängigkeit von der Anpassungsqualität.

Die Verwendung eines vergleichsweise komplexen CNN bringt zwar im Vergleich zu einem einfachen MLP (leicht) bessere Anpassungswerte an die Tabellendaten, aber keinen eindeutigen Vorteil bei der Spielstärke. Es gibt unter den untersuchten Kombinationen von Lernversion und Netzarchitektur keinen eindeutigen Sieger. Gegen den Minimax-Spieler mit Tiefe 6, also den stärksten Vergleichsmaßstab, gegen den alle Kombinationen getestet werden konnten, schneidet mit Weiß Version

2 mit MLP am besten ab, mit Schwarz Version Schwarz mit CNN.

Festzuhalten ist wiederum, dass es eine große Anzahl an Trainings- und Architekturparametern gibt, für die bei der experimentellen Überprüfung bei Weitem nicht alle Werte und Wertekombinationen ausprobiert werden konnten. Es ist also nicht auszuschließen, dass bei (noch) umfangreicheren Experimenten eine noch höhere Spielstärke hätte erreicht werden können.

6 Implementierung eines tiefen RL-Reversi-Spielers

Nachdem in Abschnitt 4 die Sammlung von Erfahrungswerten in einer Tabelle und in Abschnitt 5 die Auswertung dieser Erfahrung mithilfe eines neuronalen Netzwerks als zwei separate Algorithmen entwickelt wurden, erscheint es als logischer nächster Schritt, beide Tätigkeiten in einem Verfahren zusammenzuführen. Ein Algorithmus, in dem wiederholt neue Spielerfahrungen gesammelt werden, auf deren Grundlage dann jeweils anschließend ein Bewertungsnetzwerk aktualisiert wird, wird als „tiefer wertbasierter RL-Algorithmus“ bezeichnet (siehe [15], S. 76). In diesem Abschnitt wird ein solcher Algorithmus auf Basis der bisher betrachteten Verfahren hergeleitet und experimentell überprüft.

6.1 Abwechselnde Integration von Erfahrungssammlung und Netztraining in einen Algorithmus

Für einen tiefen wertbasierten RL-Algorithmus kann weitgehend der gleiche Lernende Spieler wie in Abschnitt 4 verwendet werden. Stellungsbewertungen werden dabei jedoch nicht der Tabelle entnommen, sondern mit einem Bewertungsnetz ermittelt. Ein weiterer Unterschied ergibt sich nach Abschluss einer Partie bei der Verwendung der mit dem Partieergebnis bewerteten Stellungen: Diese werden nicht unmittelbar zur Ergänzung und Aktualisierung der Bewertungsfunktion genutzt, sondern in einem Zwischenspeicher (*replay buffer*, vergleiche [15], S. 79f) zwischengespeichert. Da eine Partie zwischen 10 und 70 Zügen lang ist (siehe Abschnitt 3.2), ergeben sich aus einer einzelnen Partie 5 bis 35 Stellungen mit jeweils der gleichen Bewertung, falls nur für eine Farbe gelernt wird, oder 10 bis 70 Stellungen mit jeweils einer von zwei komplementären (= sich zu 64 ergänzenden) Bewertungen, falls für beide Farben gelernt wird. Die Daten einer einzelnen Partie sind also hochgradig korreliert und ergeben keine geeignete Grundlage für eine Aktualisierung der Netzgewichte, da ein Netz, das mit Daten aktualisiert wird, die alle den gleichen Zielwert (= Bewertung) haben, für jegliche Eingabe auf diesen Zielwert hinsteuern wird (siehe [15], S. 77). Um dies zu umgehen, werden die Netzgewichte daher jeweils erst nach einer bestimmten, hinreichend großen Anzahl Partien mithilfe einer Zufallsstichprobe der Daten aus dem Zwischenspeicher weitertrainiert (vergleiche [12], S. 4f). Für das Bewertungsnetz können dabei die gleichen Architekturen wie in Abschnitt 5 verwendet werden. Die *σ-greedy* Variante dieses Algorithmus ist schematisch in Abb. 14 dargestellt.

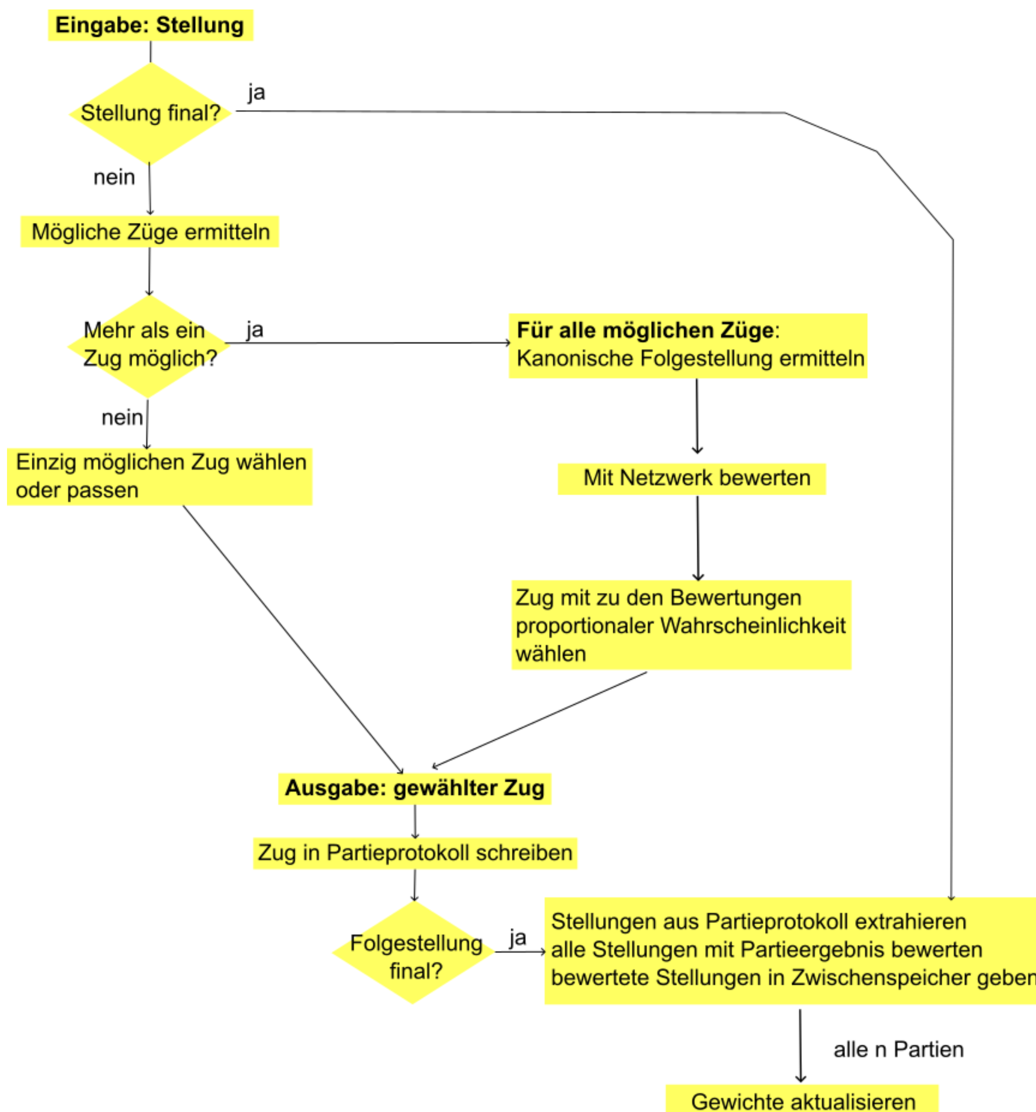

Abbildung 14: Algorithmus des tiefen Lernenden Spielers

Um sicherzustellen, dass schon bei der ersten Gewichtsanpassung hinreichend viele (und hinreichend diverse) Bewertungen im Zwischenspeicher vorliegen, bietet sich an, anfangs eine größere Anzahl Partien zur Auffüllung des Zwischenspeichers spielen zu lassen. Der Gesamtablauf lässt sich dann wie folgt beschreiben: Nach der anfänglichen Zwischenspeicherauffüllung wird eine festgelegte Anzahl Lernzyklen durchgeführt, die jeweils aus einer Erfahrungssammelphase, einer Trainingsphase und einer Testphase bestehen. In der Erfahrungssammelphase spielt der Lernende Spieler eine festgelegte Anzahl Partien gegen sich selbst oder gegen einen nicht lernenden Gegenspieler und die dabei erhaltenen Stellungsbewertungen werden im Zwischenspeicher abgelegt. In der Trainingsphase werden die Netzgewichte mit ei-

