WIRTSCHAFT
HOCHSCHULE MAINZ
UNIVERSITY OF
APPLIED SCIENCES

Bachelorarbeit

Studiengang Wirtschaftsinformatik B.Sc. dual

Evaluation der Reinforcement Learning Algorithmen Sarsa und Q-Learning am Beispiel eines Strategiespiels

Hochschule Mainz

University of Applied Sciences

Fachbereich Wirtschaft

Vorgelegt von:	Jonas Bingel
	[redacted]
	[redacted] Wiesbaden
	Matrikel-Nr. [redacted]
Vorgelegt bei:	Prof. Dr. Frank Mehler
Eingereicht am:	28.02.2022

Erklärung

Hiermit erkläre ich, dass ich die vorliegende Bachelorarbeit

> **Evaluation der Reinforcement Learning Algorithmen Sarsa und Q-Learning am Beispiel eines Strategiespiels**

selbstständig und ohne fremde Hilfe angefertigt habe. Ich habe dabei nur die in der Arbeit angegebenen Quellen und Hilfsmittel benutzt.

Zudem versichere ich, dass ich weder diese noch inhaltlich verwandte Arbeiten als Prüfungsleistung in anderen Fächern eingereicht habe oder einreichen werde.

Wiesbaden, den 28.02.2022 Jonas Bingel

Management Summary

Die vorliegende Bachelorarbeit befasst sich mit Reinforcement Learning, das ein Teilgebiet des Machine Learning ist. Reinforcement Learning und die zugehörigen Methoden zur Lösung sequenzieller Entscheidungsprobleme werden erklärt und von den anderen Teilgebieten des Machine Learning abgegrenzt. Der Fokus der Arbeit liegt auf dem Teilgebiet des Temporal-Difference Learning und den Algorithmen Q-Learning und Sarsa. Die Arbeit erläutert das Konzept von Temporal-Difference Learning. Basierend darauf werden Q-Learning und Sarsa erklärt und miteinander verglichen. Zur weiteren Evaluation der Algorithmen werden diese für das Strategiespiel Tic-Tac-Toe in Java implementiert. Es wird untersucht, ob beide Algorithmen das Spiel durch Spiele gegen sich selbst erlernen können und welche Auswirkungen verschiedene Hyperparameter auf das Training haben. Die trainierten Agenten werden auf Basis der Rate optimaler Aktionen während des Trainings und der erreichten Spielstärke verglichen. Die Auswertung zeigt, dass Q-Learning im Durchschnitt schneller konvergiert und eine höhere Spielstärke erreicht als Sarsa.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

MDP Markov Decision Process (dt. Markov-Entscheidungsprozess)

ML Machine Learning (dt. Maschinelles Lernen)

RL Reinforcement Learning (dt.. bestärkendes Lernen)

SA Tupel State-Action Tupel (dt. Zustand-Aktion Tupel)

TD Temporal-Difference

TDL Temporal-Difference Learning

TTT Tic-Tac-Toe

Symbolverzeichnis

Lateinische Buchstaben

$\mathbb{E}$	Erwartungswert
$\mathbb{1}_{prädikat}$	Indikatorfunktion, 1 wenn das Prädikat wahr ist, sonst 0
s, s'	Zustände in einem MDP, s' ist der Folgezustand
$A(s)$	Menge aller möglichen Aktionen in einem Zustand s
a	eine Aktion in einem MDP
r	ein Reward in einem MDP
t	diskreter Zeitschritt
T	letzter Zeitschritt einer Episode
S	Menge aller Zustände ohne Terminalzustände
S^+	Menge aller Zustände inkl. Terminalzustände
Y	Menge aller Afterstates
y	ein Afterstate
w	W-Value eines Afterstates
S_t	Zustand der Umgebung zum Zeitschritt t
S_T	Terminalzustand
A_t	Aktion, die Agent im Zeitschritt t wählt
R_t	Reward, der im Zeitschritt t ausgegeben wird
G_t	Return, kumulierter Reward nach Zeitschritt t
v_π	State-Value Funktion zur Policy π
v_*	optimale State-Value Funktion
V	Schätzwert für die State-Value Funktion

q_π Action-Value Funktion zur Policy π

q_* optimale Action-Value Funktion

Q Schätzwert für die Action-Value Funktion

B_G Bitboard einer Spielfeldkonstellation, das Symbole nicht unterscheidet

B_X Bitboard des Symbols X

B_O Bitboard des Symbols O

B_s Bitboard des Zustands s

Griechische Buchstaben

α Lernrate

γ Diskontierungsfaktor

δ_t Temporal-Difference Error zum Zeitschritt t

ϵ Explorationswahrscheinlichkeit

π eine Policy

π_* optimale Policy

Formelverzeichnis

Listingverzeichnis

Algorithmusverzeichnis

1. Einleitung

1.1. Motivation

Ein zentrales Teilgebiet Künstlicher Intelligenz ist Machine Learning (dt. Maschinelles Lernen, ML), das sich mit Methoden befasst, um Computer aus Erfahrungen, in Form von Trainingsdaten, lernen zu lassen [1, S. 178], [2, S. 524]. Jedoch ist es bei interaktiven Anwendungen nicht praktikabel repräsentative Trainingsdaten für alle Situationen zu erhalten oder die korrekte Entscheidung ist nicht bekannt [3, S. 24]. Zudem sind Entscheidungen oft sequenziell und gegebenenfalls muss die Entscheidung getroffen werden auf eine direkte Belohnung zu verzichten, um später eine größere Belohnung zu erhalten [4, S. 1].

Reinforcement Learning (dt.. bestärkendes Lernen, RL) ist das Teilgebiet von ML das sich mit der Erstellung von Agenten beschäftigt, die sequenziell Entscheidungen treffen, um einen Gesamtgewinn zu maximieren [4, S. 1], [3, S. 1]. Dafür interagiert ein Agent mit seiner Umgebung (engl. Environment), um Erfahrungen zu sammeln und aus diesen, ohne zusätzlich eingebrachtes Wissen zu lernen. Aufgrund dieser Eigenschaft ist RL das Teilgebiet von ML, das dem menschlichen Lernen am nächsten ist. [3, S. 4]

Zu einer Anwendung und Evaluation von RL Algorithmen eigenen sich (Strategie-)spiele, insbesondere wegen des sequenziellen Aufbaus, klar definierten Umgebung und der Reproduzierbarkeit [4, S. 1]. Beispielsweise wurden in [5] und [4] für die Spiele Backgammon und Othello RL Agenten trainiert, die allein durch Self-play eine hohe Spielstärke erreichen. Beim Self-play spielt ein Agent zum Training nur gegen sich selbst und erhält kein externes Wissen. Externes Wissen könnte beispielsweise bereitgestellt werden in Form von Expert Play, d.h. Gegnern, die optimalen Spielstrategien folgen und so auf Experten-Niveau spielen [6, S. 561]. In beiden Beispielen wurden Algorithmen aus dem RL Teilbereich des Temporal-Difference Learning (TDL) verwendet. Zwei richtungsweisende und verbreitete TDL Algorithmen sind Q-Learning und Sarsa, die eng miteinander verwandt sind [3, S. 138], [4, S. 1], [7, S. 1].

In der vorliegenden Arbeit soll das simple Strategiespiel Tic-Tac-Toe (TTT) zur Evaluation dieser Algorithmen genutzt werden, das sich aufgrund verschiedener Aspekte dafür eignet. Erstens ist TTT ein simples und bekanntes Spiel und dadurch ein anschauliches Beispiel [8, S. 6]. Zweitens ist TTT ein gelöstes Spiel mit optimalen Strategien, was eine Evaluation der Agenten ermöglicht [9, S. 533ff.]. Drittens, besitzt TTT einen diskreten und kleinen Zustands- sowie Aktionsraum, sodass Agenten schnell trainiert werden können [10, S. 3]. Außerdem können dieselben Spielzustände durch mehrere Spielabfolgen erreicht werden. Dies ermöglicht eine Evaluation des Konzepts der Afterstates (dt. Folgezustände), das die gesammelte Erfahrung effektiver nutzen soll. [3, S. 136f.]

1.2. Ziele und Forschungsfragen

Die vorliegende Arbeit verfolgt zwei zentrale Ziele. Zum einen die Erklärung der RL Algorithmen Q-Learning und Sarsa sowie die Erarbeitung von deren Unterschieden. Zum anderen die Anwendung und Evaluation der beiden Algorithmen am Beispiel des Strategiespiels Tic-Tac-Toe. Die zentrale Forschungsfrage für das zweite Ziel ist, ob die beiden Algorithmen durch Self-play eine hohe Spielstärke erreichen und optimale Spielstrategien bzw. Expert Play erlernen. Unter diese Forschungsfrage gliedern sich die folgenden weiteren Forschungsfragen:

- Eine Abschätzung für die Anzahl der Trainingsepisoden, die ungefähr benötigt werden, um Expert Play in TTT durch Self-play zu erlernen?
- Welche Auswirkung hat die Verwendung des Konzepts der Afterstates auf das Konvergenzverhalten und Spielstärke der Agenten?
- Was sind die besten Hyperparameter für Q-Learning und Sarsa?
- Wie unterscheiden sich Q-Learning und Sarsa hinsichtlich ihrer Konvergenz?
- Welcher Agent erreicht eine bessere Spielstärke, wenn beide Algorithmen mit ihren optimalen Hyperparametern trainiert werden?
- Was ist aus Sicht der Agenten die optimale erste Aktion?

1.3. Aufbau der Arbeit

Die vorliegende Bachelorarbeit gliedert sich inklusive dieser Einleitung in sechs Kapitel. Kapitel 2 dient der Darstellung der Grundlagen, die notwendig sind, um die Algorithmen Q-Learning und Sarsa zu verstehen und auf TTT anwenden und evaluieren zu können. Dazu werden die Bestandteile von RL und das zu implementierende Spiel erklärt und die beiden Algorithmen verglichen. Kapitel 3 beschreibt die Methodik und Anwendung der Algorithmen auf TTT sowie die Kodierung des Spiels selbst. Anschließend wird der gewählte Trainingsaufbau der Agenten und die Evaluation mit den zugehörigen Metriken vorgestellt. In Kapitel 4 wird die gewählte Architektur sowie die Implementierung in Java vorgestellt. Kapitel 5 diskutiert die Ergebnisse der Evaluation der Algorithmen und vergleicht diese. Anschließend werden auf Basis der gesammelten Erkenntnisse die Forschungsfragen beantwortet. Zum Schluss wird in Kapitel 6 eine Zusammenfassung der Ergebnisse vorgenommen und der Inhalt der Arbeit wird kritisch betrachtet, bevor ein Ausblick für künftige Arbeiten gegeben wird.

2. Grundlagen

Das Kapitel behandelt zuerst die Grundlagen von Reinforcement Learning. Basierend darauf werden das Konzept des Temporal-Difference Learning und die Algorithmen Q-Learning und Sarsa erklärt. Anschließend wird der Minimax-Algorithmus erläutert, der für die Evaluation der Agenten in der Auswertung genutzt wird. Abschließend wird das Strategiespiel Tic-Tac-Toe erklärt.

2.1. Reinforcement Learning

Zunächst wird der Begriff Reinforcement Learning erklärt und von den anderen Teilgebieten des Machine Learning abgegrenzt. Anschließend werden die zu lösenden Probleme und dafür genutzten Methoden formalisiert.

2.1.1. Begriffserklärung und Abgrenzung

Reinforcement Learning (RL) ist das Teilgebiet des Machine Learning , das sich mit der Lösung von Entscheidungsproblemen beschäftigt [3, S. 1], [4, S. 1]. RL basiert auf der sogenannten „Reward Hypothese“, nach der jedes sequentielle Entscheidungsproblem als Optimierungsproblem beschrieben werden kann, bei dem der kumulierte Reward (dt. Belohnung) aller getroffenen Entscheidungen maximiert werden soll [3, S. 53], [11, S. 24].

Das Ziel der RL Methoden ist einen Agent zu trainieren, der eine Abbildung von Situationen auf die jeweils optimale Aktion lernt und so den Gesamtreward maximiert. Dem Agent wird nicht mitgeteilt, was die optimale Aktion in einer Situation ist. Stattdessen ist das grundlegende Paradigma von RL Methoden, dass ein Agent mit seiner Umgebung interagiert und Erfahrung sammelt. Der Agent muss durch „trial and error“ lernen, welche Aktionen in einer Situation gut sind und welche mit dem größten Reward verbunden ist. [3, S. 1ff.] Ein Beispiel der realen Welt sind Kinder, die laufen lernen. Dies erfolgt ohne Instruktionen, sondern nur durch „trial and error“. Richtige Aktionen werden belohnt durch Vorwärtskommen, während falsche Aktionen durch Fallen bestraft, d. h. negativ belohnt, werden. [1, S. 289]

RL unterscheidet sich somit von den anderen Teilgebieten des ML. Beim Supervised Learning (dt. Überwachtes Lernen) lernt ein Agent basierend auf einem gelabelten Trainingsdatensatz, die Abbildung einer Inputmenge auf einen Output. Ein Supervisor gibt dem Agenten instruktives Feedback, ob die beste Aktion gewählt wurde und was diese ist [1, S. 289f.], [3, S. 2].

Ein RL Agent muss hingegen seine Daten selbst durch Interaktion mit der Umgebung sammeln. Da der Agent seine Aktionen selbst wählt und seine Umgebung dadurch aktiv beeinflussen kann, sind die Daten stochastisch abhängig von den Aktionen, die der Agent zuvor getroffen hat. [11,

S. 16] Zudem kann der RL Agent kein instruktives Feedback erhalten, da sequenzielle Entscheidungsprobleme gelöst werden sollen und die Bewertung einzelner Aktionen gegebenenfalls nicht bekannt ist. Erst nach Erreichen bzw. Nicht-Erreichen des Ziels können die Aktionen bewertet und der Agent durch ein evaluatives Feedback belohnt werden. [3, S. 17] Dieses Problem des verzögerten Rewards ist ein weiteres Alleinstellungsmerkmal von RL und bekannt als „Credit Assignment Problem“ [3, S. 17]. Jedoch ermöglicht es die Anwendung von RL auf Probleme, bei denen die optimale Strategie nicht bekannt ist. [3, S. 2], [11, S. 16] Zudem unterscheidet sich RL vom Unsupervised Learning (dt. Unüberwachtes Lernen). Dieses verwendet zwar ebenfalls ungelabelte Daten, aber versucht darin Strukturen zu finden und somit andere Probleme zu lösen [3, S. 2].

2.1.2. Formalisierung des Reinforcement Learning Problems

Eine Formalisierung des RL Problems bedeutet eine Formalisierung der Reward Hypothese. Wie in Abschnitt 2.1.1 beschrieben, interagiert ein Agent mit seiner Umgebung. Alles außerhalb der direkten Kontrolle des Agenten wird zur Umgebung gezählt. Diese Trennung zwischen Agent und Umgebung wird als Agent-Umgebung-Schnittstelle bezeichnet. [3, S. 47] Der Agent interagiert mit seiner Umgebung in diskreten Zeitschritten t. Man unterscheidet zwischen episodischen Problemen, die zu einem Zeitschritt T enden und Problemen unendlicher Zeitdauer, die kein definiertes Ende haben [3, S. 11].[1] Die Folge der Zeitschritte $t, t+1, ...T$ wird als Episode bezeichnet [3, S. 54].

Die Interaktion kann auf drei Signale reduziert werden, wie Abb. 2.1 zeigt. In jedem diskreten Zeitschritt t erhält der Agent den aktuellen Zustand $S_t \in S$ der Umgebung. Auf Basis dieser Beobachtung wählt der Agent eine Aktion $A_t \in A$. Im nächsten Zeitschritt erfährt der Agent, wie die Umgebung auf seine Aktion reagiert durch ein Tupel $(S_{t+1}; R_{t+1})$. Dies umfasst den neuen Zustand, in dem sich der Agent befindet und den Reward. Der letzte Zustand einer Episode S_T wird als Terminalzustand bezeichnet. Der Reward ist definiert als $R_{t+1} \in \mathbb{R}$ und kann somit positiv oder negativ sein, um eine Belohnung und Bestrafung zu modellieren. [3, S. 47 ff.]

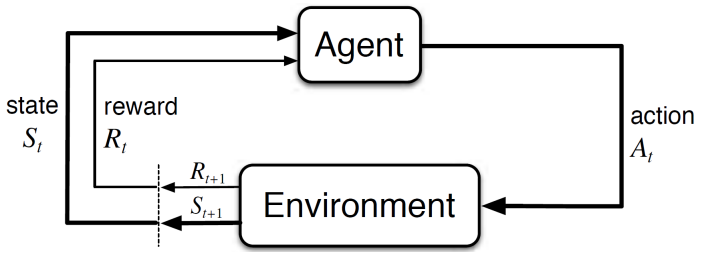


Abbildung 2.1.: Modell der Interaktion zwischen Agent und Umgebung (engl. Environment) in einem MDP [2]

Zur Modellierung der Umgebung und deren Dynamik können Markov Decision Process (dt. Markov-Entscheidungsprozess, MDP) genutzt werden. MDP basieren auf Markov Ketten und

[1] Probleme unendlicher Zeitdauer werden im Englischen als „continuous“ bezeichnet [3, S. 11]
[2] Abbildung entnommen aus [3, S. 47]

gehören zum Gebiet der mathematischen Optimierung. Ein MDP ist definiert als ein Tupel (S, A, p, r) [3, S. 47ff.], [11, S. 5]:

- S = Menge von Zuständen s
- A = Menge möglicher Aktionen a
- $p(s' \mid s, a) = Pr(S_{t+1} = s' \mid S_t = s, A_t = a)$ für das gilt: $\sum_{a \in A} p(s' \mid s, a) = 1$
  State-Transition Probability
- $r(s, a) \to \mathbb{R}$ Rewardfunktion

Die Dynamik der Zustandsübergänge wird modelliert durch die State-Transition Probability p (dt. Zustandsübergangs Wahrscheinlichkeit). Sie beschreibt die Wahrscheinlichkeit, bei Wahl von Aktion a im Zustand s in den Folgezustand s' überzugehen. Die State-Transition Probability ist als Wahrscheinlichkeitsverteilung definiert, da die Zustandsübergänge nicht deterministisch sein können.

Zudem definiert die State-Transition Probability, dass der Folgezustand s' nur vom Zustands s und der darin gewählten Aktion a abhängig ist. Zuvor besuchte Zustände oder gewählte Aktionen müssen zur Berechnung des Folgezustands nicht bekannt sein und sind implizit im Zustand s enthalten. Dies wird als Markov Eigenschaft bezeichnet und ist Voraussetzung dafür, dass ein Problem als ein MDP formuliert werden kann. [12, S. 66]

Nach jeder Aktion des Agenten verteilt die Umgebung abhängig vom aktuellen Zustand einen Reward gemäß der Rewardfunktion. Die Rewardfunktion definiert das Ziel, das der Agent erreichen soll, da dieser den kumulierten Reward maximiert. Der kumulierte Reward, den der Agent ausgehend vom Zeitschritt t erhält, wird als Return G_t bezeichnet. Für episodische Probleme ist der Return definiert als [3, S. 54]:

$$G_t = R_{t+1} + R_{t+2} + ... + R_T \tag{2.1}$$

Für Markov-Prozesse mit einer unendlichen Dauer. d. h. eine diskrete, unendliche Markov-Kette, ist G_t eine unendliche Reihe, für die zur Berechnung ein Diskontierungsfaktor $0 \leq \gamma \leq 1$ notwendig ist Der Diskontierungfaktor kann auch auf episodische Probleme angewandt werden, um das Verhalten des Agenten zu beeinflussen. Für $\gamma = 0$ beträgt $G_t = R_{t+1}$, sodass der Agent nur den direkten Reward maximiert. Hingegen beachtet der Agenten die späteren Rewards mit steigendem γ mehr. [3, S. 55f.]

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

Da ein Return G_t den darauf folgenden Return G_{t+1} enthält, kann Gleichung (2.2) umgeformt werden in Gleichung (2.3), um diese Beziehung zu verdeutlichen. Diese Beziehung aufeinander folgender Returns ist die Basis vieler RL Methoden. [3, S. 54 f.]

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) = R_{t+1} + \gamma G_{t+1} \tag{2.3}$$

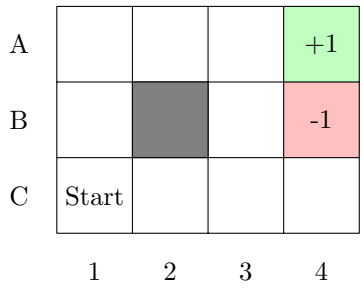


Abbildung 2.2.: Aufbau des „Gridworld“ Problems. Agent startet im Feld C1. Die Felder A4 und B4 sind Terminalzustände. B2 ist ein Hindernis.[3]

Ein Beispiel für ein RL Problem, das als MDP modelliert werden kann, ist die sogenannte „Gridworld“ in Abb. 2.2. Der Agent startet im Feld $C1$ und soll lernen das Feld A4 zu erreichen. Der Zustandsraum sind die einzelnen Felder, die der Agent betreten kann. Das Feld B2 repräsentiert ein Hindernis und zählt nicht zum Zustandsraum. Die Felder A4 und B4 sind Terminalzustände. In jedem Zustand wählt der Agent als Aktion eine Himmelsrichtung, in der er sich bewegen möchte. Somit umfasst der Aktionsraum für jeden Zustand: Norden, Westen, Süden und Osten. Bei einer Bewegung in Richtung Wand oder dem Hindernis B2 verbleibt der Agent im gleichen Feld. Die Zustandsübergänge sind nicht deterministisch. Wählt der Agent eine Richtung aus, bewegt er sich nur mit einer Wahrscheinlichkeit von 80% dorthin. Es besteht eine Wahrscheinlichkeit von je 10%, dass sich der Agent in eine der angrenzenden Richtungen bewegt. Die Rewardfunktion definiert das Ziel des Agenten. Erreicht der Agent das Feld A4 erhält er einen Reward von $+1$. Hingegen erhält er im Agent beim Erreichen des Feldes B4 eine Bestrafung von -1. Der direkt Reward für jeden anderen Zustandsübergang des Agenten beträgt $-0,1$. Da der Agent den Return maximiert, versucht er dadurch das Ziel in möglichst wenigen Zustandsübergängen zu erreichen. [11, S. 10 ff.]