ner Stichprobe aus dem Zwischenspeicher (weiter)trainiert. In der Testphase, die eventuell nicht in jedem Zyklus durchgeführt wird, wird die bislang erreichte Spielstärke durch eine festgelegte Anzahl von Partien des Optimierenden Spielers mit dem aktualisierten Bewertungsnetz gegen einen Vergleichsspieler ermittelt. Anders als in Abschnitt 5.3.1 findet kein expliziter Test der Anpassungsgüte der Netzwerkausgaben bezüglich der Trainingsdaten statt.

Bei diesem Ablauf gibt es eine große Zahl an Parametern, für die geeignete Werte zu finden sind: die Anzahl an Stellungen, mit der der Zwischenspeicher anfänglich aufgefüllt wird, die Partieanzahl in der Erfahrungssammelphase, die Größe der Stichprobe und die Anzahl der Durchgänge durch die Stichprobendaten in der Trainingsphase, die Anzahl der Testpartien, die Anzahl der Lernzyklen. Zudem ist die Größe des Zwischenspeichers festzulegen. In Abschnitt 4 wurde stillschweigende unterstellt, dass die Bewertungstabelle unbegrenzt erweitert werden kann. Ein Zwischenspeicher hat jedoch typischerweise eine festgelegte Größe (vergleiche wiederum [12], S. 4f). Sobald mehr Stellungen vorliegen, als in den Zwischenspeicher passen, werden die ältesten Stellungsbewertungen gelöscht.

Das Vorgehen aus den Abschnitten 4 und 5 lässt sich als Variante des gerade beschriebenen Gesamtablaufs auffassen, bei dem es nur einen Lernzyklus gibt, in dem in der Erfahrungsammelphase eine Million Partien gespielt werden, der Zwischenspeicher bis zu 52 Millionen Stellungen aufnehmen kann, die Stichprobe in der Trainingsphase alle Stellungsbewertungen aus dem Zwischenspeicher enthält und das Training fünf volle Durchläufe durch die Stichprobe umfasst. Es bleibt jedoch ein Unterschied zwischen dem Vorgehen aus den Abschnitten 4 und 5 und dem tiefen RL-Algorithmus: In der Tabelle aus Abschnitt 4 kommt jede Stellung nur ein mal vor und wird mit dem Durchschnitt der einschlägigen Partieergebnisse bewertet. Im Zwischenspeicher kann jedoch ein und die gleiche Stellung mehrfach mit unterschiedlichen Bewertungen (= Partieergebnissen) gespeichert sein.

Der hier dargestellte Algorithmus ist an den Deep Q-Net Algorithmus (DQN) aus [12] angelehnt, den ersten erfolgreich in einem Spielkontext eingesetzten tiefen wertbasierten RL-Algorithmus. DQN wurde jedoch nicht für Brettspiele, sondern für Ein-Personen-Videospiele entwickelt, bei denen es nach jeder Aktion sofort eine nichttriviale Belohnung (= Änderung des Punktestands) für den Spieler gibt. Sei wie in Abschnitt 2.1 S_t der Zustand zum Zeitpunkt t, A_t die gewählte Aktion und R_{t+1} die nach Ausführung der Aktion erhaltene Belohnung, und sei $\hat{q}(s,a)$ die Ausgabe des Bewertungsnetzes für das Zustands-Aktions-Paar (s,a). Dann wird in [12] das Paar (S_t, A_t) im Zeitpunkt $t+1$ mit $R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a)$ bewertet (vergleiche den Term ganz rechts in Formel 3 in Abschnitt 2.1). Mit diese Bewertung wird (S_t, A_t) in den Zwischenspeicher gegeben und danach wird eine Minibatch-Stichprobe aus dem Zwischenspeicher gezogen und ein Gewichtsanpassungsschritt für das Bewertungsnetz $\hat{q}$ durchgeführt. Bei DQN erfolgt also nach jeder einzelnen Aktionswahl unmittelbar ein einzelner Gewichtsanpassungsschritt, während im hier entworfenen Brettspielalgorithmus erst nach einer Mehrzahl von Partien mit jeweils vielen Zügen Gewichtsanpassungsschritte vorgenommen werden. Insofern könnte man

DQN, wie schon das Vorgehen in den Abschnitten 4 und 5, als „Extremvariante“ des Brettspielalgorithmus auffassen, und zwar als eine,bei der es sehr viele Lernzyklen gibt, in deren Erfahrungsammelphase jeweils nur ein Zug gespielt wird und in deren Trainingsphase mit einer sehr kleinen Stichprobe jeweils nur ein Gewichtsanpassungsschritt ausgeführt wird. Der hier entwickelte tiefe Algorithmus liegt in diesem Sinne zwischen DQN und dem RL-Algorithmus mit aus Tabellendaten generiertem Netz.

Festzuhalten ist zudem, dass bei DQN das Bewertungsnetz auf ein Ziel hin trainiert wird, das wiederum mithilfe des Bewertungsnetzes berechnet wurde (nämlich $R_{t+1} + \gamma \max_a \hat{q}(S_{t+1}, a)$). Mit jeder Anpassung der Gewichte des Bewertungsnetzes verschiebt sich daher auch das Trainingsziel, es wird also auf ein „bewegliches Ziel“ hin trainiert. Dies führt zu besonderen Herausforderungen für die Konvergenz des Trainingsprozesses (siehe [23], S. 264; [15], S. 77f). In der Brettspielvariante ergibt sich das Trainingsziel nur aus dem beobachteten Spielergebnis. Auf den ersten Blick sollte dies die besonderen Konvergenzprobleme tiefer RL-Algorithmen vermeiden. Allerdings hängt das Partieergebnis, das sich aus einer bestimmten Stellung heraus ergibt, sofern es sich nicht um eine Endstellung handelt, davon ab, wie in den folgenden Stellungen gespielt wird. Bei einem lernenden Spieler und einem lernenden Gegenspieler ergibt sich damit ebenfalls ein „bewegliches Ziel“. Konvergenzprobleme sind somit auch hier nicht auszuschließen.

Ein weiterer Unterschied zu DQN besteht darin, dass das Netzwerk des hier konstruierten Algorithmus immer nur eine Folgestellung (also eine Aktion) bewertet und daher bei jeder Zugauswahl so oft zum Einsatz kommt, wie es jeweils mögliche Züge gibt. Bei DQN hingegen gibt das Netzwerk jeweils bei Eingabe des aktuellen Zustands eine Bewertung für alle Aktionen aus. Dies ist dadurch begründet, dass es in den Videospielen in [12] relativ wenige Aktionen gibt und dass jede dieser Aktionen in allen Zuständen möglich ist. Bei Reversi gibt es hingegen vergleichsweise viele potentielle Züge (nämlich 61, entsprechend der Anzahl der im Partieverlauf zu besetzenden Felder plus eins für „Passen“), von denen jedoch in jeder Stellung nur wenige tatsächlich möglich sind. Da hier entwickelte Vorgehen mit jeweils einem Netzaufruf pro möglichem Zug ähnelt in dieser Hinsicht einem anderen bekannten tiefen wertbasierten Ansatz, dem Fitted Q-Iteration Algorithmus (siehe [16]).

6.2 Implementierungsdetails

Die Spielerklassen aus spieler.py und die Klasse Partieumgebung können unverändert verwendet werden. Lernende und Optimierende Spieler werden jeweils mit einem Bewertungsnetz-Objekt verbunden. Damit dieses dann auch mit der Partieumgebung interagieren kann, muss die Klasse Bewertungsnetz um eine Methode `bewertung_aktualisieren` ergänzt werden. Genau wie beim „tabularen“ RL-Algorithmus ruft die Partieumgebung diese Methode nach jeder abgeschlossenen Partie auf. Das Bewertungsnetz bewertet dann alle Stellungen aus der Partie mit dem Partieergebnis für die jeweilige Farbe und legt sie als Objekt der in bewer-

tungsgeber.py definierten Klasse `BewertungsDaten` im Zwischenspeicher ab.

Für die Organisation der Stellungsdaten wird die Bibliothek tensordict genutzt (siehe [3]), indem von deren Klasse `tensorclass` in bewertungsgeber.py eine Klasse `BewertungsDaten` abgeleitet wird, mit der in der Bewertungsnetz-Methode `bewertung_aktualisieren` Stellungen und dazugehörige Bewertungen gekoppelt werden, bevor sie als BewertungsDaten-Objekt an den Zwischenspeicher weitergegeben werden.

Die Klasse Bewertungsnetz erhält ein zusätzliches Attribut `zwischenspeicher` mit einem Verweis auf den Zwischenspeicher, an den die bewerteten Stellungen übergeben werden. Zudem erhält sie ein Attribut `transformation`, das bestimmt, ob und wie Stellungen vor der Zwischenspeicherung transformiert werden sollen. Standardmäßig ist dieses Attribut mit der Funktion `als_kanonische_stellung` belegt, Stellungen werden also in kanonischer Form gespeichert, sofern keine andere Transformation festgelegt wird (siehe Abschnitt 6.3.3).

Für die Implementierung des Zwischenspeichers wird die Klasse `data.ReplayBuffer` aus der PyTorch-Ergänzung `torchrl` verwendet (siehe [3]). In den Skripten tiefes_lernskript.py (für das Verfahren in Abschnitt 6.3.1) und tiefes_lernskript_transformation.py (für das Verfahren in Abschnitt 6.3.3) wird die Trainingsstichprobe durch wiederholten Zugriff eines Samplers „ohne Zurücklegen“ auf den ReplayBuffer gebildet. In tiefes_lernskript_mehrfach.py gibt es neben dem Zwischenspeicher einen zweiten ReplayBuffer für die Stichprobe, der durch einmaligen Zugriff auf den Zwischenspeicher gefüllt und dann mehrfach durchlaufen wird.

Wie in den Skripten für das Netztraining in Abschnitt 5.2 werden in den tiefen Lernskripten eine Funktion `training_loop` und eine Funktion `test_loop` definiert. Die Funktion `training_loop` hat dabei den gleichen Ablauf wie ihr Analogon in den Netztrainingsskripten. Die Funktion `test_loop` führt in den tiefen Lernskripten hingegen anders als in den Netztrainingsskripten keine Anpassungstests durch, sondern prüft mithilfe von Partieumgebungs-Objekten die erreichte Spielstärke. Für eine festgelegte Anzahl an Zyklen werden jeweils zuerst eine festgelegte Anzahl von Partien ausgeführt (= Erfahrungssammelphase), dann die Funktion `training_loop` aufgerufen (= Trainingsphase) und dann (in jedem n-ten Zyklus) die Funktion `test_loop` aufgerufen (= Testphase) und zusätzlich die aktuellen Netzgewichte gespeichert.

Während in tiefes_lernskript.py, tiefes_lernskript_transformation.py und tiefes_lernskript_mehrfach.py die lernenden und optimierenden Spielerobjekte und die Partieumgebungen mit dem selben Bewertungsnetzobjekt verbunden sind, werden in tiefes_hybrides_lernskript.py (für das Verfahren in Abschnitt 6.3.4) die Spielerobjekte mit einem (= jeweils dem selben) Bewertungsnetzobjekt und die in der Erfahrungssammelphase genutzte Partieumgebung mit einem separaten (tabularen) Bewertungstabelleobjekt verbunden. Die jeweiligen Klassendefinitionen erlauben dies problemlos. Durch Aufruf der Methode `bewertungen_aktualisieren` auf ihrem Bewertungstabelleobjekt gibt die Partieumgebung nach jeder Partie in der Erfahrungssammelphase die neuen Beobachtungswerte in die Tabelle ein. Nach der

Erfahrungssammelphase und vor Aufruf von `training_loop` wird dann in tiefes_hybrides_lernskript.py eine Stichprobe von bewerteten Stellungen aus der Tabelle gezogen und als BewertungsDaten-Objekt in einem ReplayBuffer, der genau die Größe der Stichprobe hat, abgelegt. Dieser ReplayBuffer wird dann an `training_loop` übergeben, um das Netzgewichttraining mit den darin gespeicherten Daten durchzuführen.

6.3 Experimentelle Ergebnisse des tiefen RL-Reversi-Spielers

Wie erwähnt, gibt es für tiefe Lernalgorithmen eine große Anzahl an Architektur- und Trainigsparametern. Zudem wurden in Abschnitt 4.4 drei Lernversionen und in Abschnitt 5.1 mehrere Netzarchitekturen vorgestellt. Die Vielzahl möglicher Kombinationen ist insbesondere im Rahmen einer Masterarbeit aufgrund des hohen Zeitaufwandes für jeden einzelnen Trainingsprozess nicht handhabbar. Da in Abschnitt 5.3.2 mit einem CNN keine eindeutig besseren Ergebnisse als mit einem MLP erzielt werden konnten, werde ich für den tiefen RL-Spieler nur die einfacherer MLP-Variante untersuchen. Ferner werde ich mich auf die Lernversion 2 – also σ-greedy Lernender Spieler gegen sich selbst – beschränken, da die Version 1 mit dem ϵ-greedy Lernenden Spieler in Abschnitt 5.3.2 schlechtere Ergebnisse geliefert hat und Version 3 nicht eindeutig besser war, aber bei jedem Trainingsdurchlauf nur Gewichte für eine Farbe liefert.

In der experimentellen Überprüfung ist zu untersuchen, ob mit dem tiefen RL-Algorithmus eine bessere oder zumindest eine ähnliche Spielstärke wie mit dem Verfahren aus Abschnitt 5 erreicht werden kann. Da sich die Experimente auf Lernversion 2 mit MLP beschränken, ist der relevante Zielwert für die Erfolgsquoten gegen einen Minimax-Spieler mit Tiefe 6 der mit dieser Kombination erreichte Bestwert aus Abschnitt 5.3.2.6, also 86,80% mit Schwarz und 79,40% mit Weiß.

6.3.1 Grundvariante mit einfacher Trainingsphase

In der einfachsten Version des tiefen RL-Algorithmus habe ich Erfahrungssammelphasen mit 1.000, 2.500 und 5.000 Partien ausprobiert. Der Zwischenspeicher hat eine Größe von einer Millionen Stellungen, die Trainingsstichprobe umfasst 320.000 Stellungen (also knapp ein Drittel der zwischengespeicherten Stellungen) und das Training besteht aus einem Durchgang durch die Stichprobe. Das Training geht in der Regel über 500.000 Partien zuzüglich zu den Partien der anfänglichen Zwischenspeicherauffüllung. (Einige Trainingsprozesse sind mit einer Million Partien durchgeführt worden, haben jedoch keine besseren Ergebnisse gebracht.)

Als Optimierungsalgorithmus wird aufgrund der Erfahrungen aus Abschnitt 5.3.1 Stochastic Gradient Descent verwendet mit einer anfänglichen Lernrate von 0,005 (bei Training über 100 Zyklen) oder 0,01 (bei Training über 200 oder 250 Zyklen), die nach jedem Zyklus um 5% abgesenkt wird.