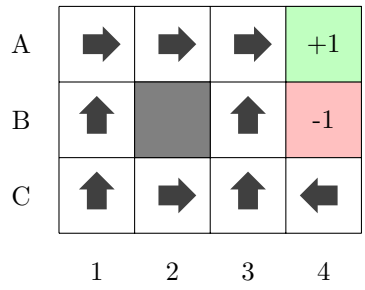
2.1.3. Policy und State-Value Funktion

Der Return G_t ist abhängig von den Aktionen, die der Agent in jedem Zustand wählt. Die Strategie, nach der Aktionen gewählt werden, wird als Policy π bezeichnet. Eine Policy π ist eine Funktion $\pi(a \mid s)$, die für alle Zustände $s \in S$ die Wahrscheinlichkeit definiert, dass der Agent eine Aktion $a \in A$ wählt. Somit beschreibt eine Policy das Verhalten eines Agenten in jedem Zustand. Eine Policy kann stochastisch $\pi(a \mid s)$ oder deterministisch $\pi(s)$ sein. [3, S. 58]

Auf der Basis einer Policy kann jedem Zustand ein Wert zugewiesen werden, der als State-Value (dt. Zustandswert) bezeichnet wird. Der State-Value ist der Erwartungswert des Return G_t, wenn der Agent im Zustand s der Policy π bis zum Ende der Episode folgt. Diese Abbildung von Zustand auf State-Value ist die State-Value Funktion:

$$v_\pi(s) = \mathbb{E}[G_t | S_t = s] \tag{2.4}$$

[3] eigene Darstellung in Anlehnung an [11, S. 10ff.]



(a) Mögliche Policy π

A	0,31	0,51	0,72	+1
B	0,15		0,36	-1
C	0,01	0,01	0,015	-0,09
	1	2	3	4

(b) State-Values zur Policy π aus (a)

Abbildung 2.3.: Policy und State-Value für das Beispiel Gridworld[4]

Die State-Value Funktion kann mittels Gleichung (2.3) als Return aufeinander folgender Zeitschritte und somit aufeinander folgender State-Value Funktionen umgeformt werden:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[R_{t+1}+\gamma G_{t+1}] = R_{t+1}+\gamma\mathbb{E}[G_{t+1} \mid S_{t+1} = s'] = R_{t+1}+v_\pi(s') \quad (2.5)$$

Abb. 2.3 zeigt für das Beispiel Gridworld eine mögliche deterministische Policy und die dazugehörige Zuweisung von State-Values. Ein Pfeil symbolisiert die Aktion, die der Agent gemäß Policy im jedem Zustand wählt. Zur Berechnung der State-Values wurde die oben beschriebene Rewardfunktion und ein Diskontierungsfaktor $\gamma = 0,9$ verwendet.

Die State-Value Funktion ermöglicht den Vergleich von Policies. Eine Policy π ist besser als eine andere Policy π' wenn ihre zugehörige State-Value Funktion für alle Zustände s einen größeren Wert hat, als die Policy π': $v_\pi(s) \geq v_{\pi'}(s) \forall s \in S$

Es kann mindestens eine Policy geben, die besser als alle anderen Policies oder gleich gut ist. Diese Policy wird als optimale Policy π_* bezeichnet und die zugehörige Value Funktion als optimale State-Value Funktion v_*. [3, S. 61 ff.] Eine Policy ist optimal, wenn der Return G_t für jeden Zustand $s \in S$ den maximal möglichen Wert annimmt. Somit ist die optimale Policy π_* die Lösung für den MDP und das darin formulierte RL Problem [3, S. 62 ff.]. Ist für einen MDP das Tupel (S, A, p, r) gegeben, kann die optimale Policy durch Dynamic Programming ermittelt werden. Beim Dynamic Programming wird ein Gleichungssystem aufgestellt und durch iteratives Verbessern der Policy und State-Value Funktion gelöst.[3, S. 73ff.]

2.2. Temporal-Difference Learning

Dieser Abschnitt behandelt das RL Teilgebiet des Temporal-Difference Learning (TDL). Zunächst wird das Konzept von TDL am Beispiel von Temporal-Difference (TD) Prediction erklärt. Anschließend werden die darauf basierenden TD Algorihtmen Q-Learning und Sarsa vorgestellt und miteinander verglichen.

[4] eigene Darstellung in Anlehnung an [11, S. 75.]

2.2.1. Temporal-Difference Prediction

Wenn ein RL Problem als ein MDP beschrieben werden kann, aber die State-Transition Probability oder die Rewardfunktion nicht bekannt sind, kann kein Modell der Umgebung erstellt werden. Folglich lassen sich diese RL Probleme nicht mittels Dynamic Programming lösen. TDL ist ein Teilgebiet von RL, das Methoden definiert, die kein Modell der Umgebung und ihrer Dynamik benötigen. Stattdessen lernen TD Methoden direkt durch Interaktion mit der Umgebung. TDL gilt daher als einer der zentralen Meilensteine von RL und TD Methoden werden als modellfreies RL bezeichnet [3, S. 119]. Die Voraussetzung zur Anwendung von TDL ist, dass der Agent wie in Abschnitt 2.1.2 beschrieben mit der Umgebung gemäß einer Policy π interagieren kann [11, S.88, S. 94].

Von der Umgebung erhält der Agent seinen aktuellen Zustand S_t und nach einer Aktion A_t im nächsten Zeitschritt den Folgezustand S_{t+1} und Reward R_{t+1}. Das Tupel $(S_t, A_t, S_{t+1}, R_{t+1})$ ist ein Sample (dt. Probe) von der Umgebung. Auf Basis der Samples schätzt TD Prediction die zur Policy π zugehörige State-Value Funktion v_π. Der Schätzwert der State-Value-Funktion V wird zu Beginn für jeden Zustand willkürlich initialisiert. [3, S. 120f.]

Das Schätzen der State-Value Funktion durch Interaktion ist möglich aufgrund der in Gleichung (2.5) definierten Beziehung zwischen State-Values aufeinander folgender Zeitschritte. Nach dieser Gleichung ist der exakte Wert der State-Value Funktion unter Policy π eines Zustands S_t der direkte Reward R_{t+1} und der State-Value des Folgezustands S_{t+1}. [3, S. 120f.]

$$v_\pi(S_t) = R_{t+1} + v_\pi(S_{t+1}) \tag{2.6}$$

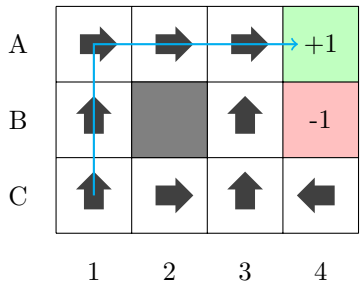
Da in einem Sample der Zustand, Reward und Folgezustand enthalten sind, kann der State-Value für den Zustand S_t auf Basis des darauffolgenden Zeitschritts geschätzt werden. Dieser Schätzwert für $V(S_t)$ wird als TD Target bezeichnet. [3, S. 120f.]

$$V(S_t) = R_{t+1} + \gamma V(S_{t+1}) \tag{2.7}$$

Da das TD Target den Reward aus der Umgebung enthält, ist es ein besserer Schätzwert für S_t als der willkürlich initialisierte Schätzwert $V(S_t)$. Die Differenz zwischen beiden Schätzwerten wird als TD Error δ_t bezeichnet. Der TD Error beschreibt den Unterschied zwischen zwischen dem alten und neuen Schätzwert und somit wie gut die alte Schätzung war. [3, S. 120f.]

$$\delta_t = R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{2.8}$$

Da das TD Target ein besserer Schätzwert für S_t ist, muss der Schätzwert $V(S_t)$ in Richtung des TD Target korrigiert werden. Diese Aktualisierung ist das TD Update und ein exponentiell geglätteter Mittelwert. [3, S. 33] Gleichung (2.9) zeigt die Formel für das TD Update. Im TD Update wird der alte Schätzwert gemäß einer Lernrate α aktualisiert, die die Gewichtung des TD Error festlegt. [3, S. 120]



$$V(A3) \leftarrow V(A3) + \alpha[R + \gamma V(A4) - V(A3)]$$
$$V(A3) = 0 + 0,1[-0,1 + 0,9 \cdot 1 - 0] = 0,1$$

Abbildung 2.4.: TD Update am Beispiel von Gridworld. Der blaue Pfad zeigt den Pfad es Agenten in der ersten Episode. Rechts wird das TD Update für den Zustand A3 berechnet. [5]

$$V(S_t) \leftarrow V(S_t) + \alpha[R_t + \gamma V(S_{t+1}) - V(S_t)] \tag{2.9}$$

Um die State-Value Funktion für alle Zustände S zu schätzen, interagiert der Agent wiederholt über mehrere Episoden mit der Umgebung. Nach jedem Zeitschritt führt der Agent für den im vorherigen Zeitschritten besuchten Zustand das TD Update durch. Dieser Ablauf bildet die Grundlage für alle TD Methoden. [3, S. 121] In [12] und [13] wurde bewiesen, dass TD Prediction zur wirklichen State-Value Funktion v_π konvergiert. Dafür müssen zwei Bedingungen erfüllt werden. Zum einen müssen alle Zustände hinreichend oft besucht werden. Zum anderen muss die im TD Update genutzte Lernrate α hinreichend klein sein oder im Laufe des Trainings abnehmen. [3, S. 33, S. 124], [13, S. 285 f.]

Am Beispiel von Gridworld aus Abschnitt 2.1.3 soll das TD Update durchgeführt werden. Die Rewardfunktion und State-Transition Probability sind unverändert, dem Agenten jedoch nicht bekannt. In Abb. 2.4 wird die zu untersuchende Policy dargestellt. Die State-Values aller Zustände werden mit 0 initialisiert. Der blaue Pfeil zeigt die Zustandsübergänge des Agenten in der ersten Episode. Rechts neben der Abbildung ist das TD Update mit Lernrate $\alpha = 0,1$ und Diskontierungsfaktor $\gamma = 0,9$ für den Zustand A3. Das TD Update wird im darauf folgenden Zeitschritt durchgeführt und somit nachdem der Agent das Ziel A4 erreicht hat.

2.2.2. Sarsa

Der TD Algorithmus Sarsa verwendet das Konzept von TD Prediction, um die optimale Policy π_* zu ermitteln. Statt der State-Value Funktion v_π approximiert Sarsa die Action-Value Funktion q_π. Der Schätzwert der Action-Value Funktion q_π ist Q. [3, S. S. 129] Die Action-Value Funktion $q_\pi(s, a)$ weist State-Action Tupel (dt. Zustand-Aktion Tupel, SA Tupel) einen Q-Value zu. Der Q-Value ist der kumulierte Reward, den der Agent erwarten kann, wenn er im Zustand s die Aktion a wählt und danach der Policy π folgt. Für die Action-Value Funktion gilt daher auch die Beziehung aufeinander folgender Returns, sodass das TD Update wie folgt formuliert werden kann [3, S. 129f.]:

[5] eigene Darstellung in Anlehnung an [11, S. 99]

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{2.10}$$

Um das Update durchführen benötigt Sarsa das Tupel $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, was der Ursprung für den Namen des Algorithmus ist. Auf Basis der Update-Regel kann Sarsa eine Schätzung Q der Action-Value Funktion zur Policy π aufstellen, nach der Sarsa mit der Umgebung interagiert. [3, S. 129f.] Der Agent verwaltet die Abbildung von SA Tupel auf ihren geschätzten Q-Value in einer Q-Tabelle. Mittels dieser kann Sarsa für jeden Zustand die Aktion ermitteln, die im höchsten Q-Value resultiert. Da Sarsa die Q-Values mit jeder neuen Episode aktualisiert, kann Sarsa zur Interaktion mit der Umgebung einer Greedy-Policy folgen. Bei einer Greedy-Policy wird immer die Aktion gewählt, die den höchsten Q-Value hat.[6] Dadurch wird iterativ die Action-Value Funktion verbessert und die Policy nähert sich optimalen Policy π_*. [3, S. 129f.]

Würde Sarsa ausschließlich gemäß einer Greedy-Policy mit der Umgebung interagieren, könnte der Agent in lokalen Maxima verweilen und würde nicht die optimale Policy lernen. Eine mögliche Lösung ist die ϵ-greedy-Policy. Die ϵ-greedy-Policy wählt mit einer Wahrscheinlichkeit von ϵ eine zufällige Aktion und $1-\epsilon$ die Aktion mit dem höchsten Q-Value. Der Hyperparameter ϵ ist die Explorationswahrscheinlichkeit. [3, S. 28f.]

Damit Sarsa zur optimalen Policy π_* konvergiert, muss neben den Konvergenzbedingungen von TD Prediction das Epsilon während dem Training gegen 0 konvergieren [3, S. 129]. Bei der Wahl der Explorationswahrscheinlichkeit ϵ muss eine Balance zwischen Exploration und Exploitation, d. h. Ausnutzung der Aktionen mit höchsten Q-Value, ermittelt werden. Das Abwägen beider Aspekte wird als Explorations-Exploitations-Dilemma bezeichnet [3, S. 3].

Der Ablauf des Sarsa Algorithmus ist als Pseudocode formuliert in Algorithmus 1.[7]

Algorithmus 1 : Sarsa

Algorithm parameters : step size $\alpha \in (0; 1]$, small $\epsilon > 0$

Initialize $Q(s, a)$, for all $s \in S^+, a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$

foreach *episode* **do**
    Initialize S
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    foreach *step of episode* **do**
        Take action A, observe R, S'
        Choose A' from S' using policy derived from Q (e.g., ϵ-greedy)
        $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma Q(S', A') - Q(S, A)]$
        $S \leftarrow S'; A \leftarrow A';$
    end
end

[6] Gibt es mehrere Aktionen mit dem höchstenQ-Value, wird willkürlich eine der Aktionen gewählt
[7] eigene Darstellung in Anlehnung an [3, S. 130]

2.2.3. Q-Learning

Q-Learning lernt ebenfalls die optimale Policy durch Schätzung der Action-Value Funktion. Im Gegensatz zu Sarsa approximiert Q-Learning jedoch direkt die optimale Action-Value Funktion q_* und somit die optimale Policy π_*. Dafür nutzt Q-Learning bei der Aktualisierung seines Schätzwertes $Q(S_t, A_t)$ immer die Aktion a des Folgezustands S_{t+1}, die den maximalen Q-Value hat. Dies ist unabhängig davon welche Aktion A_{t+1} der Agent schlussendlich ausführt. [14, S. 177] Das Update für Q-Learning kann daher wie in Gleichung (2.11) formuliert werden[3, S. 131].

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.11}$$

Die Policy π, die zur Interaktion mit der Umgebung genutzt wird, ist jedoch dennoch wichtig, da sie entscheidet welche Samples der Agent erhält. Da für die Aktualisierung jedoch immer der maximale Q-Value genutzt wird, ist eine Abnahme der Explorationswahrscheinlichkeit bei Nutzung von ϵ-greedy keine Bedingung für die Konvergenz zur optimalen Policy [3, S. 131f.]. Wenn die Konvergenzbedingungen von TD Prediction eingehalten werden, ist Q-Learning garantiert zur optimalen Policy π_* zu konvergieren [13, S. 286].

Der Ablauf des Q-Learning Algorithmus ist als Pseudocode formuliert in Algorithmus 2.[8]

Algorithmus 2 : Q-Learning

Algorithm parameters : step size $\alpha \in (0; 1]$, small $\epsilon > 0$
Initialize $Q(s, a)$, for all $s \in S^+, a \in A(s)$, arbitrarily except that $Q(terminal, \cdot) = 0$
foreach *episode* **do**
  Initialize S
  foreach *step of episode* **do**
    Choose A from S using policy derived from Q (e.g., ϵ-greedy)
    Take action A, observe R, S'
    $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$
    $S \leftarrow S'$
  end
end

2.2.4. Vergleich von Q-Learning und Sarsa

Die Algorithmen Q-Learning und Sarsa haben zwei zentrale Unterschiede. Zum einen die Aktion des Folgezustands und somit der Q-Value, der für das Update genutzt wird. Zum anderen der Zeitpunkt, zu dem das Update durchgeführt wird [15]. Sarsa nutzt für das Update die Aktion, die es gemäß seiner Policy auswählt. Hingegen nutzt Q-Learning für das Update immer die Aktion mit dem höchsten Q-Value, auch wenn es gemäß Policy eine andere Aktion ausführt. Aufgrund dessen wird Sarsa als ein On-Policy und Q-Learning als ein Off-Policy Algorithmus klassifiziert [3, S. 132].

[8] eigene Darstellung in Anlehnung an [3, S. 131]

Zur Veranschaulichung des Unterschiedes wird das Update von Sarsa und Q-Learning am Beispiel von Gridworld erklärt. Abb. 2.5 zeigt die ϵ-greedy-Policy, nach der beide Agent mit der Umgebung interagieren. Die State-Transition Probability und Rewardfunktion sind unverändert, aber den Agenten nicht bekannt. Es gilt $\gamma = 0,9$ und $\alpha = 0,1$. Der blaue Pfeil zeigt den Pfad einer Episode für beide Agenten. Tabelle 2.1 enthält einen Ausschnitt einer willkürlich gefüllten Q-Tabelle für die Zustände C1 und C2.

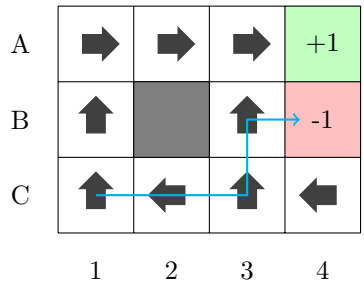


Abbildung 2.5.: Sample Episode für Gridworld[9]

Zustand S	Aktion A	$Q(S,A)$
C1	Norden	10,5
	Westen	5,0
	Süden	8,7
	Osten	0,1
C2	Norden	8,2
	Westen	-2,0
	Osten	9,8
	Süden	6,5

Tabelle 2.1.: Willkürlich gefüllte Q-Tabelle für Gridworld

Im Zustand C1 wählt der Agent die Greedy-Aktion "Norden". Wegen der Transposition-Probability wechselt der Agent jedoch in den Zustand C2. Im Zustand C2 ist die Greedy-Aktion "Osten". Der Agent wählt aufgrund von ϵ-greedy zufällig die Aktion "Westen". Das Sample $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$ der Umgebung ist somit: $(C1; N; -0,1; C2; W)$

Der Sarsa-Agent führt das Update in Gleichung (2.12) durch. Hingegen wählt der Q-Learning Agent für den Folgezustand die Aktion mit höchstem Q-Value $Q(C2, Osten) = 9,8$ führt daher das Update aus Gleichung (2.13) durch.

$$\begin{aligned} Q(C1,N) &= Q(C1,N) + \alpha[-0,01 + 0,09Q(C2,W) - Q(C1,N)] \\ Q(C1,N) &= 10,5 + 0,1[-0,1 + 0.9(-2,0) - 10.5] = 9,26 \end{aligned} \tag{2.12}$$

$$\begin{aligned} Q(C1,N) &= Q(C1,N) + \alpha[-0,01 + 0,09Q(C2,O) - Q(C1,N)] \\ Q(C1,N) &= 10,5 + 0,1[-0,1 + 0.9(9,8) - 10.5] = 10,32 \end{aligned} \tag{2.13}$$

Würden beide Algorithmen einer Greedy-Policy folgen, wäre das Update gleich. Während Sarsa seine Aktion A_{t+1} vor dem Update ausgewählt hat, wählt Q-Learning diese erst, nachdem das Update durchgeführt wurde. In Situationen, in denen ein Zustand der eigene Folgezustand ist, könnten Q-Learning und Sarsa unterschiedliche Aktionen A_{t+1} auswählen, da sich die Q-Values der Aktionen durch das Update geändert haben könnten. [15]

Ein weiteres Beispiel, dass den Unterschied von Q-Learning und Sarsa verdeutlicht ist „Cliff Walking" in Abb. 2.6. Bei diesem soll ein Agent vom Startbereich "S" zum Ziel "G" laufen. In

[9] eigene Darstellung in Anlehnung an [11, S. 127]

jedem Zeitschritt erhalten die Agenten einen Reward -1, sodass sie das Ziel möglichst schnell erreichen sollen. Betritt ein Agent den Bereich Klippe, erhält dieser einen Reward von -100. Die Agenten können sich wieder in alle vier Himmelsrichtungen bewegen. Im Gegensatz zur Gridworld sind die Zustandsübergänge deterministisch. Die Algorithmen Q-Learning und Sarsa lernen beide mittels einer ϵ-greedy-Policy mit konstantem ϵ=0,1. Die Abbildung zeigt den optimalen Pfad (rot) und den sicheren Pfad (blau), der einen geringeren Return erzielt, aber mit größerer Wahrscheinlichkeit im Ziel ankommt. Während des Trainings lernt Q-Learning den optimalen Pfad, betritt jedoch aufgrund von $\epsilon = 0,1$ regelmäßig die Klippe. Hingegen lernt Sarsa den sicheren Pfad, da es im Update die aktuelle Policy beachtet. Konvergiert ϵ während dem Training gegen 0 lernt Sarsa ebenfalls den optimalen Pfad. [3, S. 132]

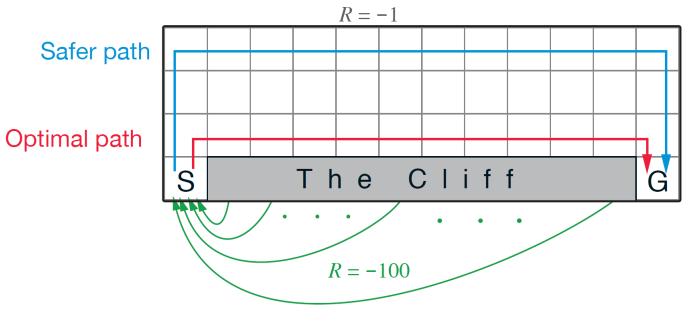


Abbildung 2.6.: Cliff Walking[10]

Beide Algorithmen haben jedoch die gleichen Limitierungen. Da beide Algorithmen durch Interaktion mit der Umgebung lernen, werden hinreichend viele Samples benötigt, um die Action-Value Funktion zu approximieren. Zudem spielt die Wahl der Lernrate eine entscheidende Rolle für den Trainingserfolg. [11, S. 131] Da für jedes SA Tupel ein eigener Eintrag in der Q-Tabelle vorhanden sein muss, haben die Algorithmen zwei weitere Limitierungen. Erstens, können beide Algorithmen nur für Probleme genutzt werden, bei denen eine Verwaltung der zu speichernden SA Tupel praktikabel ist. Zweitens können die Algorithmen nur optimale Entscheidungen treffen, wenn sie die SA Tupel kennen. Eine Generalisierung der gesammelten Erfahrung auf neue Zustände ist nicht möglich. Um beide Limitierungen zu beheben können die Algorithmen mit neuronalen Netzten verknüpft werden. Diese werden dann genutzt, um die State-Value Funktion zu approximieren und so keine SA Tupel speichern zu müssen und gesammelte Erfahrung zu generalisieren. [3, S. 195f.], [11, S. 137ff.]