Ein Test mit jeweils 1.000 Partien pro Farbe erfolgt abhängig von der Partienanzahl in der Erfahrungssammelphase alle 5, 10 oder 20 Zyklen und somit alle 20.000

oder 25.000 Partien. Als Vergleichsspieler im Test während des Trainingsprozesses dient der Stochastische Spieler, da Test mit einem Minimax-Spieler zu zeitaufwendig sind. Tests gegen den Minimax-Spieler mit Tiefe 6 werden im Nachgang mit den gespeicherten Gewichten durchgeführt.

Die Daten aus dem erfolgreichsten Trainingsdurchlauf sind in Tabelle 21 und Abb. 15 dargestellt. Bei diesem Durchlauf betrug die Partienanzahl in der Erfahrungssammelphase 5.000. Dargestellt werden die während des Trainingsprozesses alle fünf Zyklen erhobenen Erfolgsquoten gegen den Stochastischen Spieler sowie die mit den ab Zyklus 60 ebenfalls alle fünf Zyklen gespeicherten Netzgewichten nachträglich ermittelten Erfolgsquoten gegen den Minimax-Spieler mit Tiefe 6, jeweils getrennt für Schwarz und Weiß.

Zyklen	Erfolg schwarz stochastisch	Erfolg weiß stochastisch	Erfolg schwarz Minimax	Erfolg weiß Minimax
0	48,50	50,40		
5	95,95	95,30		
10	97,20	95,60		
15	97,70	98,05		
20	98,20	98,05		
25	98,35	98,00		
30	98,60	98,55		
35	98,75	97,90		
40	98,65	98,50		
45	99,55	99,00		
50	99,00	98,45		
55	99,15	99,20		
60	99,15	99,25	67,00	64,85
65	99,15	99,25	62,05	58,55
70	99,30	99,15	78,35	72,45
75	99,40	99,15	69,25	58,25
80	98,90	99,15	62,00	68,30
85	99,25	99,05	73,15	65,90
90	99,10	98,80	56,30	65,15
95	99,15	98,55	69,45	58,90
100	99,05	99,25	75,65	70,90

Tabelle 21: Erfolgsquoten im tiefen Lernprozess

Die Erfolgsquoten gegen den Stochastischen Spieler steigen ganz am Anfang des Trainingsprozesses stark an und verbessern sich danach nur noch wenig. Die höchste Erfolgsquote mit Weiß wird gegen den Stochastischen Spieler nach 45 Zyklen mit 99,55% erreicht, mit Schwarz nach 60 Zyklen mit 99,25%. Dabei ist anzumerken,

dass sich bei Werten von über 99% eine weitere Verbesserung der Spielstärke, selbst wenn sie zustande käme, kaum noch nachweisbar wäre, sodass sich wie schon in Abschnitt 5.3.2.6 erneut zeigt, dass der Stochastische Spieler als aussagekräftiger Vergleichsmaßstab wenig geeignet ist.

Abbildung 15: Testdaten Grundvariante; orange: stochastischer Test mit Schwarz; blau: stochastischer Test mit Weiß; grün: Minimax-Test mit Schwarz; rot: Minimax-Test mit Weiß

Die Erfolgsquoten gegen den Minimax-Spieler schwanken im überprüften Bereich vom 60. Zyklus bis zum 100. Zyklus stark. Diese Schwankungen deuten auf unzureichende Konvergenz des zugrunde liegenden Lernprozesses hin. Der beste Wert mit Schwarz wird nach 70 Zyklen mit 78,35% erreicht, mit Weiß ebenfalls nach 70 Zyklen mit 72,45%. Die Korrelation zwischen den Erfolgsquoten im stochastischen und im Minimax-Test liegt mit Schwarz (nur) bei 44,58%, mit Weiß (sogar nur) bei 32,38% (berechnet als Pearson's Korrelationskoeffizient, siehe [9], S. 285f). (Ein ähnliches Auseinanderfallen der Testwerte gegen den Stochastischen Spieler

und gegen den Minimax-Spieler wurde auch schon in Abschnitt 4.4.4 beobachtet.) Dies impliziert insbesondere, dass es nicht zielführend wäre, bei den nachträglichen Minimax-Tests nur die Gewichte zu verwenden, mit denen die besten Werte im stochastischen Test erreicht wurden. Stattdessen erscheint es unumgänglich, trotz des erheblichen Zeitaufwandes für Minimax-Tests der Tiefe 6 eine größere Anzahl verschiedener Gewichte durchzuprobieren. Dies verringert wiederum die Möglichkeit, Trainingsprozesse mit vielen verschiedenen Parameterkombinationen auszuprobieren.

Die Korrelation zwischen den Erfolgsquoten im Minimax-Test für Schwarz und für Weiß beträgt (nur) 41,91%. Auch wenn im betrachteten Trainingsdurchgang die besten Werte für beide Farben mit den gleichen Gewichten erreicht werden, kann man daher nicht davon ausgehen, dass Gewichte, die ein besseres Ergebnis für Schwarz liefern, auch automatisch ein besseres Ergebnis für Weiß liefern. Für die in Abschnitt 5.3.2.3 aufgestellte Vermutung, dass ein trainiertes Netzwerk von der Farbe unabhängige Stellungsmerkmale identifizieren kann, findet sich damit vorerst keine Bestätigung.

Insgesamt ist somit festzustellen, dass die Grundvariante zwar eine Spielstärke liefert, die erheblich über der des Minimax-Spielers mit Tiefe 6 liegt, es aber nicht gelungen ist, an die in Abschnitt 5 erzielten Werte heranzureichen.

6.3.2 Grundvariante mit intensivierter Trainingsphase

In Abweichung von der Grundvariante aus dem vorhergehenden Abschnitt werden in diesem Abschnitt Trainingsprozesse betrachtet, bei denen in der Trainingsphase jeder Datenpunkt aus der Stichprobe nicht nur einmal zur Aktualisierung der Netzgewichte genutzt wird, sondern mehrere – und zwar drei – Durchgänge durch die Stichprobe erfolgen. Es werden Erfahrungssammelphasen mit 2.500 und 5.000 Partien ausprobiert, die Trainingsstichprobe umfasst 160.256 Stellungen, die übrigen Trainingsparameter und das Testregime entsprechen denen aus der Grundvariante.

Die Daten aus dem erfolgreichsten Trainingsdurchlauf sind in Tabelle 22 und Abb. 16 dargestellt. Über 100 Zyklen wurden in jeder Erfahrungssammelphase 5.000 Partien gespielt. Erfolgsquoten gegen den Stochastischen Spieler wurden alle 5 Zyklen ermittelt. Ab Zyklus 60 wurden alle fünf Zyklen die jeweils aktuellen Netzgewichte gespeichert, um mit ihnen nachträglich Erfolgsquoten gegen den Minimax-Spieler mit Tiefe 6 zu bestimmen.

Zyklen	Erfolg schwarz stochastisch	Erfolg weiß stochastisch	Erfolg schwarz Minimax	Erfolg weiß Minimax
0	39,60	43,60		
5	98,10	96,80		
10	96,75	97,40		
15	98,10	98,05		
20	98,15	97,55		
25	98,95	99,05		
30	98,20	98,75		
35	98,80	98,50		
40	98,85	98,70		
45	99,05	99,20		
50	98,90	99,45		
55	98,35	98,75		
60	99,05	99,15	69,15	64,55
65	98,90	98,40	72,10	70,80
70	99,70	98,75	77,25	71,60
75	99,00	98,95	68,70	68,05
80	99,25	99,40	77,40	73,70
85	99,80	99,50	74,50	66,30
90	99,15	99,00	65,75	55,20
95	99,25	98,95	66,75	70,25
100	99,30	98,95	73,15	57,10

Tabelle 22: Erfolgsquoten im tiefen Lernprozess mit intensivierter Trainingsphase

Wie im vorhergehenden Abschnitt steigen die Erfolgsquoten gegen den Stochastischen Spieler ganz am Anfang des Trainingsprozesses stark an. Der anschließende schwache Anstieg zieht sich hier etwas länger hin als in der Grundvariante. Die höchste Erfolgsquote gegen den Stochastischen Spieler wird sowohl mit Schwarz als auch mit Weiß nach 85 Zyklen erreicht und beträgt 99,80% beziehungsweise 99,50%.

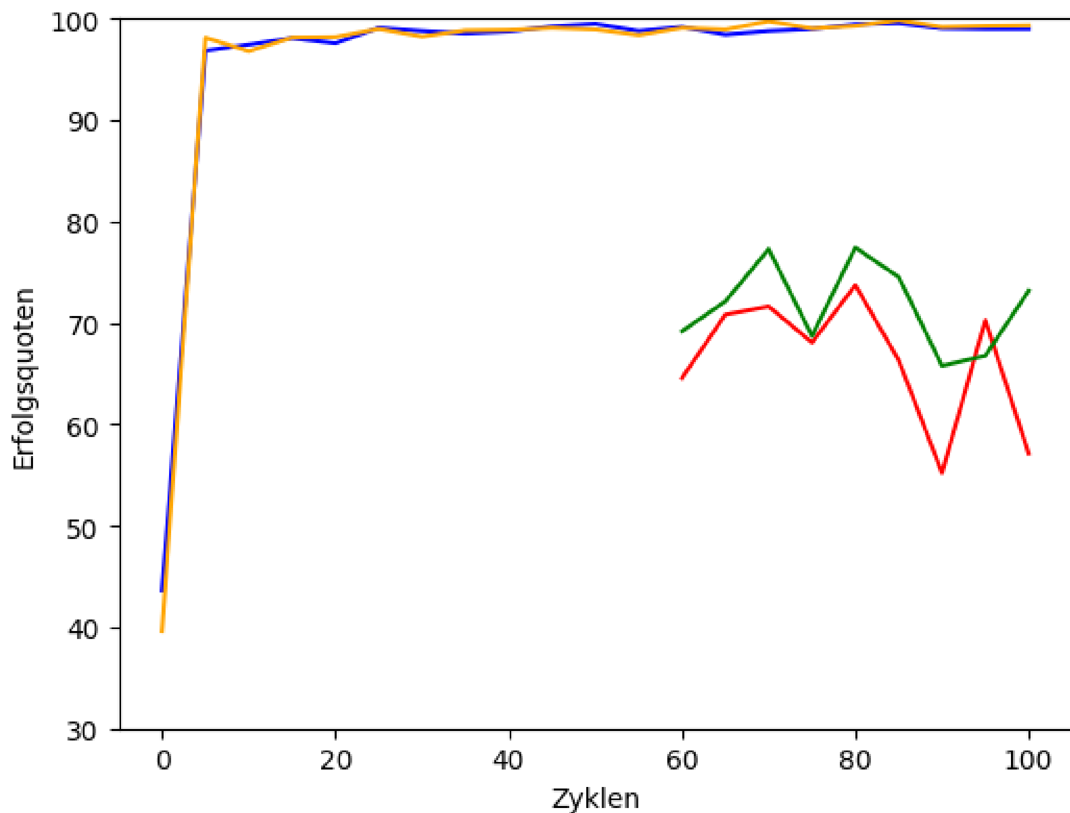

Abbildung 16: Testdaten mit intensivierter Trainingsphase; orange: stochastischer Test mit Schwarz; blau: stochastischer Test mit Weiß; grün: Minimax-Test mit Schwarz; rot: Minimax-Test mit Weiß

Die Erfolgsquoten gegen den Minimax-Spieler schwanken vom 60. bis zum 100. Zyklus etwas weniger zickzackförmig als in Abb. 15, entwickeln sich aber auch hier in keine eindeutige Richtung. Der beste Wert wird jeweils nach 80 Zyklen mit 77,40% für Schwarz und 73,70% für Weiß erreicht. Es ergibt sich somit kein signifikanter Unterschied in der Spielstärke im Vergleich zur einfachen Grundvariante.

6.3.3 Training mit angereicherten Daten

Genau wie in der Tabelle in Abschnitt 4 werden im Zwischenspeicher alle Stellungen in kanonischer Form gespeichert. Am Ende von Abschnitt 5.3.2.4 wurde bereits kurz angemerkt, dass es für ein Netzwerktraining möglicherweise nicht ideal ist, wenn es nur mit kanonischen Stellungen durchgeführt wird. Es bietet sich daher an, die im Zwischenspeicher gesammelten Stellungsdaten anzureichern, indem

man sie jeweils zufällig durch Spiegelung oder Drehung in strategisch äquivalente nicht-kanonische Stellungen umwandelt (und die jeweilige Bewertung nicht verändert). Dies entspricht einem typischen Vorgehen für die Anreicherung (*augmentation*) von Datenmengen für das Netzwerktraining mit zweidimensionalen Eingaben wie Bildern (siehe [6], S. 233; vergleiche auch Abschnitt 5.1 in [4]). Dafür wird in diesem Abschnitt ein Verfahren angewendet, dass die einzelnen Stellungen vor der Zwischenspeicherung statt sie zu kanonisieren jeweils mit einer Wahrscheinlichkeit von 25% an der Hauptdiagonalen des Spielbretts spiegelt, an der Nebendiagonalen spiegelt, um 180 Grad dreht oder unverändert lässt.

Mit diesem Verfahren wurden Trainingsdurchläufe mit Erfahrungssammelphasen mit 2.000, 2.500 und 5.000 Partien ausprobiert, die Trainingsstichprobe umfasst 160.000 Stellungen, die übrigen Trainingsparameter und das Testregime entsprechen denen aus der Grundvariante. Die Daten aus dem erfolgreichsten Trainingsdurchlauf sind in Tabelle 23 und Abb. 17 dargestellt. Bei diesem Durchlauf wurden über 200 Zyklen in jeder Erfahrungssammelphase 2.500 Partien gespielt. Erfolgsquoten gegen den Stochastischen Spieler wurden alle 10 Zyklen ermittelt. Ab Zyklus 100 wurden alle 10 Zyklen die jeweils aktuellen Netzgewichte gespeichert, um mit ihnen nachträglich Erfolgsquoten gegen den Minimax-Spieler mit Tiefe 6 zu bestimmen.

Zyklen	Erfolg schwarz stochastisch	Erfolg weiß stochastisch	Erfolg schwarz Minimax	Erfolg weiß Minimax
0	44,05	48,20		
10	95,20	97,20		
20	98,25	97,35		
30	97,70	97,55		
40	98,80	98,95		
50	97,70	98,00		
60	98,80	98,40		
70	98,25	98,95		
80	99,10	99,35		
90	99,10	99,05		
100	98,75	98,15	80,45	72,25
110	99,15	98,75	77,55	69,35
120	98,85	98,85	78,60	69,70
130	98,65	98,70	78,75	63,05
140	98,25	98,90	78,85	61,15
150	99,15	98,70	77,95	59,00
160	98,40	98,60	78,65	64,00
170	99,10	98,20	79,30	61,30
180	98,90	99,15	80,85	61,25
190	98,80	98,25	80,10	57,15
200	99,25	99,05	78,70	60,30

Tabelle 23: Erfolgsquoten im tiefen Lernprozess mit angereicherten Daten

Wie in den vorhergehenden Abschnitten steigen die Erfolgsquoten gegen den Stochastischen Spieler ganz am Anfang des Trainingsprozesses stark an. Danach zeigt sich für Schwarz ein kontinuierlicher schwacher weiterer Anstieg, für Weiß ist dies weniger eindeutig. Die höchste Erfolgsquote gegen den Stochastischen Spieler wird für Schwarz nach 200 Zyklen mit 99,25% erreicht, für Weiß schon nach 80 Zyklen mit 99,35%.