2.3. Minimax-Algorithmus

Jedes Zug-basierte Spiel kann als ein gerichteter Graph dargestellt werden, der als Spielbaum bezeichnet wird. Jeder Knoten im Spielbaum entspricht einem legalen Spielzustand. Der Ausgangszustand des Spielfelds ist der sogenannte Wurzelknoten. Die Knoten sind verbunden durch Kanten, die die legalen Aktionen und somit Zustandsübergänge repräsentieren. Nehmen zwei Spieler abwechselnd Aktionen vor, entscheidet die Tiefe des Spielbaums ausgehend vom Wurzelknoten, welcher Spieler am Zug ist. Spielendzustände werden durch Terminal-

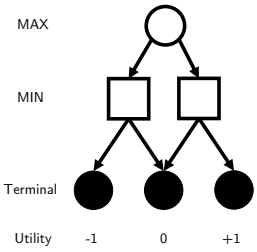
[10] Abbildung entnommen aus [3, S. 132]

knoten repräsentiert, von denen keine weiteren Kanten ausgehen. [16, S. 650f.], [2, S. 123 ff.]

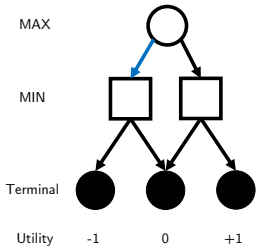
Jedem Terminalknoten bzw. Spielergebnis kann mittels einer Bewertungsfunktion ein Wert zugewiesen werden. Dieser wird als Utility bezeichnet und ist abhängig von der Art des Spiels, das der Baum repräsentiert. Ein solcher Baum kann zur Darstellung eines Nullsummenspiels genutzt werden. [16, S. 650f.], [2, S. 123 ff.] Ein Nullsummenspiel, wie beispielsweise Tic-Tac-Toe oder Schach, ist ein Spiel, bei dem der Gewinn des einen Spielers ein Verlust für die anderen Spieler bedeutet [8, S. 6]. Somit gibt es zwei äquivalente Möglichkeiten das Ziel jedes Spielers ausdrücken: Ein Spieler versucht zu gewinnen bzw. dafür zu sorgen, dass der Gegner verliert. Als Utility wird bei Zweispieler-Nullsummenspielen üblicherweise der Wert $+1$ für den Sieg des beginnenden Spielers und -1 für den Sieg des nachziehenden Spielers genutzt. Ein Unentschieden hat die Utility 0 [16, S. 649ff.], [2, S. 123 ff.]

Werden die Ziele der Spieler auf Basis der Utility formuliert, so versucht der beginnende Spieler die Utility zu maximieren, d. h. das Spiel mit $+1$ zu beenden. Der nachziehende Spieler möchte, dass der Gegner verliert und somit die Utility minimieren, d. h. das Spiel soll mit mit -1 enden. Geht man von zwei optimalen Spielern aus, werden diese in jedem Knoten entsprechend die Aktion wählen, die die Utility maximiert oder minimiert. [2, S. 124] Dieser Sichtwechsel zwischen Maximieren und Minimieren bildet die Grundlage des Minimax-Algorithmus. [16, S. 655] Der Minimax-Algorithmus basiert auf dem Minimax-Theorem der Spieltheorie [17] und wurde erstmals 1950 in [18] für Schach formuliert. Es ist ein Verfahren um in Spielbäumen von Zweispieler-Nullsummenspielen die optimale Aktion zu finden, unter der Annahme, dass der Gegner ebenfalls seine optimale Aktion wählt [18, S. 7].

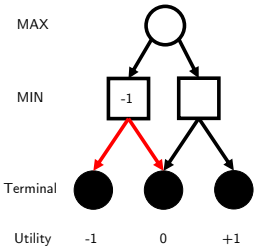
Minimax ist ein rekursiver depth-first (dt. Tiefensuche) Algorithmus, der im Wurzelknoten beginnt. Je nach Rolle bzw. Tiefe des Knoten, wird die minimale oder maximale Utility des Kindknoten und die dafür notwendige Aktion übernommen. Dafür betrachtet der Algorithmus die Utility aller Kindknoten. Da nur Terminalknoten eine Utility durch die Bewertungsfunktion zugewiesen wird, werden solange die Kindknoten jedes Knoten betrachtet, bis die Terminalknoten erreicht wurden. Der Elternknoten übernimmt dann die Utility des Kindknoten mit dem minimalen oder maximalen Wert, sodass der darüber liegende Knoten die zu erwartende Utility erhält und wählen kann. [2, S. 123ff.], [16, S. 654ff.] Dieser Prozess ist visualisiert in Abb. 2.7 und als Pseudocode beschrieben in Algorithmus 3.



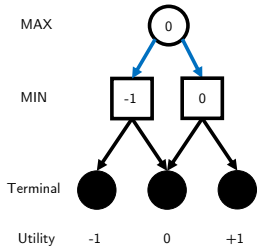
(a) Spielbaum im Ausgangszustand



(b) MAX prüft Utility der Kindknoten



(c) MIN prüft Utility der Terminalknoten und übernimmt kleinste Utility



(d) b) und c) werden für rechte Seite wiederholt und MAX wählt größte Utility aus

Abbildung 2.7.: Visualisierung des Minimax-Algorithmus für ein fiktives Zweispieler-Nullsummenspiel in vier Schritten. Knoten des maximierenden Spielers (MAX) sind Kreise, Knoten des minimierenden Spielers (MIN) Quadrate.

Algorithmus 3 : Pseudocode Minimax-Algorithmus

```
1  Function Minimax(NODE, PLAYER):
2      if NODE is terminal then
3          return utility assigned to NODE
4      else if PLAYER is maximizing player MAX then
5          foreach child of NODE do
6              Minimax(CHILD, MIN)
7          end
8          return maximal CHILD utility
9      else
10         foreach child of NODE do
11             Minimax(CHILD, MAX)
12         end
13         return minimal CHILD utility
14     end
15 end function
```

Da manche Knoten in Spielbäumen durch unterschiedliche Zugfolgen erreicht werden können, kann der Minimax-Algorithmus durch eine Transposition table (dt. Zugumstellungs-Tabelle) erweitert werden. Bei diesen werden Informationen zu Knoten in einer Lookup-Tabelle hinterlegt, sodass direkt auf diese zugegriffen werden kann. [16, S. 651], [19, S. 22], [20, S. 807] Der Name Transposition table leitet sich vom Schach ab, bei dem unterschiedliche Spielzüge, um einen gleichen Zustand zu erreichen, als Transposition (dt. Zugumstellung) bezeichnet werden [16, S. 651]. Andere Vereinfachungen und Erweiterungen des Minimax-Algorithmus, wie beispielsweise „Negamax“ oder Alpha-Beta-Pruning [1, S. 115] werden in dieser Arbeit nicht betrachtet.

2.4. Tic-Tac-Toe

In diesem Abschnitt wird das Strategiespiel Tic-Tac-Toe (TTT) erklärt. Anschließend wird begründet, wieso die vorgestellten Algorithmen darauf angewandt werden können.

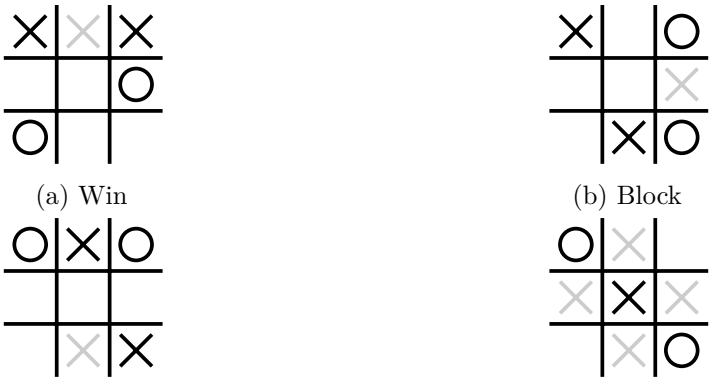
2.4.1. Spielerklärung

Tic-Tac-Toe ist ein Strategiespiel, das von zwei Spielern gespielt wird [8, S. 6]. Das Spielfeld ist unterteilt in neun Bereiche (Slots), die in einer 3x3 Matrix angeordnet sind. Die Spieler platzieren abwechselnd ihr zugeteiltes Symbol, entweder X oder O, in einem Slot auf dem Spielfeld. Der beginnende Spieler verwendet in der Regel das Symbol X. Ziel jedes Spielers ist es, drei eigene Symbol entweder vertikal, horizontal oder diagonal in einer Reihe zu platzieren. Somit gibt es insgesamt acht mögliche Gewinnkonstellationen: drei Zeilen, drei Spalten und zwei Diagonalen. Sind alle Slots belegt ohne, dass einer der Spieler gewonnen hat, endet das Spiel in einem Unentschieden. [9, S. 533]

TTT besitzt 5478 legale Spielfeldkonstellationen und hat im Vergleich zu Spielen wie Schach oder Vier Gewinnt eine deutlich geringere Zustandsraum-Komplexität [8, S. 159]. Dies kann sogar weiter reduziert werden, da aufgrund von Rotations- und Spiegelsymmetrien bis zu acht Spielfeldkonstellationen äquivalent sein können.[11] Dadurch reduziert sich die Zustandsraum-Komplexität auf 765. [10, S. 3]

In der Spieltheorie ist TTT ein klassisches Beispiel für ein Nullsummenspiel, da ein Gewinn immer mit einer Niederlage für den Gegner einhergeht [8, S. 6], [9, S. 533]. Zudem gehört TTT zu den Spielen mit perfekter Information, weil beide Spieler Zugriff auf alle Informationen des aktuellen Spielzustands haben [8, S. 156], [21, S. 38]. TTT ist ein stark gelöstes Spiel, d. h. es gibt eine optimale Strategie für jede legale Position [8, S. 6] Spielen beide Spieler optimal endet TTT immer in einem Unentschieden und wird daher als futile Game (dt. vergebliches Spiel) bezeichnet [22, S. 177]. Somit ist das beste Spielergebnis immer zu Gewinnen und gegen einen optimalen Spieler Unentschieden zu spielen. Regeln für ein perfektes Spiel wurden erstmals 1972 in [23] formuliert. Crowley und Siegler erweiterten diese unter der Bezeichnung „Model of Expert Performance“, wie im Anhang Anhang A gelistet [9, S. 536]. Die folgenden vier Regeln bzw. Aktionen bilden in der dargestellten Reihenfolge die zentrale Strategie für Expert Play. Beispielhafte Spielfeldkonstellationen werden in Abbildung Abb. 2.8 gezeigt.

1. **Win**: Wenn der Spieler eine Möglichkeit hat zu gewinnen, soll diese genutzt werden
2. **Block**: Wenn der Gegner zwei Symbole in einer Reihe hat, blockiere den Gewinn des Gegners
3. **Fork**: Wenn sich zwei Reihen schneiden, in denen der Spieler sein Symbol platziert hat und der Slot, wo sie sich schneiden leer ist, belege diesen Slot
4. **Block Fork**: Wenn die Möglichkeit besteht, zwei Symbole in eine Reihe zu legen, wähle diesen Slot, um den Gegner zum Blocken zu zwingen. Andernfalls wähle einen Slot, um ein Fork des Gegners zu verhindern



(a) Win (b) Block

(c) Fork (d) Block Fork

Abbildung 2.8.: Beispielhafte Spielfeldkonstellationen für TTT Expert Play. Optimale Aktionen sind grau gefärbt [12]

Der zweite Teil der Strategie beschäftigt sich mit dem optimalen ersten Zug für X. Aufgrund der Äquivalenz der möglichen Aktionen reduziert sich die Auswahl auf drei Möglichkeiten: Ecke, Kante und Mitte. In [21, S. 38] wird die Ecke als stärkste Aktion dargestellt, da der Gegner mit der Mitte kontern muss, um nicht zu verlieren. Hingegen zeigen Studien wie [25], dass die Mitte die beste Aktion ist, da diese in der Menge aller möglichen Spielabfolgen die Startposition mit den meisten gewonnen Spielen ist. Spielen zwei optimale Spieler gegeneinander ist die Wahl der ersten Aktion unbedeutend [21, S. 38].

2.4.2. Anwendbare Algorithmen

Eine Voraussetzung für die Anwendung des Minimax-Algorithmus ist, dass der Spielbaum eine geringe Komplexität hat [10, S. 10]. Die Spielbaum-Komplexität ist definiert als die Anzahl der Terminalknoten des vollständig konstruierten Spielbaums und wird angegeben als log zur Basis 10 [8, S. 160]. Aufgrund des kleinen Zustands- und Aktionsraums von TTT, ist dessen Spielbaum klein. Die Spielbaum-Komplexität beträgt 5 ist im Vergleich zu Spielen wie Schach (123) oder Shogi (226) relativ gering [10, S. 11]. Somit erfüllt TTT diese Bedingung für die Anwendung des Minimax-Algorithmus. Eine weitere Bedingung des Minimax-Algorithmus ist, dass das Spiel ein Zweispieler-Nullsummenspiel mit perfekter Information sein muss [10, S. 10]. Da TTT auch diese Bedingung erfüllt, kann der Minimax-Algorithmus angewendet werden. In Abb. 2.9 ist ein Auszug des Spielbaums von TTT mit der möglichen Umsetzung eines Minimax-Algorithmus dargestellt. Jedoch ist anzumerken, dass der Minimax-Algorithmus einen optimalen Gegner erwartet und daher nur gegen diese optimal spielt. Minimax kann nicht verlieren, aber wählt seine Aktionen nicht, um seine Gewinnwahrscheinlichkeit gegen nicht optimale Spieler zu maximieren. [12, S. 8].

Voraussetzung für die Anwendung von TDL Algorithmen ist, dass das zu lösende Problem als MDP formuliert werden kann. Der mögliche Zustandsraum S und Aktionsraum A sind für TTT diskret und bekannt. Da TTT ein Spiel mit perfekter Information ist, kann bei gegebenem

[11] 4 Rotationen und 4 Spiegelungen, jeweils horizontal/vertikal mit und ohne 90° Drehung

[12] Spielfeldkonstellationen übernommen aus [9, S. 539], Alle TTT Spielfelder werden erstellt mit Code von [24]

Zustand s und Aktion a der Folgezustand s' bestimmt werden. Somit erfüllt TTT die Markov Eigenschaft und kann als ein MDP modelliert werden. Die weitere Voraussetzung ist, dass der Agent mit seiner Umgebung interagieren kann und von dieser Samples in Form des Tupels $(S_t, A_t, R_{t+1}, S_{t+1})$ erhält. [11, S. 88] Diese kann im Rahmen der Implementierung beachtet werden. Da somit beide Voraussetzungen erfüllt sind, können TDL Methoden wie Q-Learning und Sarsa angewendet werden.

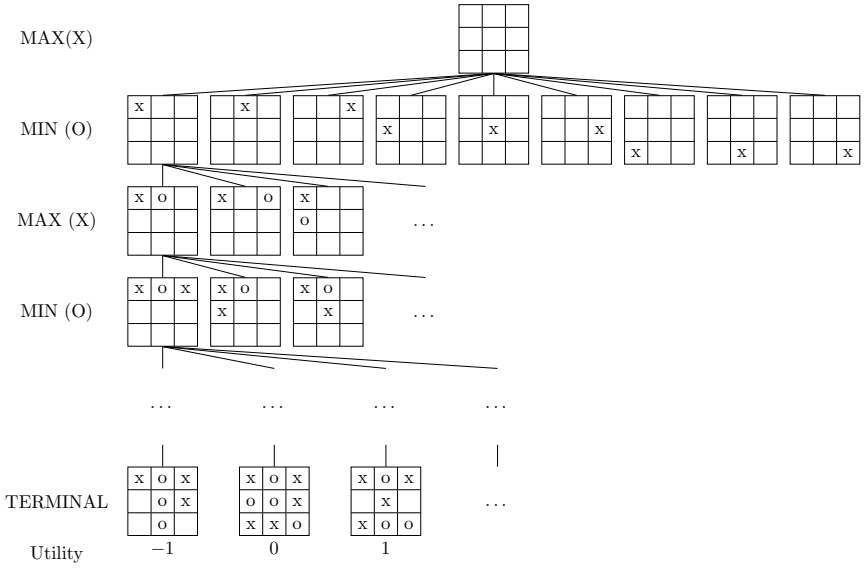


Abbildung 2.9.: Auszug aus dem Spielbaum von TTT [13]

[13] Darstellung übernommen aus [2, S. 125], Abbildung erstellt mit Code von [26]

3. Methodik und Anwendung der Algorithmen

In diesem Kapitel wird erklärt, wie die vorgestellten TD Konzepte auf TTT angewandt werden können. Anschließend wird beschrieben, wie TTT als Umgebung genutzt wird. Abschließend wird die Vorgehensweise beim Training und der Evaluation vorgestellt.

3.1. Anwendung von Temporal-Difference Learning auf Tic-Tac-Toe

Die folgenden Abschnitte behandeln die Umsetzung des Temporal-Difference Update und einer möglichen Rewardfunktion. Anschließend wird der Aufbau der Q-Tabelle erklärt und das Konzept der Afterstates eingeführt.

3.1.1. Temporal-Difference Update und Rewardfunktion

Zur Anwendung von Reinforcement Learning muss definiert werden, was die Bestandteile des MDP sind. In Tic-Tac-Toe entspricht das Spielfeld der Umgebung und ein Zustand s ist eine Konstellation des Spielfeldes. Der Agent ist ein Spieler, der als Aktion einen freien Slot wählt, in dem das zugewiesene Symbol platziert wird. Die Menge der legalen Aktionen in einem Zustand $A(s)$ entspricht den unbelegten Slots auf dem Spielfeld. Ein Spiel kann aus maximal neun Zeitschritten bestehen und endet immer mit einem Ergebnis. TTT gehört daher zu den episodischen RL Problemen.

Die Abweichung von klassischen RL Problemen ist, dass zwei Agenten abwechselnd Symbole platzieren. Da das klassische TDL nicht für Multi-Agenten Probleme definiert ist, kann es nicht direkt angewendet werden [4, S. 2f.]. Eine mögliche Lösung bietet die Agenten-Umgebung-Schnittstelle. Demnach kann der Gegner und dessen gewählte Aktionen als Teil der Umgebung betrachtet werden, da der Agent diesen nicht kontrolliert. Somit erhält der Agent von der Umgebung nur die Zustände, in denen er eine Aktion wählen kann. Aus Sicht des Agenten ist TTT eine stochastische Umgebung, da der Zug des Gegners und somit der nächste Zustand, den ein Agent erhält nicht deterministisch sind. [4, S. 2f], [27] Unter der Voraussetzung, dass X beginnt, können alle Zustände gerader Zeitschritte bzw. Symbolanzahl dem Agenten X und ungerade dem Agent O zugeordnet werden, wie in Abbildung Abb. 3.1 dargestellt. Die zugehörigen MDP beziehen sich nun auf einen Agenten, sodass die vorgestellten Algorithmen angewendet werden können.