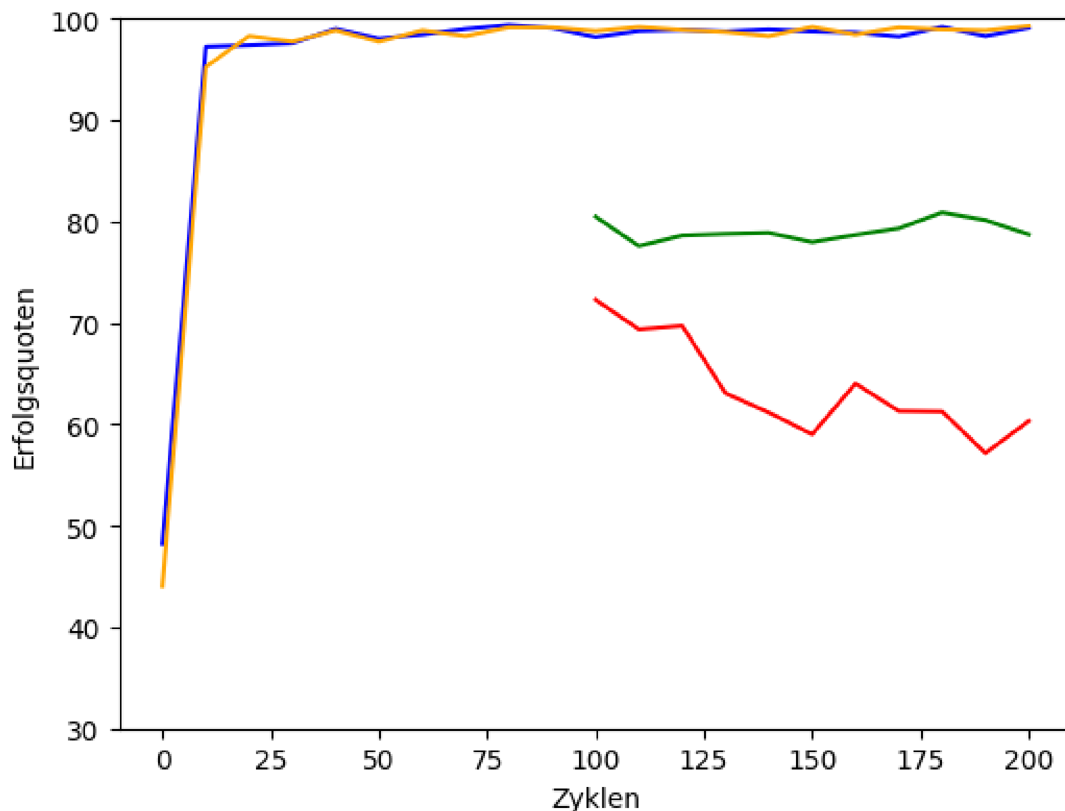

Abbildung 17: Testdaten mit intensivierter Trainingsphase; orange: stochastischer Test mit Schwarz; blau: stochastischer Test mit Weiß; grün: Minimax-Test mit Schwarz; rot: Minimax-Test mit Weiß

Die Erfolgsquoten gegen den Minimax-Spieler weisen erheblich geringere Schwankungen auf als in der Grundvariante. Allerdings entwickeln sie sich für Weiß im überprüften Bereich ab dem 100. Zyklus tendenziell in die falsche Richtung, also nach unten. Die besten Werte werden für Schwarz nach 180 Zyklen mit 80,85% erreicht und für Weiß bereits nach 100 Zyklen mit 72,25%. Es ergibt sich somit keine eindeutig höhere Spielstärke im Vergleich zur Grundvariante, auch wenn der flache Verlauf der Minimax-Erfolgsquotenkurve auf ein besseres Konvergenzverhalten schließen lässt.

Beim Vergleich mit den Erfolgsquoten aus Abschnitt 5.3.2 ist zudem zu berücksichtigen, dass die dort verwendeten Netzwerke nicht mit angereicherten Daten, sondern nur mit kanonischen Stellungen trainiert wurden. Für einen methodisch korrekten Vergleich der jeweils erreichten Spielstärke sollte das Training dieser Netzwerke ebenfalls mit angereicherten Daten durchgeführt werden.

6.3.4 Hybrides Training mit Netz und Tabelle

Wie in Abschnitt 6.1 dargestellt, unterscheiden sich die tiefen Verfahren aus den drei vorhergehenden Abschnitten insbesondere dadurch vom Vorgehen aus den Abschnitten 4 und 5, dass im Zwischenspeicher die gleiche Stellung mehrfach mit unterschiedlichen Bewertungen vorkommen kann, während in der Tabelle jede Stellung nur einmal und mit einer durchschnittlichen Bewertung verzeichnet ist. Bei den tiefen Verfahren ergibt sich die Durchschnittsbildung gewissermaßen stochastisch im Erwartungswert durch die Zufallsauswahl der Stellungen beim Netztraining. Zusätzlich werden die Bewertungen dadurch aktualisiert, dass ältere Bewertungen aus dem Zwischenspeicher gelöscht werden, wenn dieser voll ist.

Das tiefen Verfahren mag rechentechnisch effizienter erscheinen, aber es bleibt die Frage, ob durch die explizite Durchschnittsbildung in der Tabelle nicht doch „genauere" Bewertungen zur Verfügung gestellt werden, die den Lernprozess erleichtern. Um dies zu untersuchen, wird in diesem Abschnitt ein tiefes Verfahren mit expliziter Durchschnittsbildung entwickelt. Dabei wird für die Speicherung der Beobachtungsdaten eine Tabelle genutzt, für die Zugauswahl ein Bewertungsnetz. Ich bezeichne dieses Verfahren daher als „hybrid". Für das Training des Bewertungsnetzes wird zu Beginn jeder Trainingsphase eine Stichprobe bewerteter Stellungen aus der Tabelle gezogen. Es wird weiterhin davon ausgegangen, dass die Tabelle beliebig erweitert werden kann (was bei Trainingsdurchläufen mit bis zu einer Millionen Partien tatsächlich problemlos möglich ist).

Mit diesem Verfahren wurden Trainingsdurchläufe mit Erfahrungssammelphasen mit 2.000, 2.500 und 5.000 Partien ausprobiert, die Trainingsstichprobe umfasst 160.000 Stellungen, die übrigen Trainingsparameter und das Testregime entsprechen denen aus der Grundvariante. Die Daten aus dem erfolgreichsten Trainingsdurchlauf sind in Tabelle 24 und Abb. 18 dargestellt. Bei diesem Durchlauf wurden über 250 Zyklen in jeder Erfahrungssammelphase 2.000 Partien gespielt. Erfolgsquoten gegen den Stochastischen Spieler wurden alle 10 Zyklen ermittelt. Ab Zyklus 150 wurden alle 10 Zyklen die jeweils aktuellen Netzgewichte gespeichert, um mit ihnen nachträglich Erfolgsquoten gegen den Minimax-Spieler mit Tiefe 6 zu bestimmen.

Zyklen	Erfolg schwarz stochastisch	Erfolg weiß stochastisch	Erfolg schwarz Minimax	Erfolg weiß Minimax
0	38,25	42,25		
10	95,45	94,55		
20	97,90	98,30		
30	97,10	98,35		
40	98,20	98,35		
50	98,60	98,05		
60	98,70	97,90		
70	98,35	98,50		
80	98,50	99,30		
90	98,80	98,65		
100	98,10	98,65		
110	99,15	98,90		
120	99,15	98,25		
130	99,20	98,70		
140	99,30	99,10		
150	99,00	98,70	71,85	66,55
160	98,70	99,10	70,75	68,50
170	98,75	98,95	71,10	67,20
180	98,75	99,00	72,25	65,35
190	98,50	99,35	71,25	67,40
200	98,80	98,20	74,35	64,90
210	98,20	99,25	70,25	72,15
220	98,75	99,15	69,95	65,05
230	98,70	98,55	70,20	66,85
240	99,05	98,45	69,90	63,55
250	98,90	98,75	69,05	65,00

Tabelle 24: Erfolgsquoten im tiefen hybriden Lernprozess

Wie in den vorhergehenden Abschnitten steigen die Erfolgsquoten gegen den Stochastischen Spieler ganz am Anfang des Trainingsprozesses stark an. Der folgende schwache weitere Anstieg scheint für Schwarz etwas gleichmäßiger zu verlaufen als für Weiß. Die höchste Erfolgsquote gegen den Stochastischen Spieler wird für Schwarz nach 140 Zyklen mit 99,30% erreicht, für Weiß nach 190 Zyklen mit 99,35%.