Für das TD Update ist neben dem jetzigen Zustand $S_t$und der gewählten Aktion a_t der nächste Zustand S_{t+2}, den der Agent erhält, notwendig. Da der nächste Zustand des Agenten abhängig von der Aktion des Gegners ist, erfolgt ein Update für einen Zustand erst, wenn der Gegner eine Aktion durchgeführt hat. Eine Besonderheit sind Terminalzustände, da diese das Spiel beenden und somit keine Aktion durch den Gegner möglich ist. Ein Terminalzustand S_T wird erreicht,

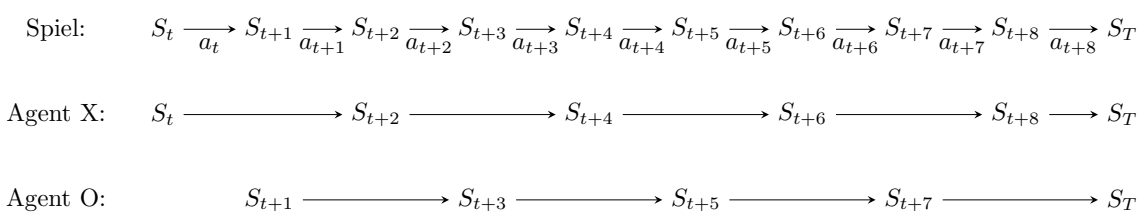


Abbildung 3.1.: Zuordnung der TTT Zustände und Zeitschritte zu den Agenten

wenn ein Spieler eine Aktion wählt, durch die er das Spiel gewinnt oder in einem Unentschieden beendet. Wechselt ein Agent durch eine Aktion in einen Terminalzustand, wird dieser für das Update beider Agenten verwendet. [4, S. 2f], [27] Dies ist möglich, da der Q-Value jedes Terminalzustands per Definition 0 ist [3, S. 74]. Die Unterscheidung, ob ein Spielergebnis gut oder schlecht für den Agenten ist, erfolgt im Update durch den Reward.

Beim Minimax-Algorithmus erhalten beide Symbole die gleiche Utility, weshalb es einen minimierenden und maximierenden Spieler gibt. Die TD Agenten hingegen versuchen ihren Return zu maximieren, sodass Reward positiv oder negativ, entsprechend des Spielergebnisses aus Agenten-Sicht sein muss. TTT ist ein Nullsummenspiel mit drei möglichen Spielergebnissen: Gewinn, Niederlage und Unentschieden. Wie in Abschnitt 2.3 beschrieben, ist eine gängige Zuordnung $+1$, -1 und 0. Das Spielverhalten kann jedoch weiter differenziert werden durch Betrachtung der Spiellänge, gemessen in durchgeführten Aktionen. Ein optimaler Agent sollte versuchen, in möglichst wenigen Zügen zu gewinnen und eine Niederlage möglichst lange hinauszögern [9, S. 533]. Eine mögliche Umsetzung ist, den Reward je Spielzug in der Episode um 0.1 gegen 0 zu korrigieren. Als Korrektur wird 0.1 verwendet, da neun Spielzüge möglich sind und selbst bei Gewinn im letzten Zug ein positiver Reward zugewiesen wird. Diese Korrektur wird im Folgenden als „Depth Penalty“ bezeichnet, da eine Korrektur entsprechend der Tiefe (engl. depth) im Spielbaum erfolgt. Dieser Reward bewertet, ob der Agent das Ziel, TTT zu Gewinnen, erreicht hat und wird daher als letzter Reward einer Episode ausgeteilt. Die Rewards aller anderen Aktionen sind 0, da deren Bedeutung für das Spielergebnis nicht bekannt ist. Der Reward kann somit mittels Gleichung (3.1) berechnet werden und liefert die in Tabelle 3.1 gelisteten Rewards. Da der unkorrigierte Reward für Sieg und Niederlage 1 und -1 ist, kann er genutzt werden, um das Vorzeichen der Korrektur anzupassen. Andere Definitionen eines Rewards sind möglich, werden in dieser Arbeit jedoch nicht betrachtet [28].

$$\text{reward}_{\text{angepasst}} = \text{reward} - (\text{reward} \cdot \text{depth} \cdot 0,1) \tag{3.1}$$

3.1.2. Q-Tabelle und Kompression durch Afterstates

Die Q-Tabelle enthält die Abbildung der SA Tupel auf einen Q-Value. Da jeder Zustand und somit jedes SA Tupel eindeutig einem Symbol zugeordnet werden kann, können die Q-Values für beide Symbole in einer gemeinsamen Q-Tabelle gespeichert werden. Wird die Q-Tabelle als eine Lookup-Tabelle visualisiert, wie in Tabelle 3.2, enthält diese für jeden der 5478 Zustände

Tabelle 3.1.: Mögliche Rewards pro Spielzug bei Korrektur des Rewards mittels Depth Penalty

Zug	Reward Agent	Reward Gegner
1	/	/
2	/	/
3	/	/
4	/	/
5	0,5	-0,5
6	-0,4	0,4
7	0,3	-0,3
8	-0,2	0,2
9	0,1	-0,1

Tabelle 3.2.: Q-Tabelle für TTT. S_{5477} ist ein Beispiel für einen Terminalzustand, in dem keine Aktionen möglich sind, dargestellt durch „/“

State	a_0	a_1	...	a_8
S_0	$Q(S_0, a_0)$	$Q(S_0, a_1)$	...	$Q(S_0, a_8)$
S_1	$Q(S_1, a_0)$	$Q(S_1, a_1)$	...	$Q(S_1, a_8)$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
S_{5477}	/	/	...	/

eine Zeile. Die Aktionen sind die Spalten, wobei die Aktionen nicht für jeden Zustand gleich sind, sondern den freien Slots auf dem Spielfeld entsprechen.

Wie im vorigen Abschnitt beschrieben, besteht eine Runde aus einem deterministischen Zustandswechsel durch den Agenten und einem stochastischen Zustandswechsel durch die Umgebung. In TTT können unterschiedliche SA Tupel dieselbe Spielfeldkonstellation erzeugen, bevor der Gegner zum Zug kommt, wie in Abb. 3.2 dargestellt. Die Zustände nach dem deterministischen Zustandswechsel durch den Agenten werden als Afterstate (dt. Folgezustand) bezeichnet und können als surjektive Abbildung der SA Tupel auf die Menge der Afterstates Y beschrieben werden: $(S, A) \rightarrow Y$. SA Tupel, die denselben Afterstate haben, sollten einen gemeinsamen Q-Value besitzen. Dadurch wird Erfahrung, die durch ein SA Tupel gesammelt wird, mit den anderen geteilt, sodass der Lernprozess beschleunigt wird. [3, S. 136f.]

Mit der vorgestellten Q-Tabelle ist dies jedoch nicht möglich, da die Q-Funktion jedem SA Tupel einen individuellen Q-Value zuweist und der resultierende Zustand dem anderen Symbol zugeordnet ist. Daher wird eine Afterstate-Value Funktion genutzt, die statt einem SA Tupel (S, A) dem resultierenden Afterstate y einen Wert zuweist. Der zugewiesene Wert kann, zur Unterscheidung vom Q-Value, als W-Value w bezeichnet werden.[29]

Ein Agent wählt nun in einem Zustand, die Aktion, die im Afterstate mit dem höchsten W-Value resultiert. Die bisherige Q-Tabelle wird geteilt in zwei separate Tabellen, wie in Tabelle 3.3 und Tabelle 3.4 dargestellt. Einerseits die surjektive Abbildung $(S, A) \rightarrow Y$ sowie die Zuweisung des W-Value zu einem Afterstate: $y \rightarrow w$, die im Folgenden als Afterstate-Tabelle und W-Tabelle

[1] Abbildung entnommen aus [3, S. 137]

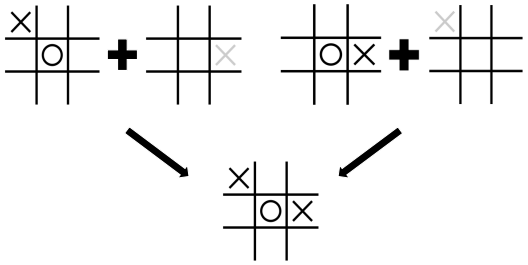


Abbildung 3.2.: Beispiel für Afterstates in TTT. Zwei SA Tupel können im selben Afterstate resultieren [1]

bezeichnet werden. Im Gegensatz zur Q-Tabelle umfasst der W-Tabelle nur 5477 Zeilen, da der Ausgangszustand kein Afterstate ist. Die Anzahl der W-Values ist im Vergleich zur Anzahl von Q-Values deutlich komprimiert, da SA Tupel, die denselben Afterstate y haben einen W-Value teilen. Streng genommen benötigt ein Agent mit Afterstate-Value Funktion nur die Menge der erreichbaren Afterstates und nicht den Zustand oder die legalen Aktionen selbst, da diese durch die Abbildung $(S, A) \rightarrow Y$ implizit enthalten sind. [29]

Tabelle 3.3.: Afterstate-Tabelle für TTT

State	a_0	a_1	$\dots$	a_8
S_0	y_0	y_1	$\dots$	y_8
S_1	y_1	y_0	$\dots$	$\dots$
$\vdots$	$\vdots$	$\vdots$	$\ddots$	$\vdots$
S_{5477}	$\dots$	$\dots$	$\dots$	$\dots$

Tabelle 3.4.: W-Tabelle für TTT

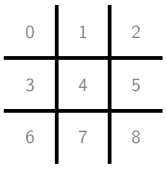
Afterstate	W-Value
y_0	$w(y_o)$
y_1	$w(y_1)$
$\vdots$	$\vdots$
y_{5476}	$w(y_{5476})$

3.2. Kodierung von Tic-Tac-Toe

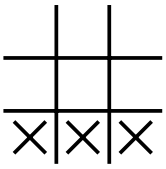
Das Tic-Tac-Toe Spielfeld besteht aus neun Slots, die entweder leer oder durch ein Symbol belegt sind. Ein Spielfeld kann daher als Folge von Bits betrachtet werden. Diese Form der Kodierung wird als Bitboard bezeichnet und genutzt zur speichereffizienten Beschreibung von Spielfeldern, die mit Bitoperationen manipuliert werden können. [19, S. 86]

Werden die Slots, beginnend bei 0, von oben links nach unten rechts indexiert, kann der Zustand eines Spielfeldes als neunstellige Binärzahl B_G ausgedrückt werden. Der Index eines Slots ist der Exponent zur Basis 2. Beispielsweise beschreibt der Zustand 0_2 ein leeres Spielfeld und 111000000_2 die Konstellation, in der die Slots $0-5$ frei sind und die letzte Reihe gefüllt ist, wie in Abb. 3.3 dargestellt.

Jedoch sind neun Bits nicht ausreichend, da zwischen zwei verschiedenen Symbolen und somit insgesamt drei Zuständen pro Slot unterschieden werden muss. Daher muss die Konstellation des Spielfeldes für jedes Symbol in einem dedizierten Bitboard kodiert werden, die im Folgenden als B_X und B_O bezeichnet werden.

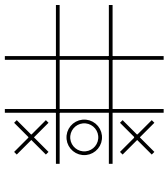


(a) Indexierung der Slots



(b) Zustand 111000000_2

Abbildung 3.3.: TTT Indexierung und Beispiel für B_G



$B_G = 111000000_2$
$B_X = 101000000_2$
$B_O = 010000000_2$
$B_s = 101000000010000000_2 = 163968_{10}$

Abbildung 3.4.: Beispiel für die unterschiedlichen Spielfeldkodierungen

Durch Konkatenation der beiden Bitboards bzw. Binärzahlen ist die Kodierung des gesamten Spielfeldes mit Unterscheidung der Symbole für einen Zustand s in einer 18-stelligen Binärzahl möglich. Diese wird als B_s bezeichnet. Dabei wird festgelegt, dass die ersten neun Bits für B_O und die anderen für B_X stehen. Der Wert von B_s zur Basis 10 kann mit Gleichung (3.2) eineindeutig berechnet werden.

$$B_s = \sum_{i=9}^{17} 2^i \cdot \mathbb{1}_{X_i=true} + \sum_{i=0}^{8} 2^i \cdot \mathbb{1}_{O_i=true} \tag{3.2}$$

Durch Logische OR Verknüpfung der Bitboards beider Symbole kann die neunstellige Spielfeldkodierung B_G vom Anfang erreicht werden, die nicht zwischen Symbolen unterscheidet: $B_X \vee B_O = B_G$. Die Bits in B_G mit Wert False sind die freien Slots und kodieren die Menge der legalen Aktionen A. Somit hat ein Agent maximal neun mögliche Aktionen, die im Interval $[0; 8]$ liegen. Wird eine Aktion von einem Spieler durchgeführt, so wird der Index des von ihm belegten Slot in seinem eigenen Bitboard auf True gesetzt. Ein Beispiel für eine Kodierung einer Spielfeldkonstellation ist in Abb. 3.4 dargestellt.

Die acht möglichen Gewinnkonstellationen $M_i \mid i \in [1, 8]$ von TTT können durch Bitoperationen geprüft werden. Dafür wird eine logische AND Operation einer der Gewinnkonstellationen mit dem Bitboard eines Spielers durchgeführt und das Ergebnis auf Äquivalenz mit der ursprünglichen Gewinnkonstellation geprüft: $B_X \wedge M_i = M_i$. Ist die Äquivalenz dieser Gleichung gegeben, bedeutet dies, dass die Gewinnkonstellation M_i vorliegt

Da ein Spieler nur Aktionen durchführen kann, die eine Gewinnkonstellation für ihn selbst konstruieren, muss nach einer Aktion nur das Bitboard des Spielers geprüft werden, der die letzte Aktion durchgeführt hat. Liegt keine Gewinnkonstellation vor und das Bitboard B_G enthält keine Bits mit Wert False, endet das Spiel in einem Unentschieden.

3.3. Trainingsaufbau der Agenten

In den folgenden Abschnitten wird der verwendete Trainingsaufbau erklärt. Dies umfasst die Arten von Self-play sowie die Auswahl der Hyperparameter, die untersucht werden. Abschließend werden die genutzten Metriken vorgestellt.

3.3.1. Self-play

Zum Training spielt ein Agent wiederholt gegen Gegner. Die Spielstärke der Gegner sollte vergleichbar mit der des Agenten sein und während des Trainings steigen [30, S.1 ff.]. Da die Bereitstellung unterschiedlicher Gegner ansteigender Spielstärke für komplexere Strategiespiele nicht praktikabel ist, gibt es das Konzept des Self-play. [6, S. 565f.] [31, S. 1ff.] Im Self-play lernt der Agent das Spiel, indem er gegen eine Kopie von sich selbst spielt. Dadurch wird stetig gegen einen Gegner gleicher Stärke gespielt, ohne dass dieser zusätzlich implementiert werden muss [6, S. 565f.]. Jedoch hat Self-play das Problem, das beide Agenten gleichzeitig lernen und in lokalen Maxima stecken bleiben können. Dies hat zur Folge, dass das Lernen unvollständig ist und die Agenten frühzeitig mit dem Lernen aufhören und den Zustandsraum nicht ausreichend erkunden. [32, S. 16], [6, S. 565f.], [33]

Eine Möglichkeit den Zustandsraum repräsentativer zu proben, ist die Agenten abwechselnd lernen zu lassen. Während ein Agent lernt, nutzt der andere seine derzeitige greedy Policy[2]. Nach einer festen Anzahl von Episoden, die im Folgenden Batch genannt werden, wechseln die Rollen. [34] Dies wird wiederholt, bis die Trainingsepisodenanzahl erreicht wurde, wie im Algorithmus 4 dargestellt. Da zu Beginn des Trainings der Agent das Spiel nicht kennt, spielt der lernende Agent im ersten Batch gegen einen Spieler mit Zufallsstrategie.

Im Rahmen der Bachelorarbeit werden das klassische Self-play und das beschriebene Self-play mit alternierenden Rollen für Tic-Tac-Toe implementiert und in der Auswertung verglichen. Aufgrund von Vorexperimenten wird festgelegt, dass in beiden Fällen zum Training 150.000 Episoden gespielt werden. Ein Batch beim alternierendem Self-play umfasst 100 Episoden.

Algorithmus 4 : Pseudocode alternierendes Self-play

```
Input : agentX, agentO, training_episodes
learningAgent ← agentX
set hyperparameters of both agents according to their role
for episode ← 0 to training_episodes do
    play an episode of tic-tac-toe
    if episode mod batch_size = 0 then
        switch learningAgent
        set hyperparameters of both agents according to their role
    end
end
```

[2]Hyperparameter werden entsprechend gesetzt: $\alpha = \epsilon = 0$

3.3.2. Auswahl der Hyperparameter

Die Hyperparameter können entweder konstant sein oder im Verlauf des Trainings abnehmen. Zur Komplexitätsreduktion wird für die Verkleinerung von Hyperparametern nur eine Methode genutzt. Diese Methode zur degressiven Abnahme wird als Step-Based bezeichnet und ist durch folgende Formel für einen beispielhaften Parameter η definiert [35, S. 3] :

$$\eta(x) = \eta_{start}(\frac{\eta_{end}}{\eta_{start}})^{\frac{x}{x_{max}}} \tag{3.3}$$

Der Parameter η hat in der ersten Episode ($x = 0$) den Wert η_{start} und in der Episode x_{max} den Wert η_{end}. Mit steigender Episode x wird die Anpassung zunehmend kleiner. Zur weiteren Komplexitätsreduktion wird nur eine kleine Untermenge möglicher Parameterkombinationen untersucht, die durch Vorexperimente ausgewählt wurden.

Als Diskontierungsfaktor γ wird 1 verwendet, da TTT ein endliches Spiel und die Diskontierung somit nicht notwendig ist. Da die Definition des optimalen Verhaltens bereits durch den Reward erfolgt, wird γ nicht weiter betrachtet.

Die Lernrate α muss während des Trainings gemindert werden, um die Konvergenz der Algorithmen zu gewährleisten. Jedoch wird diese Konvergenzbedingung nur selten in der Praxis eingehalten. Zudem ist in nicht-stationären Umgebungen eine konstante (kleine) Lernrate zu bevorzugen. [3, S. 33] Da der Gegner vom Agent als Teil der Umgebung betrachtet wird und im klassischen Self-play die Agenten gleichzeitig lernen, kann TTT als nicht-stationäre Umgebung aufgefasst werden. In Vorexperimenten zeigte sich, dass konstante α genutzt werden können, wenn $\alpha < 0.3$ gilt. Daher werden $\alpha = 0,1$, $\alpha = 0,2$ sowie ein α, das zu Beginn 1 ist und dann gegen 0.1 konvergiert untersucht,

Der Explorationswahrscheinlichkeit ϵ muss so gewählt werden, dass möglichst alle SA Tupel besucht werden. Insbesondere für Sarsa ist es jedoch notwendig, dass ϵ keinen konstant hohen Wert hat, damit Exploitation erfolgen kann. Durch Vorexperimente wurde ermittelt, dass dies zuverlässig erfolgt, wenn zu Beginn $\epsilon = 1$ gilt und im Rahmen des Trainings abnimmt und nach zwei Drittel der Episoden $\epsilon = 0.1$ gilt.

3.4. Evaluation und Metriken

Zur Beantwortung der Forschungsfragen werden die Algorithmen mit unterschiedlichen Hyperparameterkombinationen hinsichtlich ihrer Konvergenz und Spielstärke bewertet. Außerdem soll ermittelt werden, was laut den Agenten die optimale erste Aktion für das Symbol X ist. Während der Trainings- und Evaluationsepsioden werden Daten gesammelt, auf Episoden-Ebene aggregiert und geloggt, sodass diese ausgewertet und geplottet werden können. Zum Plotten aller Liniendiagramme in dieser Arbeit wird die Python Bibliothek Matplotlib [36] verwendet. Das dafür angefertigte Jupyter Notebook wird auf Github.com/JonasBingel bereitgestellt.

Ein interessanter Aspekt beim Vergleich von RL-Algorithmen ist, wie schnell diese während des Trainings zur optimalen Policy konvergieren [3, S. 33]. Als Konvergenzmetrik wird die Rate

optimaler Aktionen, die vom Agent gewählt werden, genutzt. Mit zunehmendem Lernen des Agenten sollte diese gegen 1 konvergieren. Während den Trainingsepisoden werden daher alle Aktionen des lernenden Agenten durch den Minimax-Algorithmus bewertet. Da sich die Rate zwischen den Episoden stark unterscheiden kann, werden die Werte der einzelnen Episoden kumuliert betrachtet. Zur Berechnung der Rate optimaler Aktionen gibt es zwei Möglichkeiten, da der Agent mit Wahrscheinlichkeit ϵ explorative Aktionen wählt. Einerseits Gleichung (3.4), die die absolute Anzahl optimaler Aktionen inklusive explorativer Aktionen betrachtet. Andererseits Gleichung (3.5), die explorative Aktionen herausrechnet. Im Rahmen von Vorexperimenten wurde festgestellt, dass dies einen deutlichen Unterschied macht, wie Abb. 3.5 verdeutlicht. Der Graph für Gleichung (3.5) nimmt aufgrund der Explorationswahrscheinlichkeit ϵ einmalig den Wert 1 oder 0 an. Zu Beginn des Trainings gilt $\epsilon = 1$, sodass der Agent nur zufällige Aktionen wählt und folglich keine Rate für diese Episoden berechnet werden kann. Sinkt Epsilon mit fortschreitender Episodenanzahl, kann der Agent selbst eine Aktion wählen. Da der Agent zu Beginn jedoch nur einzelne Aktionen wählt, nimmt die Rate optimaler Aktionen kurzfristig Extremwerte an.

Die Formel in Gleichung (3.4) hat den Vorteil, dass der aktuelle Zustand des Agenten in dieser Episode erfasst wird. Jedoch bietet Gleichung (3.5) einen besseren Vergleichsmaßstab, da diese den Agenten als fertige Strategie bewertet, die während der Evaluation getestet wird. Für die Auswertung der Agenten wird in dieser Arbeit Gleichung (3.5) verwendet.

$$\text{Rate}_{\text{mit Exploration}} = \frac{\text{Anzahl optimaler Aktionen}}{\text{Anzahl Aktionen des Agenten}} \tag{3.4}$$

$$\text{Rate}_{\text{ohne Exploration}} = \frac{\text{Anzahl optimaler Aktionen ohne Exploration}}{\text{Anzahl Aktionen des Agenten} - \text{Anzahl explorativer Aktionen}} \tag{3.5}$$

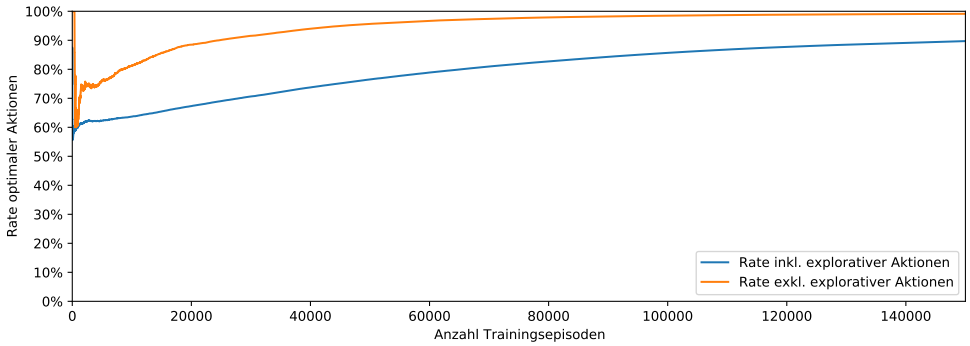


Abbildung 3.5.: Vergleich der Raten optimaler Aktionen

Um die Spielstärke der trainierten Agenten zu messen, spielt der Agent jeweils 10.000 Evaluationsspiele für beide Symbole X und O. Die Hyperparameter α, ϵ des Agenten werden auf 0 gesetzt, damit die greedy Policy genutzt wird und kein Lernen erfolgt. Zum einen spielt der Agent gegen den Minimax-Algorithmus. Sollten mehrere Aktionen optimal sein, wählt Minimax aus dieser Menge zufällig aus, sodass unterschiedliche Zustände evaluiert werden. Zum anderen spielt der Agent gegen einen Spieler mit Zufallstrategie Random, damit seine

Spielstärke gegen einen nicht optimalen Gegner ermittelt werden kann. Aufgrund von Vorexperimenten wurde festgelegt, dass fünf Agenten trainiert und das arithmetische Mittel von deren Evaluationsspielen genutzt wird, um eine bessere Vergleichbarkeit der trainierten Agenten zu ermöglichen.

Die Spielstärke kann quantifiziert werden, wenn die Spielergebnisse mit der in Abschnitt 3.1 beschriebenen Rewardfunktion bewertet werden. In den insgesamt 20.000 Spielen gegen Minimax, kann bestenfalls ein Gesamtreward von 0 erzielt werden, da Minimax optimal spielt und das Spiel somit nur in einer Niederlage oder einem Unentschieden enden kann. In Tabelle 3.1 wurde gezeigt, dass X frühestmöglich nach fünf und O nach sechs Zügen gewinnen kann. Unter der unrealistischen Annahme, dass ein Spieler jedes Spiel gegen Random in möglichst wenigen Aktionen gewinnt, beträgt der bestmögliche Reward somit 9.000, wie Gleichung (3.6) zeigt. Der schlechteste theoretische Wert ist -18.000 und beschreibt das Szenario, in dem jedes Spiel gegen Minimax und Random in einer schnellstmöglichen Niederlage endet. Das Intervall der Spielstärke beträgt somit $[-18.000; 9.000]$

$$\begin{aligned} \text{beste Spielstärke} &= 20.000 \cdot 0 + 10.000 \cdot 0,5 + 10.000 \cdot 0,4 = 9.000 \\ \text{schlechteste Spielstärke} &= 20.000 \cdot -0,5 + 20.000 \cdot -0,4 = -18.000 \end{aligned} \tag{3.6}$$

4. Architektur und Implementierung

In den folgenden Abschnitten wird zunächst die allgemeine Architektur der Implementierung erklärt. Anschließend werden die wichtigsten Klassen vorgestellt, sodass der Trainings- und Evaluationsprozess nachvollzogen werden können.

4.1. Architektur

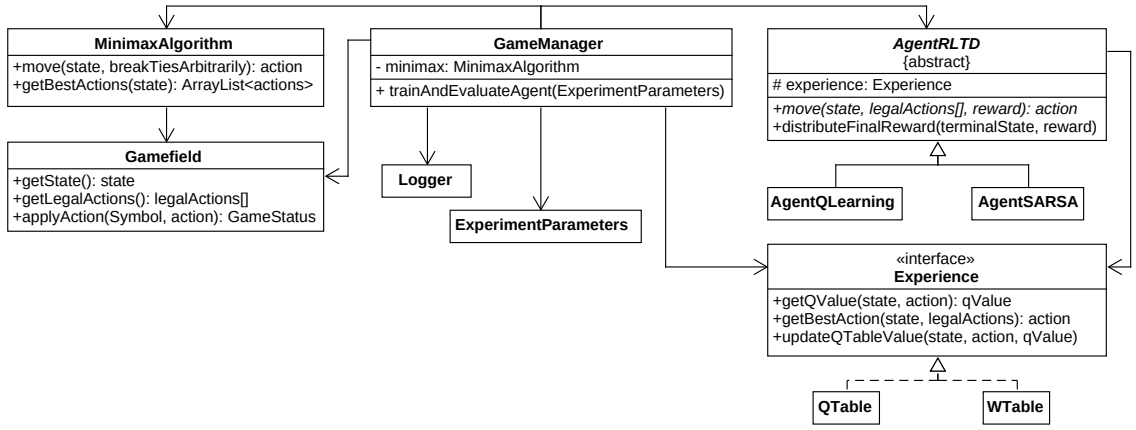


Abbildung 4.1.: Grobe Übersicht der wichtigsten Klassen und deren Interaktion

In Abb. 4.1 ist eine Übersicht der wichtigsten Klassen. Das UML-Diagramm stellt keinen Anspruch an Vollständigkeit, sondern dient der Vermittlung eines grundlegenden Verständnisses für die wichtigsten Bestandteile der Anwendung sowie deren Interaktion. Der GameManager ist die zentrale Klasse, die alle anderen Klassen verbindet, um Training und Evaluation der Agenten durchzuführen. Auf Basis, der in ExperimentParameters definierten Werte erstellt der GameManager zwei Agenten und weist diesen eine gemeinsame Experience Instanz zu. Die beiden Agenten werden mittels Self-play trainiert und anschließend in Evaluationsspielen getestet. Zur Bewertung der Agenten verwendet der GameManager den Minimax-Algorithmus. Währenddessen erstellt der GameManager Log-Dateien auf deren Basis die Auswertung erfolgt. Einerseits CSV-Dateien, die zum Plotten der Konvergenz und Berechnen der Spielstärke verwendet werden. Zum anderen Dateien, wie im Anhang B, die für Training und Evaluation verwendete Experimentparameter und Spielergebnisse enthalten. In den folgenden Abschnitten werden die Bestandteile genauer betrachtet.

Für die Implementierung wurde Java 16 und somit die derzeit aktuellste Java Version verwendet. Um die volle Kontrolle über die Implementierung der Agenten zu haben, wurden

keine zusätzlichen Bibliotheken eingebunden. Die einzige Ausnahme ist die Bibliothek Apache CommonsCSV, die zur Erstellung von CSV-Dateien genutzt wird. Der Source-Code und die Dokumentation der Anwendung sowie Verwendungshinweise werden bereitgestellt auf Github.com/JonasBingel

4.2. Gamefield

Die Klasse Gamefield implementiert das Tic-Tac-Toe Spielfeld, wie es in Abschnitt 3.2 beschrieben wurde. Gamefield verwaltet pro Symbol ein Bitboard. Die Bitboards B_X und B_O wurden umgesetzt mittels der Java BitSet-Klasse, die einen Vektor von Bits verwaltet. Zudem implementiert die BitSet-Klasse die klassischen Logikoperationen, sodass die beschriebene Methodik zur Erfassung der legalen Aktionen und Gewinnprüfung umgesetzt werden können.

In der Anwendung werden möglichen Aktionen und die Binärrepräsentation B_s eines Zustands jeweils in einem int zur Basis 10 gespeichert. Der primitve Datentyp int umfasst 32 Bits und ist somit ausreichend, um die 18 Bit von B_s zu verwalten. Listing 4.1 zeigt die Berechnung von B_s und somit die Implementierung der Gleichung (3.2). Bevor die zu den Bitboards zugehörigen Zahlen addiert werden, erhält B_X mittels LSHIFT den beschriebenen Offset von 9 Bit, d.h. der Spielfeldgröße, damit keine Überschneidung von B_X und B_O auftritt.

Listing 4.1.: getState-Methode zur Berechnung von B_S

```
1  public int getState() {
2                  int stateAsInt = 0;
3                  int bitBoardXASInt = Gamefield.convertBitBoardToInt(this.bitboardX);
4                  int bitBoardOASInt = Gamefield.convertBitBoardToInt(this.bitboard0);
5
6                  // bitshift the state number for X by the symbolspecific offset to avoid
                   ↪  overlap
7                  int bitBoardXWithOffset = bitBoardXASInt <<
                   ↪  getSymbolSpecificOffset(Symbol.SYMBOL_X);
8
9                  stateAsInt = bitBoardXWithOffset + bitBoardOASInt;
10                 return stateAsInt;
11         }
```

4.3. Minimax

Abb. 4.2 zeigt die wichtigsten Bestandteile der MinimaxAlgorithm-Klasse sowie deren Interaktion mit anderen Klassen. MinimaxAlgorithm implementiert den Minimax-Algorithmus aus Abschnitt 2.3 und dient der Evaluation der Agenten. Einerseits wird während des Trainings geprüft, ob die vom Agenten gewählten Aktionen optimal sind. Andererseits ist Minimax ein Gegner im Rahmen der Evaluationsspiele, um die Spielstärke der Agenten zu messen.

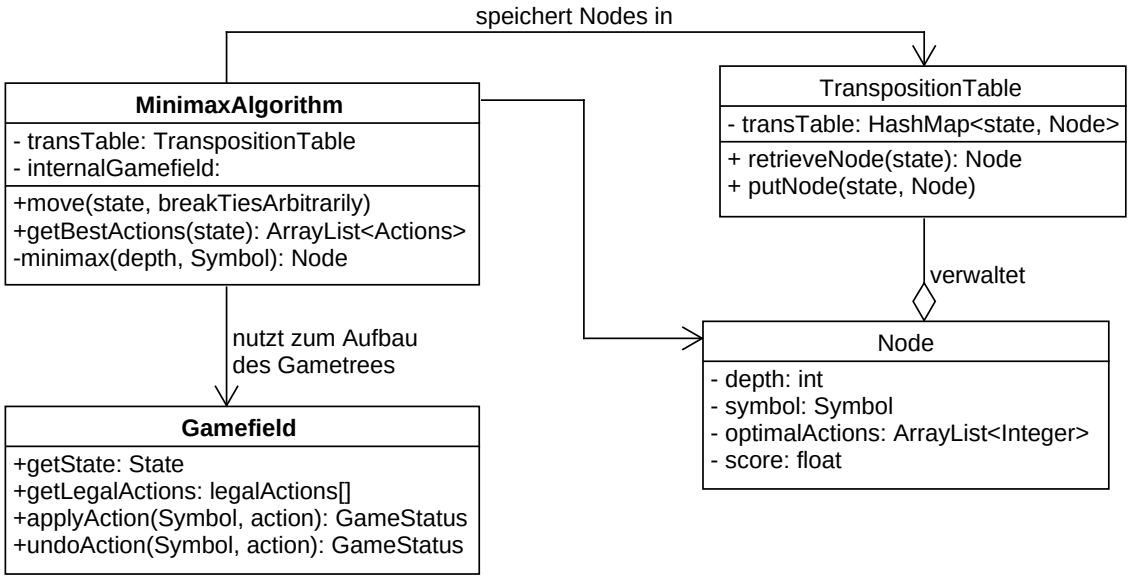


Abbildung 4.2.: MinimaxAlgorithm und verbundene Klassen

Die Klasse bietet dafür zwei Methoden. Für die Bewertung der Aktionen des Agenten gibt es die getBestActions-Methode. Diese gibt alle gleich optimalen Aktionen zu einem Zustand zurück, sodass geprüft werden kann, ob die vom Agenten gewählte Aktion ein Element dieser Liste und somit optimal ist. Die move-Methode gibt für einen Zustand eine der optimalen Aktionen zurück, sodass Minimax als Gegner genutzt werden kann. Dabei kann als zweiter Parameter übergeben werden, ob bei mehreren gleich optimalen Aktionen die Aktion mit dem niedrigsten Index oder eine zufällige ausgewählt werden soll. Da in den Evaluationsspielen möglichst viele unterschiedliche Zustände betrachtet werden sollen, wird die Option für nichtdeterministisches Spiel genutzt.

Der Spielbaum wird konstruiert durch die minimax-Methode, die in Listing Anhang C dargestellt ist. Mit jedem Aufruf wird ein weiterer Knoten (engl. Node) dem Spielbaum hinzufügt. In der minimax-Methode wird der aktuelle Zustand sowie die legalen Aktionen des intern verwalteten Spielfelds abgerufen. Eine der legalen Aktionen wird auf das Spielfeld angewendet und für den resultierenden Zustand wird erneut Minimax aufgerufen. Wird ein Terminalknoten erreicht, wird die letzte Aktion rückgängig gemacht und die nächste legale Aktion ausgeführt. Dies erfolgt rekursiv, bis alle Äste des Spielbaums konstruiert wurden. Jeder Knoten des Spielbaums entspricht einem Zustand des Spielfelds und wird repräsentiert durch eine Node-Instanz. Die Nodes werden unter der eindeutigen Nummer B_s des zugehörigen Zustands in einer Transposition table gespeichert.

Im Rahmen der Minimax-Methode werden zudem die Attribute der Node des aktuellen Zustands auf Basis der darauf folgenden Nodes aktualisiert. Dies beinhaltet die Anpassung der zu erwartenden Utility und die darauf basierende Auswahl der optimalen Aktionen, wobei Symbol X der maximierende und O der minimierende Spieler ist. Als Bewertungsfunktion

verwendet Minimax ebenfalls die in Abschnitt 3.1 vorgestellte Rewardfunktion mit Depth Penalty.

4.4. Reinforcement Learning Komponenten

In den folgenden Abschnitten werden die Komponenten beschrieben, die Reinforcement Learning implementieren. Dies umfasst die Agenten für Q-Learning und Sarsa sowie deren Lerndaten in Form der Q- und W-Tabelle.

4.4.1. Q-Learning und Sarsa Agenten

Da Q-Learning und Sarsa eng verwandte TDL Algorithmen sind, wurde eine abstrakte Klasse RLTDAgent erstellt, von der die konkreten Agenten abgeleitet werden. Neben dem Vorteil der Redundanzvermeidung nutzen AgentQLearning und AgentSARSA ein einheitliches Interface und können in den Trainings- und Evaluationsmethoden untereinander ausgetauscht werden.

Der Unterschied der beiden Algorithmen liegt in deren Vorgehen in jedem Zeitschritt, weshalb die move-Methode als abstrakt deklariert wurde. Listing 4.2 und Listing 4.3 zeigen die Implementierung der move-Methode des AgentQLearning und AgentSARSA. Wie in Abschnitt 2.2.4 erklärt, unterscheiden sie sich in zwei Aspekten. Einerseits in der Aktion, die jeweils für das Update verwendet wird. Bei Q-Learning ist dies immer Aktion mit dem höchsten Q-Value und bei Sarsa die Aktion, die der Agent wirklich durchführt. Andererseits liegt der Unterschied darin, dass Q-Learning das TD Update durchführt, bevor es die Aktion für den Zustand S' wählt. Da im Fall von TTT kein Zustandswechsel möglich ist, bei dem ein Zustand S sein eigener Folgezustand S' ist, könnte das Temporal-Difference Update auch nach der Aktionswahl durchgeführt werden. Dadurch reduziert sich der Unterschied auf die Wahl der Aktion für das Update. Auf diese Reduktion wurde verzichtet, um die Algorithmen wie vorgestellt zu implementieren und den Unterschied deutlich im Source-Code darzustellen. Beide move-Methoden implementieren die ϵ-greedy-Policy [3, S. 27f.].

4.4.2. Lerndaten

Die gesammelte Erfahrung des Agenten wird in der Anwendung durch das Experience Interface implementiert, das in Abb. 4.3 dargestellt ist. Da der Agent die Lerndaten in Form einer Q-Tabelle oder W-Tabelle organisieren kann, bietet das Experience Interface dem Agenten einheitliche Methoden zum Zugriff auf die Lerndaten. Die in Abschnitt 3.1.2 beschriebene Abbildung der SA Tupel auf den resultierenden Afterstate, wird so ebenfalls vom Agenten entkoppelt und erfolgt in der Klasse WTable.

Eine Experience Instanz wird nicht durch einen Agenten selbst erzeugt, sondern im Konstruktor übergeben. Diese Implementierung hat zwei Gründe. Zum einen ist es notwendig, damit zwei Agenten im Rahmen von Self-play gemeinsam eine Experience Instanz füllen können. Zum anderen verdeutlicht die Trennung, dass die Erfahrung kein inhärenter Bestandteil eines Agenten ist, sondern diese nur vom Agenten verwendet werden. Der Unterschied eines trainierten und

Listing 4.2.: move-Methode des Agent Q-Learning

```
public int move(int state, int[] legalActions, float reward) {
                this.qTable.initialiseQTableEntryIfNotExistent(state, legalActions);
                // The TD update is not executed if it is the first state of a new episode for
                // the agent
                if (!this.isFirstStateOfNewEpisode) {
                        this.updateQValueOfLastSATuple(state, this.qTable.getBestAction(state,
                        ↪  legalActions), reward);
                }
                int bestAction = this.qTable.getBestAction(state, legalActions);
                int chosenAction = this.pickActionUsingEpsilonGreedy(bestAction,
                ↪  legalActions);

                this.isFirstStateOfNewEpisode = false;
                this.lastState = state;
                this.lastAction = chosenAction;
                return chosenAction;
        }
```

Listing 4.3.: move-Methode des Agent Sarsa

```
public int move(int state, int[] legalActions, float reward) {
                this.qTable.initialiseQTableEntryIfNotExistent(state, legalActions);
                int bestAction = this.qTable.getBestAction(state, legalActions);
                int chosenAction = this.pickActionUsingEpsilonGreedy(bestAction,
                ↪  legalActions);

                // The TD update is not executed if it is the first state of a new episode for
                // the agent
                if (!this.isFirstStateOfNewEpisode) {
                        this.updateQValueOfLastSATuple(state, chosenAction, reward);
                }
                this.isFirstStateOfNewEpisode = false;
                this.lastState = state;
                this.lastAction = chosenAction;
                return chosenAction;
        }
```

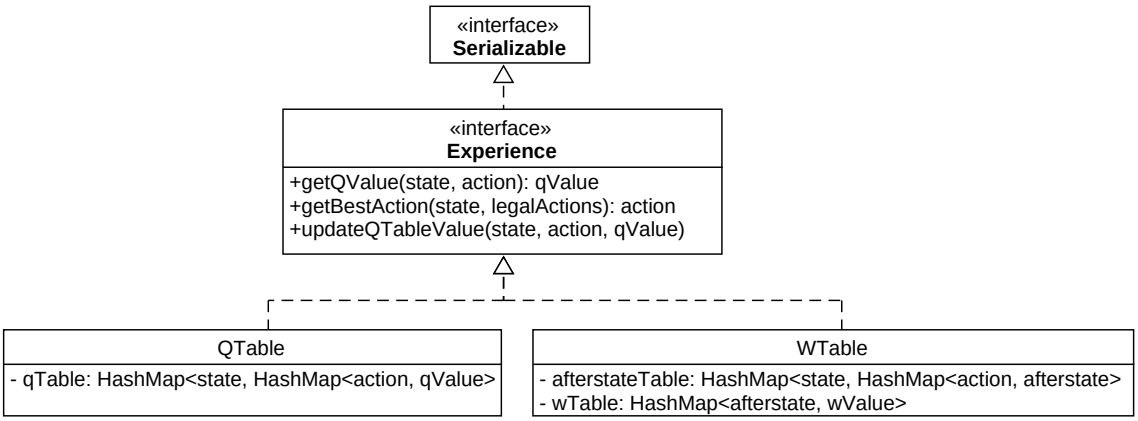


Abbildung 4.3.: Experience Interface und implementierende Klassen

untrainierten Agenten eines Algorithmus besteht, abgesehen von den Hyperparametern, in der Erfahrung, auf die jeweils zugegriffen wird. Aus diesem Grund implementiert Experience das Java Interface Serializable, wodurch Erfahrungen als Datei exportiert und importiert werden können, sodass Agenten erneut geladen werden können, um weitere Tests durchzuführen. In der Implementierung sind Q-Tabelle und der Afterstate-Tabelle beide mittels geschachtelten HashMaps umgesetzt, für die als Schlüssel jeweils die eindeutigen Zustände B_S und Aktionen verwendet werden.