Die Erfolgsquoten gegen den Minimax-Spieler verlaufen für Schwarz ähnlich schwankungsarm wie beim Verfahren mit angereicherten Daten, für Weiß ein wenig zickzackförmiger. Dabei weisen beide Erfolgsquotenkurven ein deutliches Maximum auf. Die besten Werte werden für Schwarz nach 200 Zyklen mit 74,35% und für Weiß nach 210 Zyklen mit 72,15% erreicht. Die Spielstärke liegt damit am unteren Ende der mit tiefen Algorithmen erreichten Werte. Die Vermutung, dass explizite Durch-

schnittsberechnung einen entscheidenden Vorteil bietet, kann damit hinsichtlich der Spielstärke nicht bestätigt werden. Sie scheint aber ausweislich des vergleichsweise flachen Erfolgsquotenverlaufs einen positiven Einfluss auf das Konvergenzverhalten zu haben.

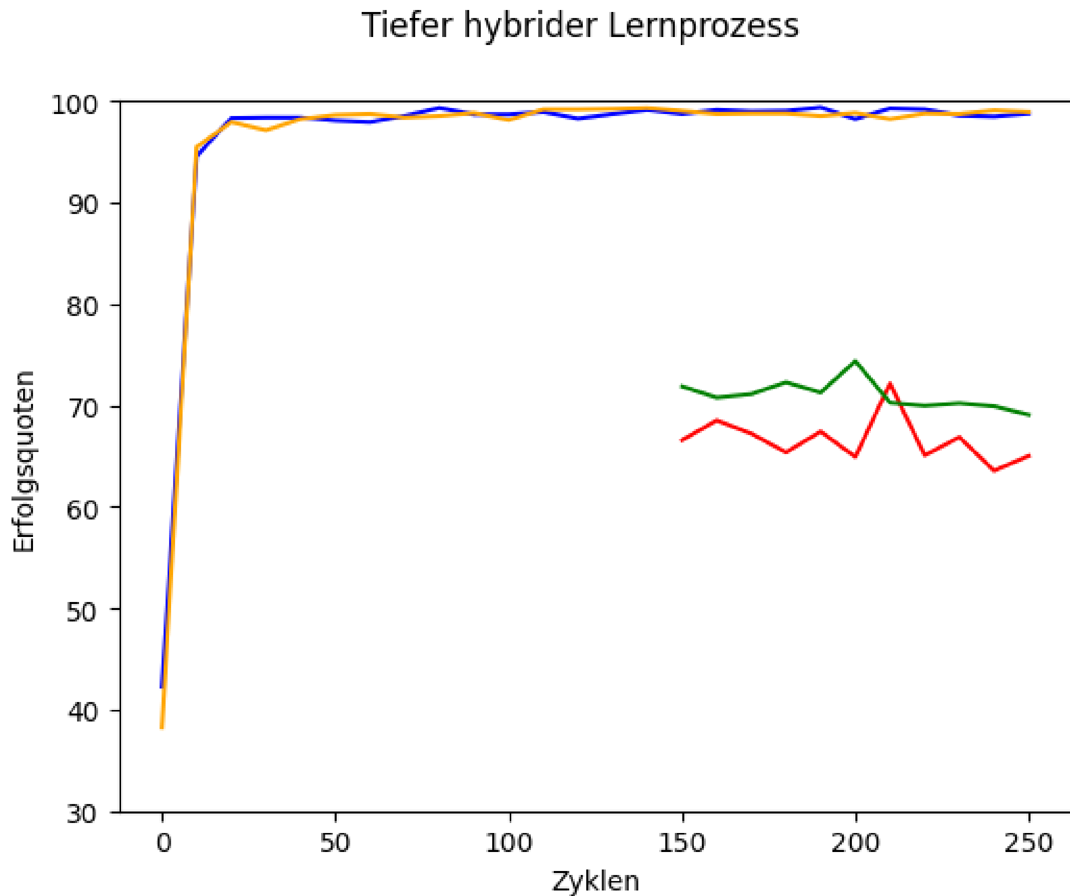

Abbildung 18: Hybride Testdaten; orange: stochastischer Test mit Schwarz; blau: stochastischer Test mit Weiß; grün: Minimax-Test mit Schwarz; rot: Minimax-Test mit Weiß

6.4 Zwischenfazit

Mit dem hier konstruierten tiefen RL-Algorithmus wurde eine Spielstärke erreicht, die einerseits deutlich über der des Minimax-Spielers mit Tiefe 6 liegt, andererseits aber nicht ganz an diejenige des in Abschnitt 5 entwickelten Algorithmus heranreicht. Verfeinerungen wie intensivierte Trainingsphasen, das Training mit angereicherten Daten oder die hybride Nutzung von Tabelle und Bewertungsnetz verbessern die Spielstärke nicht eindeutig. Sie führen allerdings zu einem weniger zick-

zackförmigen Verlauf der Erfolgsquotenkurven, was auf ein verbessertes Konvergenzverhalten hindeutet.

Die erreichte Spielstärke ergibt sich hier bereits nach deutlich weniger Partien, als bei dem Netztraining mit Tabellendaten ausgewertet wurden. Insofern mag der tiefe Algorithmus „schneller“ erscheinen. Da allerdings nicht geprüft wurde, welche Spielstärke der Algorithmus aus Abschnitt 5 mit einer entsprechend kleineren Tabelle erreichen würde, lassen sich die Geschwindigkeiten der beiden Vorgehensweisen auf Basis der vorliegenden Daten nicht wirklich vergleichen.

Erneut ist nicht auszuschließen, dass durch das Ausprobieren weiterer Werte für die Trainings- und Architekturparameter bessere Ergebnisse erreicht werden könnten. Dies gilt umso mehr, als dass wegen des hohen Zeitaufwands für Trainingsdurchläufe nur eine Lernversion und eine Netzwerkarchitektur verwendet wurden. Naheliegende wäre zudem, die konvergenzverbessernden Methoden des Trainings mit angereicherte Daten und der hybriden Nutzung von Tabelle und Bewertungsnetz zu kombinieren. Festgehalten werden kann aber zumindest, dass es nicht einfach ist, mit einem tiefen Verfahren die Ergebnisse aus Abschnitt 5 zu überbieten.

7 Zusammenfassung der Ergebnisse

Die Grundannahme des Reinforcement Learning besagt, dass Auswahlentscheidungen, die auf Grundlage der bisher mit den zur Verfügung stehenden Handlungsoptionen gemachten Erfahrungen getroffen werden, nach einer hinreichend langen Lernzeit und wenn hinreichend oft mit bislang nicht erprobten Optionen experimentiert wird, zu optimalen oder zumindest sehr guten Ergebnissen führen werden. Für den Kontext von Brettspielen lässt sich ein solches Vorgehen am einfachsten formalisieren, indem in einer Tabelle für jeden erprobten Zug festgehalten wird, welches Partieergebnis sich durchschnittlich in den Partien ergeben hat, in denen er gespielt wurde. Bei der Auswahl zwischen mehreren möglichen Zügen wird dann derjenige gewählt, für den der höchste Wert in der Tabelle steht. Dabei können Züge mit der nach ihrer Ausführung auf dem Spielbrett entstehenden Stellung identifiziert werden.

Ein derartiger Algorithmus lässt sich recht einfach implementieren. In der experimentellen Überprüfung zeigt sich jedoch, dass er auch nach einer Millionen Reversi-Lernpartien nur einen minimalen Lernfortschritt vorweisen kann. Die erreichte Spielstärke liegt nur wenig über der eines Spielers, der alle Züge rein zufällig auswählt. Dies gilt unabhängig davon, welche Strategie zur Erprobung bislang nicht gespielter Züge verwendet wird (ϵ-*greedy* oder σ-*greedy*) und ob der Spieler „gegen sich selbst“ oder gegen einen Zufallsspieler trainiert wird. Es zeigt sich, dass auch eine Tabelle mit 52 Millionen Stellungsbewertungen angesichts der sehr großen Zahl möglicher Reversi-Stellungen viel zu klein ist, um zu vermeiden, dass der Algorithmus regelmäßig in Situationen gerät, in denen die Tabelle für keinen der zur Auswahl stehenden Züge eine Bewertung enthält.