Im vorgestellten Pseudocode von Q-Learning und Sarsa werden die Q-Values aller SA Tupel vor der ersten Episode willkürlich initialisiert. Dies setzt voraus, dass eine Liste aller legalen SA Tupel vorliegt. Stattdessen werden in der Implementierung die Q-Tabelle und W-Tabelle dynamisch konstruiert. Bei jedem Aufruf der move-Methode eines Agenten reicht dieser den aktuellen Zustand sowie die legalen Aktionen zur Initialisierung weiter. Dies ermöglicht die Erfassung der Zustandsanzahl, die der Agent im Rahmen des Trainings besucht hat. Im Konstruktor kann der initiale Q-Value übergeben werden. In dieser Arbeit wird ausschließlich 0 verwendet.

4.5. GameManager

Die Klasse GameManager ist zuständig für Training und Evaluation der Agenten sowie das Logging. In Abb. 4.4 ist der vereinfachte Ablauf des Trainings mit klassischem Self-play für AgentSARSA dargestellt. Für jedes Training wird ein neues Gamefield erzeugt und Experimentparameter definiert, auf deren Basis Experience und Agenten initialisiert werden. Zu Beginn jeder Episode werden die Hyperparameter der Agenten entsprechend der Experimentparameter aktualisiert. Anschließend wird ein Spiel gestartet und in jedem Zug werden der Zustand und die legalen Aktionen des Gamefields abgerufen. Beginnend mit X werden abwechselnd die move-Methoden der Agenten aufgerufen, in dessen Rahmen der direkte Reward, $r = 0$, ausgegeben wird. Die zurückgegebene Aktion wird auf das Gamefield angewandt. Nach Ende des Spiels, wird der Reward mit Depth Penalty für die letzte Aktion beider Agenten verteilt und das Spielfeld

zurückgesetzt. Das Training mit alternierendem Self-play folgt dem gleichen Ablauf. Innerhalb eines Batchs werden die Hyperparameter des nicht lernenden Agenten so gesetzt, das dieser mit der Greedy-Strategie spielt und bei Aufruf seiner move-Methode kein TD Update durchführt. Im Anschluss an das Training werden beide Agenten im Rahmen von Evaluationsspielen bewertet und die Experience Instanz wird als Datei serialisiert.

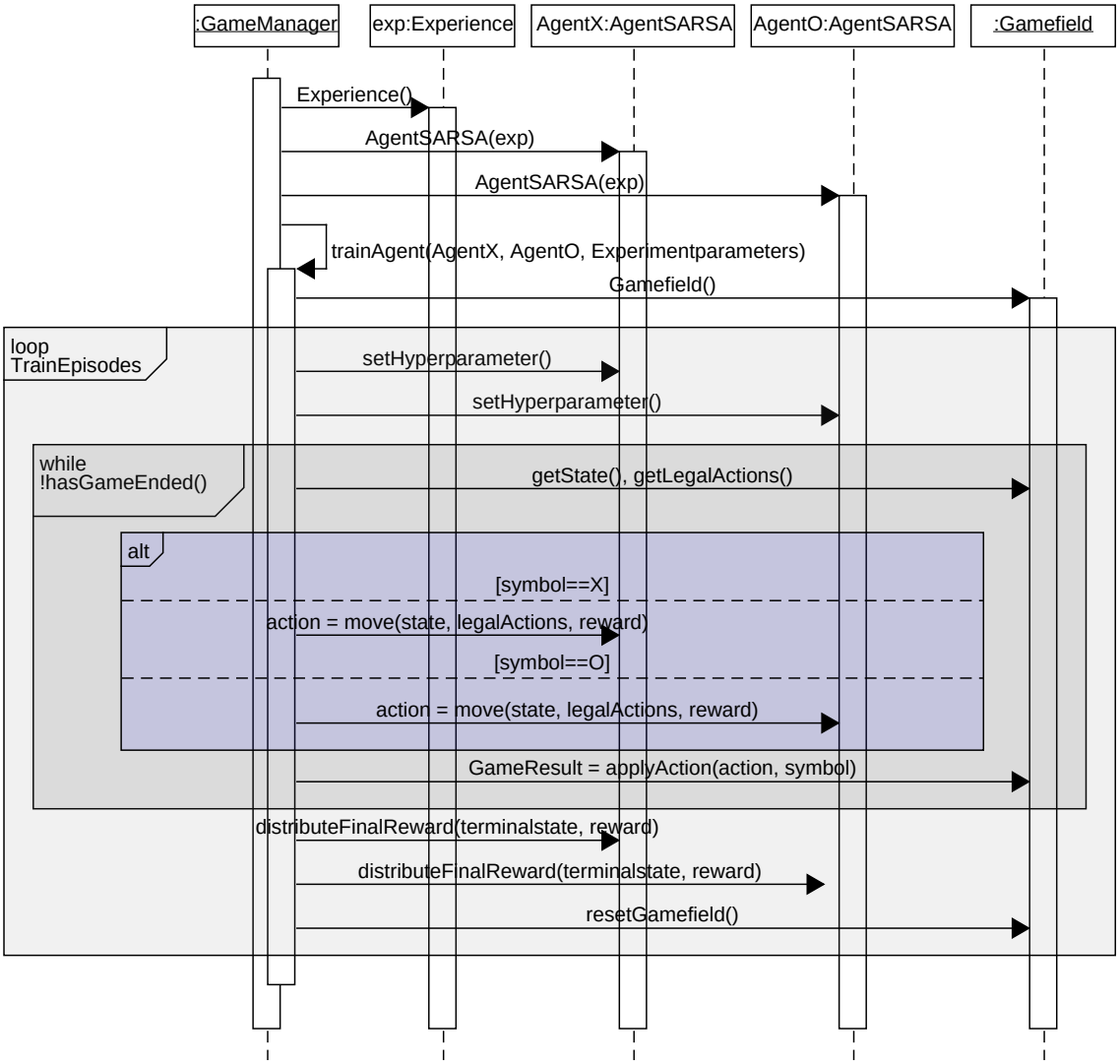


Abbildung 4.4.: Vereinfachte Darstellung des Trainingsablaufs mit klassischem Self-play

5. Ergebnisse mit Auswertung und Diskussion

In den folgenden Abschnitten werden die Ergebnisse der durchgeführten Untersuchungen vorgestellt und diskutiert. Zunächst wird ein Vergleichsmaßstab für die Spielstärke geschaffen. Anschließend werden die Ergebnisse von Q-Learning und Sarsa vorgestellt,diskutiert und verglichen. Zum Schluss werden die Forschungsfragen auf Basis der Ergebnisse beantwortet.

5.1. Vergleichsmaßstab der Spielstärke

Zur Evaluation spielen die trainierten Agenten jeweils 10.000 Spiele als beide Symbole gegen den Minimax-Algorithmus und einen Spieler mit Zufallsstrategie, Random. Damit eine Auswertung der Ergebnisse erfolgen kann, ist es zunächst notwendig einen Vergleichsmaßstab zu schaffen. Dafür spielen die Evaluationsgegner, Minimax-Algorithmus und Random, selbst gegeneinander.

Tabelle 5.1 enthält die Spielergebnisse jeder Kombination von Minimax und Random. Die dargestellten Ergebnisse sind das arithmetische Mittel von fünf Durchläufen und konsistent zu den Experimenten in [28]. Minimax spielt gegen sich selbst immer zu einem Unentschieden und gewinnt gegen Random in 97% der Spiele als X und 78% der Spiele als O. Minimax kann gegen Random nicht immer gewinnen, da die Wahrscheinlichkeit besteht, dass Random zufällig optimale Aktionen wählt und das Spiel so in einem Unentschieden endet.

Aus den Testspielen kann zudem die durchschnittliche Wahrscheinlichkeit berechnet werden, mit der eine zufällig gewählte Aktion laut Minimax optimal ist. Für das Symbol X beträgt die Wahrscheinlichkeit ungefähr 70% und für das Symbol O knapp 38%. Die Wahrscheinlichkeit ist abhängig vom Symbol, da beispielsweise laut Minimax jede erste Aktion von X optimal ist. Hingegen muss O bereits in seiner ersten Aktion einen Konter für die Aktion von X wählen. Dies erklärt zudem, wieso im Szenario Random gegen Random das Symbol X mehr als die Hälfte der Spiele gewinnt. Basierend auf diesen Spielergebnissen wurde die Spielstärke beider Evaluationsgegner berechnet. Wie in Abschnitt 3.4 beschrieben, ist das Intervall der Spielstärke $[-18.000; 9.000]$. Minimax erreicht eine Spielstärke von $6917,86 \pm 22,29$ und Random von $-6938 \pm 34,67$.

Tabelle 5.1.: Spielergebnismatrix von Minimax und Spieler mit Zufallsstrategie Random

	X Minimax		**X Random**	
O Minimax	X Minimax:	0,00% ± 0,00%	X Random:	0,00% ± 0,00%
	O Minimax:	0,00% ± 0,00%	O Minimax:	77,63% ± 4,91%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	22,37% ± 4,91%
O Random	X Minimax:	0,00% ± 0,00%	X Random:	58,38% ± 5,41%
	O Random:	0,00% ± 0,00%	O Random:	29,00% ± 1,91%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	12,62% ± 3,95%

5.2. Auswertung des Q-Learning Agenten

In den folgenden Abschnitten werden die Ergebnisse der Q-Learning Agenten vorgestellt und ausgewertet. Zunächst wird die Konvergenz während dem Training und die Spielstärke betrachtet. Anschließend wird vorgestellt, welche Auswirkung die Nutzung von Afterstates hat. Abschließend erfolgt eine Detailbetrachtung für ausgewählte Zustände.

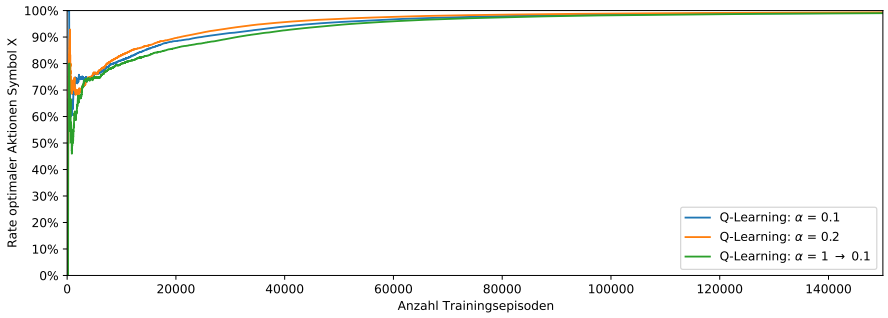
5.2.1. Konvergenz der Rate optimaler Aktionen

Abb. 5.1 zeigt den Verlauf der Rate optimaler Aktionen der Q-Learning Agenten mit den drei verschiedenen Lernraten α währen des Trainings mit klassischem Self-play. Die drei Graphen für Symbol X unterscheiden sich stärker innerhalb der ersten 5.000 Episoden. Ab ungefähr 70.000 Episoden ist die Rate stabil, wobei $\alpha = 0,2$ am schnellsten konvergiert, dicht gefolgt von $\alpha = 0,1$. Im Gegensatz dazu unterscheiden sich die Graphen für Symbol O deutlich stärker und verlaufen erst ab ungefähr 30.000 Episoden gleich. Zudem beginnt die Rate für Symbol O etwas über 0,4 und ist somit niedriger als die Rate von X. Dies kann erklärt werden durch die unterschiedliche Wahrscheinlichkeit zufällig eine optimale Aktion zu wählen, wie Abschnitt 5.1 beschrieben. Ab ungefähr 100.000 Episoden stabilisiert sich die Rate für das Symbol O auf einem niedrigeren Wert als X. Somit konvergiert Symbol O langsamer als Symbol X, wobei Symbol O für $\alpha = 0,1$ am schnellsten konvergiert. Da die Agenten durch Self-play trainiert werden und somit beide Symbole spielen, muss die Konvergenzrate auch für beide Symbole gemeinsam betrachtet werden. Daher konvergiert Q-Learning bei klassischem Self-play für $\alpha = 0,1$ am schnellsten.

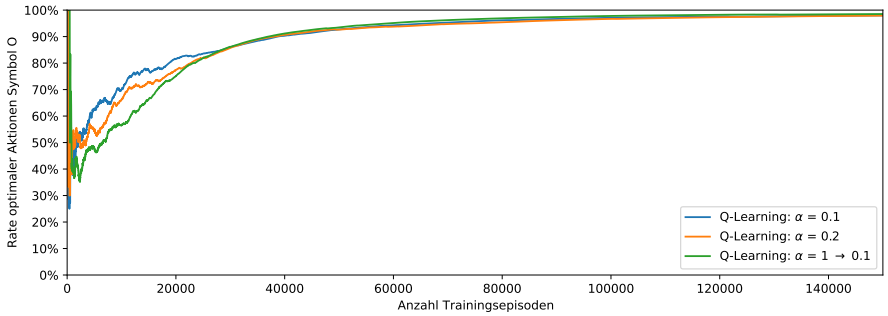
Der Vergleich der Rate optimaler Aktionen von Q-Learning für alternierendes Self-play ist im Anhang D. Für beide Symbole konvergiert alternierendes Self-play langsamer als das klassische Self-play und stabilisiert sich auf einer geringeren Rate.

5.2.2. Spielstärke

Tabelle 5.2 und Tabelle 5.3 zeigen die durchschnittliche Spielstärke der trainierten Agenten für klassisches und alternierendes Self-play. Alle Agenten, die mit klassischem Self-play trainiert wurden, erreichen eine bessere durchschnittliche Spielstärke als Minimax. Im Gegensatz dazu ist die durchschnittliche Spielstärke aller Agenten, die mit alternierendem Self-play trainiert



(a) Symbol X



(b) Symbol O

Abbildung 5.1.: Rate optimaler Aktionen von Q-Learning für verschiedene Lernraten α, klassisches Self-play (a) Symbol X (b) Symbol O

Tabelle 5.2.: Spielstärke von Q-Learning mit unterschiedlichen Lernraten, klassisches Self-play

Lernrate α	Spielstärke Q-Learning	Spielstärke Minimax	Differenz zu Minimax
Konstant 0,1	8.002,72 ± 30,03	6.917,86	1.084,86 ± 30,03
Konstant 0,2	7.801,36 ± 131,22	6.917,86	883,5 ± 131,22
Abnehmend 1 → 0,1	7.894,92 ± 132,52	6.917,86	977,06 ± 132,52

wurden schlechter als die des Minimax und deutlich schlechter im Vergleich zum klassischen Self-play.

Der Q-Learning Agent, der die Lernrate $\alpha = 0,1$ nutzt und mit klassischem Self-play trainiert wurde, konvergierte am schnellsten und erreicht mit durchschnittlich 8.002,72 die höchste Spielstärke. Tabelle 5.4 enthält die Spielergebnisse der Evaluationsspiele dieses Agenten. Der Agent spielt perfekt gegen Minimax sowohl als Symbol X als auch Symbol O. Gegen Random verliert der Agent keine Spiele und erreicht eine Gewinnrate als Symbol X von 99% und Symbol O von 91%. Die Gewinnrate ist somit besser als die von Minimax, die 97% bzw. 78% beträgt. Somit konnte Q-Learning durch Self-play lernen, TTT gegen Gegner unterschiedlicher Spielstärke erfolgreich zu spielen. Die Spielergebnismatrizen des Q-Learning Agenten für die anderen Hyperparameter und Arten vonSelf-play sind im Anhang D angegeben.

Tabelle 5.3.: Spielstärke von Q-Learning mit unterschiedlichen Lernraten, alternierendes Self-play

Lernrate α	Spielstärke Q-Learning	Spielstärke Minimax	Differenz zu Minimax
Konstant 0,1	6.465,66 ± 481,99	6.917,86	-452,2 ± 481,99
Konstant 0,2	6.904,92 ± 399,15	6.917,86	-12,94 ± 399,15
Abnehmend 1 → 0,1	6.110,56 ± 224,66	6.917,86	-807,3 ± 224,66

Tabelle 5.4.: Spielergebnismatrix für Q-Learning mit Lernrate $\alpha = 0,1$, klassisches Self-play

	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	99,22% ± 0,37%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	0,78% ± 0,37%
O Q-Learning	X Minimax:	0,00% ± 0,00%	X Random:	0,00% ± 0,00%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	91,45% ± 0,21%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	8,55% ± 0,21%

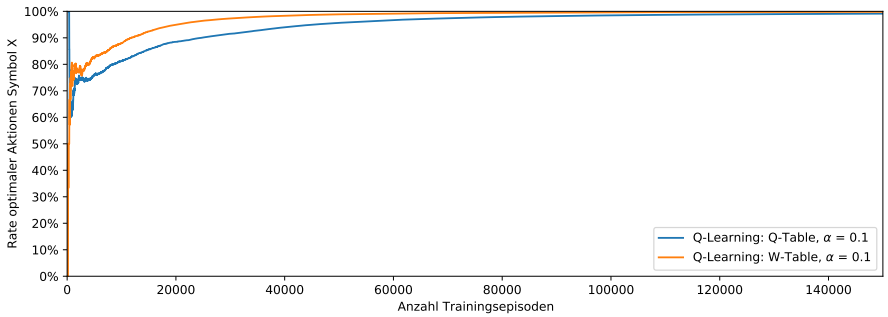
5.2.3. W-Tabelle

Zur Evaluation der Auswirkung von Afterstates auf die Konvergenz und Spielstärke wird der stärkste ermittelte Q-Learning Agent mit einem Agenten verglichen, der eine W-Tabelle und die gleichen Hyperparameter nutzt. Wie im vorigen Abschnitt beschrieben, ist dies ein Q-Learning Agent mit Lernrate $\alpha = 0,1$, der mit klassischen Self-play trainiert wird.

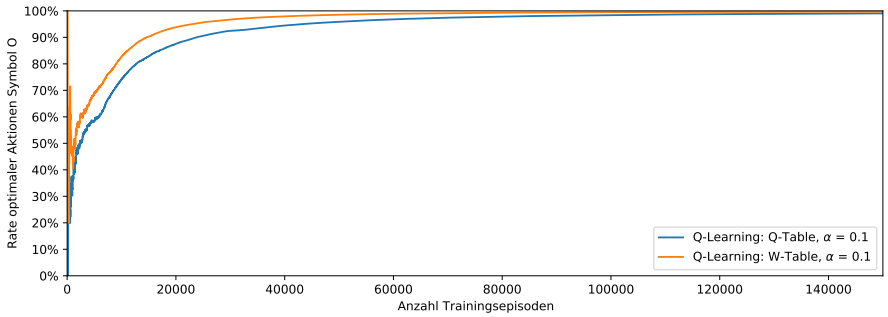
Abb. 5.2 vergleicht die Rate optimaler Aktionen von Agenten mit Q-Tabelle und mit W-Tabelle. Sowohl für Symbol X als auch Symbol O konvergiert der Agent mit W-Tabelle deutlich schneller und stabilisiert sich auf einer höheren Rate optimaler Aktionen. Dies äußert sich in der Spielstärke des Agenten, wie Tabelle 5.5 zeigt.

Durch die Nutzung des W-Tabelle im Vergleich zur Q-Tabelle steigert sich die durchschnittliche Gewinnrate gegen Random als Symbol X von 99,22% auf 99,46% und als Symbol O von 91,45% auf 91,51%. Die absolute Spielstärke steigt von 8.002,72 auf 8.036,68. Dies könnte wie in Abschnitt 3.1.2 und [3, S. 136f.] beschrieben, auf die effektivere Nutzung der gesammelten Erfahrung zurückgeführt werden. Da die Spielstärke der Spielstärke für den Q-Tabelle jedoch $\pm 75,69$ beträgt und die Stichprobenanzahl mit $N = 5$ klein ist, kann dies nicht sicher gesagt werden. Es ist daher auch möglich, dass die Verbesserung durch Zufall entstanden somit statistisch nicht signifikant ist.

Die ermittelten Spielergebnisse des Q-Learning Agenten sind konsistent mit anderen Experimenten in [28]. In diesen wurde jedoch ein Double Q-Learning Agent mit Q-Tabelle und abnehmenden Hyperparametern durch klassisches Self-play für 3.000.000 Episoden trainiert.



(a) Symbol X



(b) Symbol O

Abbildung 5.2.: Rate optimaler Aktionen von Q-Learning Lernrate $\alpha = 0,1$, W-Tabelle, klassisches Self-play (a) Symbol X (b) Symbol O

Tabelle 5.5.: Spielergebnismatrix für Q-Learning mit Lernrate $\alpha = 0,1$ und W-Tabelle, klassisches Self-play

	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	99,46% ± 0,05%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	0,54% ± 0,05%
O Q-Learning	X Minimax:	0,00% ± 0,00%	X Random:	0,00% ± 0,00%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	91,51% ± 0,20%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	8,49% ± 0,20%

5.2.4. Detailbetrachtung

Die Spielstärke des Q-Learning Agenten wirft die Frage auf, wie der Agent besser als ein Minimax-Algorithmus spielen kann. Wie in Abschnitt 2.3 erwähnt, erwartet der Minimax-Algorithmus einen optimalen Gegner und spielt nur gegen diesen optimal. Hingegen hat Q-Learning auch gelernt gegen nicht optimale Gegner optimal zu spielen. Dies wird deutlich, wenn die Einträge der Q-Tabelle bzw. W-Tabelle für einen Zustand betrachtet werden. Abb. 5.3 zeigt das Spielfeld des Zustands 6754 und Tabelle 5.6 die Bewertung der legalen Aktionen durch den Minimax und Q-Learning Agent mit W-Tabelle. In diesem Zustand ist Symbol X am Zug und die laut Expert Play optimalen Aktionen sind 4 und 8, um den Gegner zum Blocken zu zwingen. Wählt Symbol O im nächsten Zug die nicht optimale Aktion 7, gewinnt X. Die Aktion 7 garantiert hingegen, dass das Spiel unentschieden endet. Minimax weist allen Aktionen den Wert 0 zu, da dieser von einem optimalen Gegner und somit einem Unentschieden ausgeht. Der Q-Learning Agent weist den Aktionen 4 und 8 einen positiven Wert zu, da im Rahmen des Trainings auch gegen nicht optimale Gegner gespielt wurde und so Erfahrungen vorliegen, in denen der Agent das Spiel aus diesem Zustand gewonnen hat.

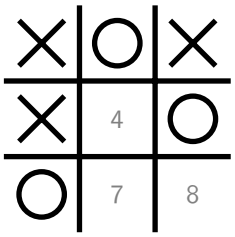


Abbildung 5.3.: Spielfeldkonstellation Zustand 6754

Aktion	Minimax	Q-Learning
4	0	0,0015
7	0	0,0000
8	0	0,0006

Tabelle 5.6.: Bewertung Aktionen in Zustand 6754

Eine Forschungsfrage ist die Ermittlung der besten ersten Aktion für das Symbol X. Zur Beantwortung soll die Bewertung des besten Q-Learning Agenten für alle legalen Aktionen im Zustand 0 betrachtet werden. Tabelle 5.7 listet die durchschnittliche Bewertung der Aktionen von fünf Q-Learning Agenten mit W-Tabelle. Die Aktion 4 (Mitte) hat mit 0,0053 den höchsten Q-Value. Jedoch zeigt die Auflistung der Q-Values ein weiteres Problem. Die Tabelle enthält jeweils vier Bewertungen für Ecken und Kanten, die alle den gleichen Wert haben sollten. Die Aktionen sollten den gleichen Wert haben, da sie in Zuständen resultieren, die nach Rotation oder Spiegelung äquivalent sind. Durch Berechnung der durchschnittlichen Bewertung je äquivalenter Aktion, wie in Tabelle 5.8, kann die durchschnittliche Rangfolge der Aktionen angegeben werden. Demnach ist die Aktion Mitte weiterhin die beste Aktion, gefolgt von der Aktion Ecke. Für eine abschließende Beantwortung der Frage wäre eine Enkodierung der Zustände notwendig, die äquivalenten Zuständen einen gemeinsamen Bezeichner und Q-Value zuweist. Das Ergebnis, dass die beste erste Aktion die Mitte ist, ist jedoch konsistent zu vorher durchgeführten Experimenten von [25].

Tabelle 5.7.: Bewertung Zustand 0 bester Q-Learning Agent

Aktion	Bewertung
0	0,0041
1	0,0041
2	0,0045
3	0,0033
4	0,0053
5	0,0030
6	0,0035
7	0,0033
8	0,0044

Tabelle 5.8.: Bewertung Aktionsklassen Zustand 0 bester Q-Learning Agent

Aktion	Bewertung
Ecke	0,0041
Kante	0,0034
Mitte	0,0053

5.3. Auswertung des Sarsa Agenten

In den folgenden Abschnitten werden die Ergebnisse der Sarsa Agenten vorgestellt und ausgewertet. Zunächst wird die Konvergenz während dem Training und die Spielstärke betrachtet. Anschließend wird vorgestellt, welche Auswirkung die Nutzung von Afterstates hat. Abschließend erfolgt eine Detailbetrachtung für ausgewählte Zustände.

5.3.1. Konvergenz der Rate optimaler Aktionen

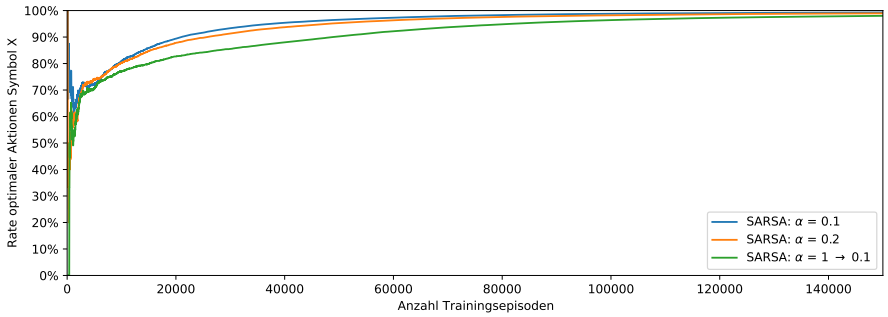
Abb. 5.4 zeigt den Verlauf der Rate optimaler Aktionen von Sarsa Agenten mit den drei verschiedenen Lernraten α während des Trainings mit klassischem Self-play. Der Agent mit Lernrate $\alpha = 0,1$ konvergiert am schnellsten, dicht gefolgt von $\alpha = 0,2$. Für das Symbol X stabilisiert sich die Rate ab ungefähr 90.000 und für O ab 110.000 Trainingsepisoden.

Die Agenten von Symbol X stabilisieren sich auf einer höheren Rate optimaler Aktionen als die von Symbol O, wobei in beiden Fällen der Agent mit $\alpha = 0,1$ die höchste Rate erreicht. Hingegen konvergiert bei beiden Symbolen der Agent mit abnehmender Lernrate langsamer und erreicht, insbesondere bei Symbol O, eine niedrigere Rate optimaler Aktionen.

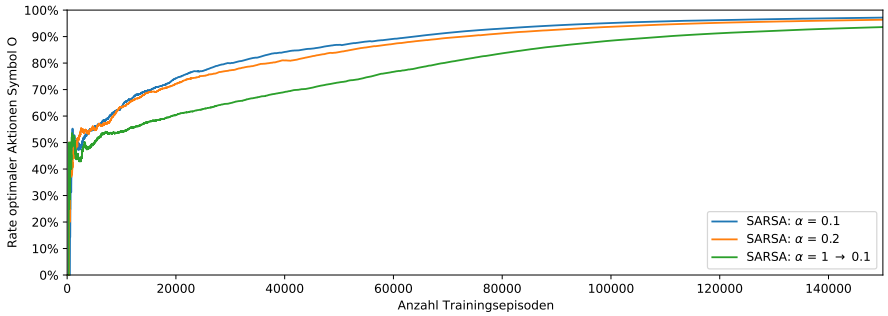
Der Vergleich der Rate optimaler Aktionen von Sarsa für alternierendes Self-play ist im Anhang E. Für beide Symbole konvergiert das alternierende Self-play langsamer als das klassische Self-play und stabilisiert sich auf einer geringeren Rate.

5.3.2. Spielstärke

Tabelle 5.9 und Tabelle 5.10 zeigen die durchschnittliche Spielstärke der trainierten Agenten für klassisches und alternierendes Self-play. Für beide Arten von Self-play erreichen die Agenten eine durchschnittliche Spielstärke, die über der des Minimax liegt. Die einzige Ausnahme sind Agenten, die mit konstanter Lernrate $\alpha = 0,1$ durch alternierendes Self-play trainiert wurden und eine durchschnittliche Spielstärke von 6.496,50 haben. Training durch klassisches Self-play resultiert für Sarsa in stärkeren Agenten als Training durch alternierendes Self-play. Der Sarsa



(a) Symbol X



(b) Symbol O

Abbildung 5.4.: Rate optimaler Aktionen von Sarsa für verschiedene Lernraten α, klassisches Self-play (a) Symbol X (b) Symbol O

Tabelle 5.9.: Spielstärke von Sarsa mit unterschiedlichen Lernraten, klassisches Self-play

Lernrate α	Spielstärke Sarsa	Spielstärke Minimax	Differenz zu Minimax
Konstant 0,1	7.905,72 ± 75,69	6.917,86	987,86 ± 75,69
Konstant 0,2	7.787,94 ± 105,20	6.917,86	870,08 ± 105,20
Abnehmend 1 → 0,1	7.754,66 ± 197,59	6.917,86	836,80 ± 197,59

Agent, der die Lernrate $\alpha = 0,1$ nutzt und mit klassischem Self-play trainiert wurde erreicht mit 7.905,72 die höchste Spielstärke.

Tabelle 5.11 enthält die Spielergebnisse der Evaluationsspiele des stärksten Sarsa Agenten. Als Symbol X spielt der Sarsa Agent optimal gegen Minimax. Zudem gewinnt der Agent 99% der Spiele gegen Random und somit mehr als der Minimax, der nur 97% gewinnt. Als Symbol O gewinnt der Agent mit 89% mehr Spiele gegen Random als Minimax, der nur eine Gewinnrate von 78% erreicht. Jedoch verliert der Agent als Symbol O 0,74% der Evaluationsspiele gegen Minimax und 0,11% gegen Random. Somit erreicht der Agent eine höhere Spielstärke als Minimax, spielt jedoch nicht optimal. Die Spielergebnismatrizen der Sarsa Agenten für die anderen Hyperparameter und Arten von Self-play sind im Anhang E angegeben.

Tabelle 5.10.: Spielstärke von Sarsa mit unterschiedlichen Lernraten, alternierendes Self-play

Lernrate α	Spielstärke Sarsa	Spielstärke Minimax	Differenz zu Minimax
Konstant 0,1	6.495,50 ± 826,74	6.917,86	-422,36 ± 826,74
Konstant 0,2	7.018,94 ± 299,04	6.917,86	101,08 ± 299,04
Abnehmend 1 → 0,1	7.625,94 ± 65,10	6.917,86	708,08 ± 65,10

Tabelle 5.11.: Spielergebnismatrix für Sarsa mit Lernrate $\alpha = 0,1$, klassisches Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	99,01% ± 0,04%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	0,99% ± 0,04%
O Sarsa	X Minimax:	0,74% ± 1,48%	X Random:	0,11% ± 0,22%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	89,69% ± 0,38%
	Unentschieden:	99,26% ± 1,48%	Unentschieden:	10,205% ± 0,30%

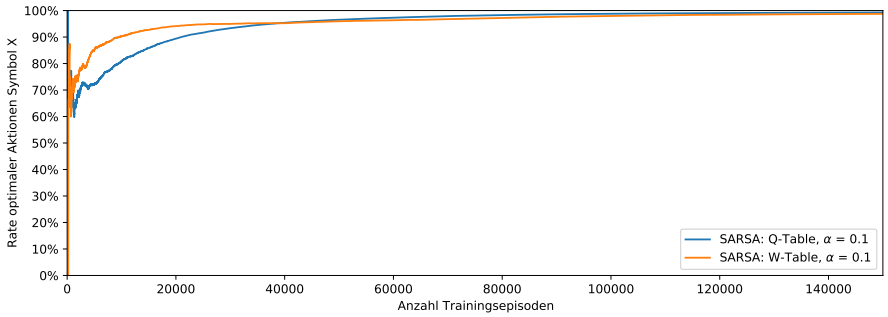
5.3.3. W-Tabelle

Abb. 5.5 vergleicht die Rate optimaler Aktionen des besten Sarsa Agenten mit Q-Tabelle und W-Tabelle. Sowohl für Symbol X als auch für Symbol O konvergiert der Agent mit W-Tabelle schneller und erreicht eine höhere Rate.

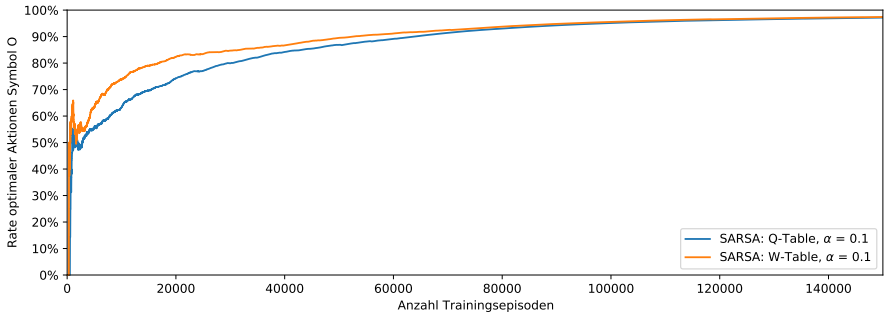
Tabelle 5.12 zeigt die Spielergebnismatrix des Sarsa Agenten mit W-Tabelle. Im Gegensatz zum Agenten mit Q-Tabelle verliert der Agent keine Spiele mehr und spielt optimal gegen Minimax. Die Gewinnrate gegen Random bleibt für das Symbol X unverändert bei 99%. Für das Symbol O sinkt die Gewinnrate von 89,69% auf 88,61%. Die durchschnittliche Spielstärke des Agenten steigt durch die Nutzung des W-Tabelle von 7.905,72 auf 7.927,68. Da die Standardabweichung für die Spielstärke des Agenten mit Q-Tabelle jedoch $\pm 30,03$ beträgt und die Stichprobenanzahl mit $N = 5$ klein ist, kann dies nicht sicher gesagt werden. Es ist daher auch möglich, dass die Verbesserung durch Zufall entstanden somit statistisch nicht signifikant ist.

Tabelle 5.12.: Spielergebnismatrix für Sarsa mit Lernrate $\alpha = 0,1$ und W-Tabelle, klassisches Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	99,01% ± 0,83%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	0,98% ± 0,83%
O Sarsa	X Minimax:	0,00% ± 0,00%	X Random:	0,00% ± 0,00%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	88,61% ± 1,28%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	11,39% ± 1,28%



(a) Symbol X



(b) Symbol O

Abbildung 5.5.: Rate optimaler Aktionen von Sarsa Lernrate $\alpha = 0,1$, W-Tabelle, klassisches Self-play (a) Symbol X (b) Symbol O

5.3.4. Detailbetrachtung

Obwohl die Spielstärke der trainierten Sarsa Agenten über dem Minimax liegt, verliert der beste Agent mit Q-Tabelle als Symbol O. Ein Zustand, der in den Evaluationsspielen mehrfach für eine Niederlage als Symbol O sorgte ist 69648, der in Abb. 5.6 dargestellt ist. Die optimalen Aktionen laut Minimax und Expert Play sind 0, 6 und 8, um den Gegner zum Blocken zu zwingen. Der Agent mit Q-Tabelle wählt jedoch gemäß Greedy-Strategie die nicht optimale Aktion 5, da diese nach Abschluss des Trainings den höchsten Q-Value hat. Die Q-Tabelle zeigt jedoch, dass die optimalen Aktionen im Vergleich zu den anderen Aktionen die höheren Q-Values haben. Lediglich der Q-Value für die Aktion 5 ist leicht höher. Der Agent benötigt demnach noch mehr Trainingsepisoden, um den Aktionen die richtige Bewertung zuzuweisen. Dies bestätigt sich, wenn der Sarsa Agent mit der W-Tabelle trainiert wird, um die gesammelte Erfahrung effektiver zu nutzen. Durch Nutzung der W-Tabelle ändert sich die Bewertung der Aktionen und die optimalen Aktionen 0, 6 und 8 haben nun die höchsten Q-Values.

Abbildung 5.6.: Spielfeldkonstellation Zustand 69648

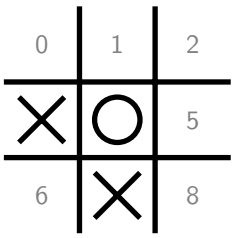


Tabelle 5.13.: Bewertung Aktionen in Zustand 69648

Aktion	Q-Tabelle	W-Tabelle
0	-0,0008	0,0016
1	-0,0851	-0,1470
2	-0,0300	-0,1466
5	0,0013	-0,1658
6	0,0002	0,0069
8	-0,0038	0,0016

Zur Prüfung welche Aktion laut den trainierten Sarsa Agenten die beste erste Aktion für Symbol X ist, wird die durchschnittliche Bewertung von fünf Agenten betrachtet, die in Abschnitt 5.3.4 gelistet sind. Alle Aktionen, mit Ausnahme der Aktion 4, haben ein negatives Vorzeichen. Wie die obige Auswertung der Spielstärke der W-Tabelle Agenten zeigt, behindert dies jedoch nicht die Spielstärke des Agenten, da bei der Aktionsauswahl lediglich die Aktion mit dem höchsten Q-Value gewählt wird. In Tabelle 5.15 wird die durchschnittliche Bewertung je äquivalenter Aktion betrachtet. Laut den trainierten Sarsa Agenten ist die Rangfolge der Aktionen: Mitte, Ecke und Kante.

Tabelle 5.14.: Bewertung Zustand 0 bester Sarsa Agent

Aktion	Bewertung
0	-0,0283
1	-0,0258
2	-0,0223
3	-0,0272
4	0,0019
5	-0,0218
6	-0,0337
7	-0,0394
8	-0,0274

Tabelle 5.15.: Bewertung Aktionsklassen Zustand 0 bester Sarsa Agent

Aktion	Bewertung
Ecke	-0,0279
Kante	-0,0286
Mitte	0,0019

5.4. Vergleich der Q-Learning und Sarsa Agenten

Die trainierten Agenten beider Algorithmen konnten erfolgreich Tic-Tac-Toe durch Self-play lernen und Spielstärken erreichen, die über dem implementierten Minimax-Algorithmus liegen. Tabelle 5.16 vergleicht die Spielstärke von Q-Learning und Sarsa, die mit klassischem Self-play trainiert wurden. Für jede untersuchte Lernrate war die Spielstärke der Q-Learning Agenten größer als die von Sarsa. Tabelle 5.17 vergleicht die Spielstärke, die bei alternierendem Self-play erreicht werden kann. Im Gegensatz zum klassischen Self-play erreichen die Sarsa Agenten eine höhere Spielstärke. Die Q-Learning Agenten verschlechtern sich deutlich bei alternierendem Self-play und sind schlechter als der Minimax-Algorithmus. Für beide Algorithmen werden

Tabelle 5.16.: Spielstärke beider Algorithmen für unterschiedliche Lernraten nach Training mit klassischem Self-play

Lernrate α	Spielstärke Q-Learning	Spielstärke Sarsa	Differenz QL - Sarsa
Konstant 0,1	8.002,72 ± 30,03	7.905,72 ± 75,69	97,00
Konstant 0,2	7.801,36 ± 131,22	7.787,94 ± 105,20	13,42
Abnehmend 1 → 0,1	7.894,92 ± 132,52	7.754,66 ± 197,59	140,26

Tabelle 5.17.: Spielstärke beider Algorithmen für unterschiedliche Lernraten nach Training mit alternierendem Self-play

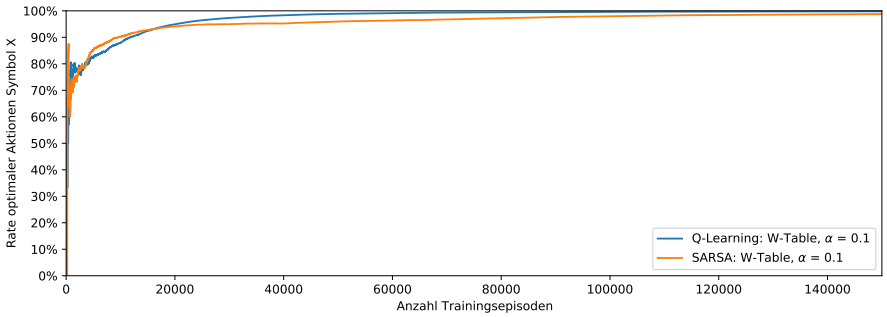
Lernrate α	Spielstärke Q-Learning	Spielstärke Sarsa	Differenz QL - Sarsa
Konstant 0,1	6.465,66 ± 481,99	6.495,50 ± 826,74	-29,84
Konstant 0,2	6.904,92 ± 399,15	7.018,94 ± 299,04	-114,02
Abnehmend 1 → 0,1	6.110,56 ± 224,66	7.625,94 ± 65,10	-1.515,38

mit dem klassischen Self-play bessere Ergebnisse erzielt. Der jeweils stärkste Agent konnte für $\alpha = 0,1$ mit klassischem Self-play erzeugt werden. Für Q-Learning beträgt die durchschnittliche Spielstärke 8.002,72 und für Sarsa 7.905,72. Für 150.000 Trainingsepisoden weist Sarsa noch eine leichte Schwäche vor und verliert als Symbol O knapp 1% seiner Spiele.