Dieses Problem kann gelöst werden, wenn die vorliegenden Stellungsbewertun-

gen dafür genutzt werden, Bewertungen für noch nicht erprobte Stellungen abzuschätzen. Hierfür wird ein neuronales Netzwerk mit Methoden des überwachten Lernens mit den Tabellendaten trainiert und dann statt der Tabelle für die Bewertung alternativer Züge genutzt. Die experimentelle Überprüfung zeigt, dass dies schon bei Nutzung eines sehr einfachen vollvernetzten vorwärtsgerichteten Netzes zu einer sehr deutlichen Steigerung der Spielstärke führt. Mit einem etwas komplexeren faltenden Netzwerk werden keine eindeutig besseren Ergebnisse erzielt. Die erreichte Spielstärke liegt deutlich über der eines Minimax-Spielers mit Tiefe 8. Bemerkenswert ist auch, dass Daten aus Tabellen, die nur für das Spiel mit den schwarzen oder weißen Steinen aufgestellt wurden, bei der Umwandlung in ein trainiertes Netzwerk genauso gut für die jeweils andere Farbe genutzt werden können.

Eine naheliegende Weiterentwicklung der betrachteten Algorithmen besteht darin, Erfahrungssammlung und Netztraining ohne Tabelle bereits ab der ersten Lernpartie zu verschränken. Hierdurch ergibt sich ein sogenannter tiefer RL-Algorithmus. Dabei wechseln sich über eine vorgegebene Anzahl Zyklen hinweg Erfahrungssammelphasen, Trainingsphasen und Testphasen ab. Schon nach vergleichsweise wenigen Zyklen ergibt sich bei diesem Vorgehen eine Spielstärke deutlich über der eines Minimax-Spielers der Tiefe 6. Es ist in der experimentellen Überprüfung jedoch nicht gelungen, mit diesen vergleichsweise komplexen Algorithmen an die Spielstärke des Verfahrens mit einem einmal mit (vielen) Tabellendaten trainierten Netzwerk heranzukommen. Verschiedene erprobte Verfeinerungen des tiefen Algorithmus können zwar Anzeichen für Konvergenzprobleme reduzieren, erreichen aber keine eindeutige Steigerung der Spielstärke.

Zu erwähnen ist, dass es bei allen erprobten Verfahren und insbesondere den zuletzt entwickelten tiefen RL-Algorithmen eine große Anzahl an alternativen Umsetzungsmöglichkeiten gibt, die im Rahmen dieser Arbeit nicht erprobt werden konnten und mit denen möglicherweise bessere Ergebnisse hätten erzielt werden können. Die Auswahl der besten Variante scheint oft eher im Bereich der (Ingenieurs-) Kunst als der Wissenschaft zu liegen. Zudem ist festzuhalten, dass die für die Untersuchung zur Verfügung stehenden Rechenkapazitäten erheblichen Einfluss auf die durchgeführten Experimente hatten, da die Anzahl verschiedener Verfahrensvarianten, die ausprobiert werden konnten, durch die lange Rechendauer der einzelnen Trainingsdurchläufe begrenzt war.

Literatur

[1] Louis Victor Allis. *Searching for Solutions in Games and Artificial Intelligence*. PhD thesis, Department of Computer Science, University of Limburg, 1994.

[2] Anton Bakhtin, Noam Brown, Emily Dinan, Gabriele Farina, Colin Flaherty, Daniel Fried, Andrew Goff, Jonathan Gray, Hengyuan Hu, Athul Paul Jacob, Mo jtaba Komeili, Karthik Konath, Minae Kwon, Adam Lerer, Mike Lewis, Alexander H. Miller, Sandra Mitts, Adithya Renduchintala, Stephen Roller, Dirk Rowe, Weiyan Shi, Joe Spisak, Alexander Wei, David J. Wu, Hugh Zhang, and Markus Zijlstra. Human-level play in the game of Diplomacy by combining language models with strategic reasoning. *Science*, 378:1067–1074, 2022.

[3] Albert Bou, Matteo Bettini, Sebastian Dittert, Vikash Kumar, Shagun Sodhani, Xiaomeng Yang, Gianni De Fabritiis, and Vincent Moens. TorchRL: A data-driven decision-making library for PyTorch. *arXiv*, 2306.00577, 2023.

[4] Xinyi Chen, Yifei Yuan, Jiaang Li, Serge J. Belongie, Maarten de Rijke, and Anders Søgaard. What if Othello-Playing Language Models Could See? *ArXiv*, 2507.14520, 2025.

[5] Richard Delorme. Edax. https://github.com/abulmo/edax-reversi, 2025.

[6] Ian Godfellow, Joshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, Cambridge, MA, 2017.

[7] Google DeepMind. Gemini 2.5 Flash. https://deepmind.google/models/gemini/flash, 2025.

[8] Rida Laraki, Jérôme Renault, and Sylvain Sorin. *Mathematical Foundations of Game Theory*. Springer Nature Switzerland, Cham, 2019.

[9] Rainer Leonhardt. *Lehrbuch Statistik*. Hogrefe, Bern, 4 edition, 2017.

[10] The Linux Foundation. PyTorch. https://pytorch.org/, 2025.

[11] Rongrong Liu, Florent Nageotte, Philippe Zanne, Michel de Mathelin, and Birgitta Dresp-Langley. Deep reinforcement learning for the control of robotic manipulation: a focussed mini-review. *Robotics*, 10(1):22, 2021.

[12] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with deep reinforcement learning. *arXiv*, 1312.5602, 2013.

[13] Antonio Norelli and Alessandro Panconesi. Olivaw: Mastering othello without human knowledge, nor a fortune. *IEEE Transactions on Games*, 15:285–291, 2021.

[14] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke E. Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Francis Christiano, Jan Leike, and Ryan J. Lowe. Training language models to follow instructions with human feedback. *Advances in neural information processing systems*, 35:27730–27744, 2022.

[15] Aske Plaat. *Deep Reinforcement Learning*. Springer Nature Singapore, 2022.

[16] Martin Riedmiller. Neural fitted q iteration — first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005*, pages 317–328, 2005.

[17] Anian Ruoss, Grégoire Delétang, Sourabh Medapati, Jordi Grau-Moya, Wenliang Kevin Li, Elliot Catt, John Reid, and Tim Genewein. Grandmaster-Level Chess Without Search. *ArXiv*, 2402.04494, 2024.

[18] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Pearson Education, 4 edition, 2021.

[19] Robert Wayne Schmittberger. The Top 100 Games 1982. *Games*, 33:41–56, 1982.

[20] Julian Schrittwieser, Ioannis Antonoglou, Thomas Hubert, Karen Simonyan, L. Sifre, Simon Schmitt, Arthur Guez, Edward Lockhart, Demis Hassabis, Thore Graepel, Timothy P. Lillicrap, and David Silver. Mastering Atari, Go, chess and shogi by planning with a learned model. *Nature*, 588:604–609, 2019.

[21] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, Timothy Lillicrap, Karen Simonyan, and Demis Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[22] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(7676):354–359, 2017.

[23] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, Cambridge, MA, 2 edition, 2018.

[24] Hiroki Takizawa. Othello is solved. *arXiv*, 2310.19387, 2024.

[25] World Othello Federation. World Othello Championships Rules, 2019.

[26] World Othello Federation. Official Rules for the Game of Othello. https://www.worldothello.org/about/about-othello/othello-rules/official-rules/english, 2024.

[27] Takuto Yamana. Egaroucid. https://www.egaroucid.nyanyan.dev, 2025.