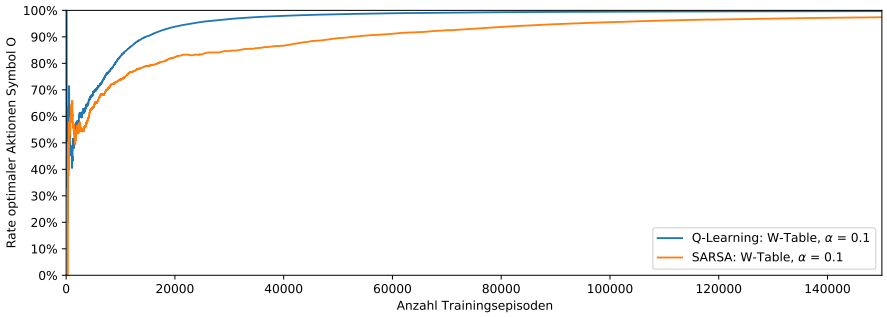
Die Verwendung von Afterstates in Form der W-Tabelle verbesserte bei beiden Agenten die Konvergenz. Abb. 5.7 vergleicht den Verlauf der Rate optimaler Aktionen während dem Training. Für Symbol X und Symbol O konvergiert der Q-Learning Agent schneller und erreicht eine höhere Rate. Zudem verbesserte sich die durchschnittliche Spielstärke beider Agenten, wie Tabelle 5.18 zeigt. Der Q-Learning Agent verbesserte seine Spielstärke um 33,96 auf 8.036,68. Die Schwäche von Sarsa wird behoben, sodass dieser als Symbol O keine Spiele mehr verliert. Die Spielstärke verbessert sich dadurch um 21,96 auf 7.927,68. Wie jedoch in den jeweiligen Abschnitten angemerkt, ist diese Verbesserung in Relation zur Standardabweichung klein und es liegt nur eine Stichprobenanzahl von $N = 5$ vor.

Tabelle 5.18.: Spielstärke beider Algorithmen mit Q-Tabelle und W-Tabelle, klassisches Self-play

Algorithmus	Spielstärke Q-Tabelle	Spielstärke W-Tabelle	Q-Tabelle - W-Tabelle
Q-Learning $\alpha = 0,1$	8.002,72 ± 75,69	8.036,68 ± 9,13	33,96
Sarsa $\alpha = 0,1$	7.905,72 ± 30,03	7.927,68 ± 43,79	21,96



(a) Symbol X



(b) Symbol O

Abbildung 5.7.: Rate optimaler Aktionen des besten Q-Learning und Sarsa Agenten während dem Training mit W-Tabelle und klassischem Self-play (a) Symbol X (b) Symbol O

5.5. Beantwortung der Forschungsfragen

Mit den vorgestellten Ergebnissen können die Forschungsfragen aus Abschnitt 1.2 wie folgt beantwortet werden:

Frage: Können die Algorithmen Expert Play in Tic-Tac-Toe durch Self-play erlernen?
Antwort: Die mit den Algorithmen trainierten Agenten spielen besser als der implementierte Minimax-Algorithmus. In den betrachteten Zuständen, zeigten die Agenten Entscheidungen, die mit dem Expert Play konsistent sind. Zur abschließenden Beantwortung ist eine Implementierung der Regeln des Expert Play notwendig, um die Aktionswahl der Agenten zu evaluieren. Die Art des Self-play ist entscheidend auf den Trainingserfolg, da beide Agenten mit alternierendem Self-play schlechtere Ergebnisse erzielten. Dabei litt Q-Learning unter Self-play deutlich mehr als Sarsa. Die erreichte Spielstärke des Q-Learning Agenten ist konsistent mit Experimenten in [28].

Frage: Eine Abschätzung für die Anzahl der Trainingsepisoden, die ungefähr benötigt werden, um Expert Play in TTT durch Self-play zu erlernen?
Antwort: In den Vorexperimenten wurde festgestellt, dass mehr als 100.000 Trainingsepisoden notwendig sind, um stabile Ergebnisse zu erhalten. Für die Experimente wurden 150.000 Episoden verwendet, in denen die Agenten Expert Play erlernt haben. Die Auswertung der Konvergenz zeigt zudem, dass Symbol X deutlich schneller konvergiert als Symbol O.

Frage: Welche Auswirkung hat die Verwendung des Konzepts der Afterstates auf das Konvergenzverhalten und Spielstärke der Agenten?
Antwort: Durch die Verwendung von Afterstates in Form einer W-Tabelle konnte die Konvergenz und Spielstärke verbessert werden. Die Verbesserungen sind jedoch in Relation zur Standardabweichung gering. Da nur eine Stichprobe von $N = 5$ vorliegt, kann nicht abschließend gesagt werden, ob die leichte Verbesserung auf die Nutzung von Afterstates oder Zufall zurückzuführen ist.

Frage: Was sind die besten Hyperparameter für Q-Learning und Sarsa?
Antwort: Aus der Menge der untersuchten Hyperparameter, erreichten Q-Learning und Sarsa die besten Ergebnisse, durch folgende Kombination von Hyperparametern:

- konstante Lernrate $\alpha = 0,1$,
- abnehmende Explorationswahrscheinlichkeit $\epsilon = 1 \rightarrow 0,1$ die nach 2/3 der Trainingsepisoden 0,1 erreicht
- Verwendung von Afterstates in Form einer W-Tabelle

Frage: Wie unterscheiden sich Q-Learning und Sarsa hinsichtlich ihrer Konvergenz?
Antwort: Q-Learning konvergiert schneller als Sarsa und erreicht eine höhere Rate optimaler Aktionen.

Frage: Welcher Agent erreicht eine bessere Spielstärke, wenn beide RL Agenten mit ihren optimalen Hyperparametern trainiert werden?
Antwort: Q-Learning erreicht eine bessere Spielstärke. Wird die Rewardfunktion aus Abschnitt 3.1 verwendet, lässt sich die Spielstärke quantifizieren. Für Q-Learning beträgt diese im Durchschnitt 8036,68 und für Sarsa 7927,68. Dabei scheint Q-Learning nahezu die maximal

mögliche optimale Spielstärke zu erreichen, die auch in mindestens einem anderen Experiment [28] gezeigt wurde.

Frage: Was ist aus Sicht der Agenten die optimale erste Aktion?
Antwort: Q-Learning und Sarsa bewerten beide die Aktion „Mitte“ als stärkste erste Aktion für X. Die zweitstärkste Aktion ist das Platzieren eines Symbols in einer der Ecken. Jedoch werden äquivalente Aktionen nicht gemeinsam bewertet, sodass die Frage nicht abschließend beantwortet werden kann. Dass die Mitte, die beste Aktion sein soll, ist jedoch konsistent mit Experimenten, die vorher durchgeführt wurden [25].

6. Konklusion

In den folgenden Abschnitten werden der Inhalt der Bachelorarbeit und die Ergebnisse der Implementierung für TTT zusammengefasst. Anschließend werden diese kritisch betrachtet bevor ein Ausblick für mögliche weitere Arbeiten gegeben wird.

6.1. Zusammenfassung

In dieser Bachelorarbeit wurden Reinforcement Learning und die zugehörigen Methoden zur Lösung sequenzieller Entscheidungsprobleme erklärt. Der Fokus lag auf dem Teilgebiet des Temporal-Difference Learning und den zugehörigen Algorithmen Q-Learning und Sarsa. Die Funktionsweise der Algorithmen wurde erklärt und deren Gemeinsamkeiten sowie Unterschiede aufgezeigt. Zur weiteren Untersuchung der Algorithmen und deren Hyperparametern wurden beide für das simple Strategiespiel Tic-Tac-Toe in Java implementiert. Untersucht wurden verschiedene Ausprägungen der Lernrate α, die im Rahmen von Vorexperimenten festgelegt wurden. Zudem wurden die Agenten mit zwei verschiedenen Arten von Self-play trainiert. Zusätzlich wurde das Konzept der Afterstates in Form einer W-Tabelle implementiert. Eine mögliche Definition einer Rewardfunktion wurde erläutert und zum Training der Agenten sowie der Quantifizierung der Spielstärke verwendet. Zur Auswertung der trainierten Agenten wurde deren Konvergenz während des Trainings sowie deren Spielstärke gegen einen Minimax-Algorithmus und Spieler mit Zufallstrategie erfasst und verglichen. Beide Algorithmen erreichten mit der gleichen Hyperparameterkombination und Nutzung von Afterstates die jeweils besten Agenten, die eine höhere Spielstärke besitzen als der Minimax-Algorithmus. Die Auswertung ergab, dass Q-Learning im Durchschnitt schneller konvergiert und eine höhere Spielstärke erreicht als Sarsa. Verwendet Sarsa keine Afterstates konnten leichte Schwächen für das Symbol O festgestellt werden. Beide Agenten bewerteten die „Mitte“ als beste erste Aktion, was konsistent zu anderen Experimenten ist.

6.2. Kritische Betrachtung der Inhalte

In der Arbeit wird ein klassischer Minimax-Algorithmus verwendet. Zum einen dient die berechnete Spielstärke des Minimax-Algorithmus als Vergleichsmaßstab bei der Bewertung der Agenten. Zum anderen basiert die Konvergenzmetrik auf den Aktionen, die laut des Minimax-Algorithmus optimal sind. Wie in Abschnitt 2.4.2 erwähnt, wählt der Agent nicht immer die optimale Aktion, da der Minimax-Algorithmus von einem optimalen Gegner ausgeht. Dadurch ist dessen Gewinnrate kleiner, als die optimal mögliche, weil nicht zwischen Aktionen unterschieden wird, die mehr Gewinnchancen haben als andere, wenn das Ergebnis gegen einen optimalen Gegner gleich ist. Für die Bewertung der Spielstärke und richtige Rate optimaler Aktionen gemäß Self-play sollte daher ein angepasster Minimax-Algorithmus genutzt werden.

Ein weiterer Aspekt ist die Verbesserung des jeweils optimalen Agenten durch das Konzept der Afterstates in Form der W-Tabelle. Relativ zur Spielstärke ist die erzielte Verbesserung klein, sodass nicht gesagt werden kann, ob diese Verbesserung nur durch Zufall entstanden ist oder wirklich auf den W-Tabelle zurückgeführt werden kann. Da wegen der Zeitbeschränkung nur fünf Agenten trainiert wurden, ist die Stichprobe zu gering für einen Signifikanztest. Um den Wert zu stabilisieren, sollten daher mehr Agenten trainiert werden.

Eine weitere Limitierung besteht in der Menge der getesteten Hyperparameterkombinationen und Methoden, wie diese während des Trainings reduziert werden können. Durch Vorexperimente wurde versucht diese möglichst sinnvoll zu wählen, aber es kann nicht ausgeschlossen werden, dass mit anderen Werten stärkere Ergebnisse möglich sind.

6.3. Anmerkungen für künftige Arbeiten

Wie im obigen Abschnitt angemerkt, liegt eine mögliche Verbesserung in der Evaluation durch den Minimax-Algorithmus. Um dies zu erreichen könnte ein Spieler implementiert werden, der die im Expert Play formulierten Regeln genau befolgt. Eine weitere Möglichkeit ist die Erweiterung des bestehenden Minimax-Algorithmus, sodass zur Bewertung der Aktionen neben dem zu erwartenden Reward auch die Anzahl potenzieller Gewinnmöglichkeiten genutzt wird. Haben zwei Aktionen einen gleich hohen Reward, wird die Aktion als optimal deklariert, die mehr Gewinnmöglichkeiten für den Agenten bietet. Neben der Verbesserung der Evaluation, könnte es lohnend sein die implementierten Standardversionen der Algorithmen mit abgewandelten Formen wie Double Q-Learning, Expected Sarsa [3, S. 133ff.], [37] oder neuronalen Netzen [38] zu vergleichen . Zudem wäre ein Vergleich mit Algorithmen interessant, die im Gegensatz zu den implementierten Algorithmen für Multi-Agenten RL vorgesehen sind. Insbesondere sogenannte Equilibrium Learners wären gut geeignet, da diese sich auf Nullsummenspiele wie TTT fokussieren. [39, S. 39] Außerdem gibt es neben dem Konzept der Afterstates weitere Möglichkeiten, um die Lerndaten effektiver zu nutzen wie z.B. Eligibility Traces [40] oder die generelle Reduktion des Zustandsraums durch bessere Enkodierung. Letzteres ist im Falle von TTT besonders interessant, da es für jeden Zustand bis zu acht weitere äquivalente Zustände geben kann.

Wörterzählung: 12.614 Worte

Literaturverzeichnis

[1] W. Ertel, *Introduction to Artificial Intelligence*, Ser. Undergraduate Topics in Computer Science. Cham: Springer International Publishing, 2017, ISBN: 978-3-319-58487-4. DOI: 10.1007/978-3-319-58487-4.

[2] S. J. Russell und P. Norvig, *Artificial Intelligence: A Modern Approach*, Fourth edition, Ser. Pearson Series in Artificial Intelligence. Hoboken: Pearson, 2021, ISBN: 978-0-13-461099-3.

[3] R. S. Sutton und A. G. Barto, *Reinforcement Learning: An Introduction*, Second edition, Ser. Adaptive Computation and Machine Learning Series. Cambridge, Massachusetts: The MIT Press, 2018, 526 S., ISBN: 978-0-262-03924-6.

[4] M. van der Ree und M. Wiering, „Reinforcement Learning in the Game of Othello: Learning against a Fixed Opponent and Learning from Self-Play," in *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, Singapore: IEEE, Apr. 2013, S. 108–115, ISBN: 978-1-4673-5925-2. DOI: 10.1109/ADPRL.2013.6614996.

[5] G. Tesauro, „Temporal Difference Learning of Backgammon Strategy," in *Machine Learning Proceedings 1992*, Elsevier, 1992, S. 451–457, ISBN: 978-1-55860-247-2. DOI: 10.1016/B978-1-55860-247-2.50063-2.

[6] I. Szita, „Reinforcement Learning in Games," in *Reinforcement Learning*, Ser. Adaptation, Learning, and Optimization, M. Wiering und M. van Otterlo, Hrsg., Bd. 12, Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, S. 539–577, ISBN: 978-3-642-27645-3. DOI: 10.1007/978-3-642-27645-3_17.

[7] M. A. Samsuden, N. M. Diah und N. A. Rahman, „A Review Paper on Implementing Reinforcement Learning Technique in Optimising Games Performance," in *2019 IEEE 9th International Conference on System Engineering and Technology (ICSET)*, Shah Alam, Malaysia: IEEE, Okt. 2019, S. 258–263, ISBN: 978-1-72810-758-5. DOI: 10.1109/ICSEngT.2019.8906400.

[8] L. V. Allis, „Searching for solutions in games and artificial intelligence," Ponsen & Looijen, Wageningen, 1994, ISBN: 9789090074887.

[9] K. Crowley und R. S. Siegler, „Flexible Strategy Use in Young Children's Tic-Tac-Toe," *Cognitive Science*, Jg. 17, Nr. 4, S. 531–561, Okt. 1993, ISSN: 03640213. DOI: 10.1207/s15516709cog1704_3. Adresse: http://doi.wiley.com/10.1207/s15516709cog1704_3 (besucht am 02. 01. 2022).

[10] Block-Berlitz, M., „ProInformatik - Funktionale Programmierung: Vom Amateur zum Großmeister - von Spielbäumen und anderen Wäldern," Vorlesung (Freie Universität Berlin SoSe 2009), 2009. Adresse: https://www.inf.fu-berlin.de/lehre/SS09/PI02/docs/MinMax.pdf (besucht am 22. 02. 2022).

[11] Kontes, G., „Seminar: Reinforcement Learning Foundations," Seminar (Fraunhofer IIS), 27. Mai 2021.

[12] R. S. Sutton, „Learning to Predict by the Methods of Temporal Differences,“ *Machine Learning*, Jg. 3, Nr. 1, S. 9–44, 1988, ISSN: 08856125. DOI: 10.1023/A:1022633531479.

[13] C. J. C. H. Watkins und P. Dayan, „Q-learning,“ *Machine Learning*, Jg. 8, Nr. 3, S. 279–292, 1. Mai 1992, ISSN: 1573-0565. DOI: 10.1007/BF00992698.

[14] C. Watkins, „Learning From Delayed Rewards,“ King's College London, 1. Jan. 1989.

[15] D. Ireland. „Comparison - Is There a Fundamental Difference between an Environment Being Stochastic and Being Partially Observable?“ Artificial Intelligence Stack Exchange. (2021), Adresse: https://ai.stackexchange.com/a/33889/51242 (besucht am 19. 01. 2022).

[16] I. Millington und J. D. Funge, *Artificial Intelligence for Games*, 2nd ed. Burlington, MA: Morgan Kaufmann/Elsevier, 2009, 870 S., ISBN: 978-0-12-374731-0.

[17] J. v. Neumann, „Zur Theorie der Gesellschaftsspiele,“ *Mathematische Annalen*, Jg. 100, Nr. 1, S. 295–320, Dez. 1928, ISSN: 0025-5831, 1432-1807. DOI: 10.1007/BF01448847.

[18] C. Shannon und B. Telephone, „XXII. Programming a Computer for Playing Chess 1,“ 1950. DOI: 10.1080/14786445008521796.

[19] D. J. Slate und L. R. Atkin, „CHESS 4.5—The Northwestern University chess program,“ in *Chess Skill in Man and Machine*, P. W. Frey, Hrsg., New York, NY: Springer New York, 1983, S. 82–118, ISBN: 978-0-387-90815-1. DOI: 10.1007/978-1-4612-5515-4_4.

[20] R. D. Greenblatt, D. E. Eastlake und S. D. Crocker, „The Greenblatt Chess Program,“ in *Computer Chess Compendium*, D. Levy, Hrsg., New York, NY: Springer New York, 1988, S. 56–66, ISBN: 978-1-4757-1970-3. DOI: 10.1007/978-1-4757-1968-0_7.

[21] Gardner, M., *Hexaflexagons and Other Mathematical Diversions: The First Scientific American Book of Puzzles & Games*. Chicago: University of Chicago Press, 1988, 200 S., ISBN: 978-0-226-28254-1.

[22] H. Wang, *Popular Lectures on Mathematical Logic*. Dover Publications, 2014, ISBN: 978-1-322-18384-8.

[23] A. Newell und H. A. Simon, *Human Problem Solving*. Englewood Cliffs, N.J: Prentice-Hall, 1972, 920 S., ISBN: 978-0-13-445403-0.

[24] SebGlav. „Creating Tic-Tac-Toe Boards with LaTeX/TikZ without Tables,“ TeX - LaTeX Stack Exchange. (21. Feb. 2022), Adresse: https://tex.stackexchange.com/questions/634666/creating-tic-tac-toe-boards-with-latex-tikz-without-tables (besucht am 22. 02. 2022).

[25] Kutschera, A. „The Best Opening Move in a Game of Tic-Tac-Toe,“ The Kitchen in the Zoo. (2018), Adresse: http://blog.maxant.co.uk/pebble/2018/04/07/1523086680000.html (besucht am 16. 02. 2022).

[26] J. L. V. „Tikz Pgf - Follow-up: Drawing (a Partial) Game Tree of Tic-Tac-Toe,“ TeX - LaTeX Stack Exchange. (2022), Adresse: https://tex.stackexchange.com/q/634724/220502 (besucht am 22. 02. 2022).

[27] Soemers, D. „Game Ai - How Can Both Agents Know the Terminal Reward in Self-Play Reinforcement Learning?“ Artificial Intelligence Stack Exchange. (2018), Adresse: https://ai.stackexchange.com/a/6574/51242 (besucht am 19. 01. 2022).

[28] Mirnov, I. „Q-Learning and Tic-Tac-Toe.“ (2020), Adresse: http://www.iliasmirnov.com/ttt/ (besucht am 23. 02. 2022).

[29] nbro. „Reinforcement Learning - How Are Afterstate Value Functions Mathematically Defined?“ Artificial Intelligence Stack Exchange. (2020), Adresse: https://ai.stackexchange.com/questions/24816/how-are-afterstate-value-functions-mathematically-defined (besucht am 18. 12. 2021).

[30] S. L. Epstein, „Toward an ideal trainer,“ *Machine Learning*, Jg. 15, Nr. 3, S. 251–277, Juni 1994, ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00993346.

[31] A. DiGiovanni und E. C. Zell. „Survey of Self-Play in Reinforcement Learning.“ arXiv: 2107.02850 [cs]. (6. Juli 2021), Adresse: http://arxiv.org/abs/2107.02850 (besucht am 01. 02. 2022).

[32] J. A. Boyan, „Modular Neural Networks for Learning Context-Dependent Game Strategies,“ Master's thesis, Computer Speech and Language Processing, 1992.

[33] M. Bowling, „Convergence and No-Regret in Multiagent Learning,“ University of Alberta Libraries, 2004. DOI: 10.7939/R3ZS2KF41. Adresse: https://era.library.ualberta.ca/items/2225eedc-947a-4ff0-bba4-8c3bd49ddc86 (besucht am 04. 12. 2021).

[34] Slater, N. „Game Ai - Why Is Tic-Tac-Toe Considered a Non-Deterministic Environment?“ Artificial Intelligence Stack Exchange. (2020), Adresse: https://ai.stackexchange.com/questions/22837/why-is-tic-tac-toe-considered-a-non-deterministic-environment (besucht am 21. 12. 2021).

[35] J. Park, D. Yi und S. Ji, „A Novel Learning Rate Schedule in Optimization for Neural Networks and It's Convergence,“ *Symmetry*, Jg. 12, Nr. 4, S. 660, 22. Apr. 2020, ISSN: 2073-8994. DOI: 10.3390/sym12040660. Adresse: https://www.mdpi.com/2073-8994/12/4/660 (besucht am 03. 02. 2022).

[36] J. D. Hunter, „Matplotlib: A 2D Graphics Environment,“ *Computing in Science & Engineering*, Jg. 9, Nr. 3, S. 90–95, 2007, ISSN: 1521-9615. DOI: 10.1109/MCSE.2007.55.

[37] M. L. Littman, „Markov Games as a Framework for Multi-Agent Reinforcement Learning,“ in *In Proceedings of the Eleventh International Conference on Machine Learning*, Morgan Kaufmann, 1994, S. 157–163. DOI: 10.1.1.48.8623.

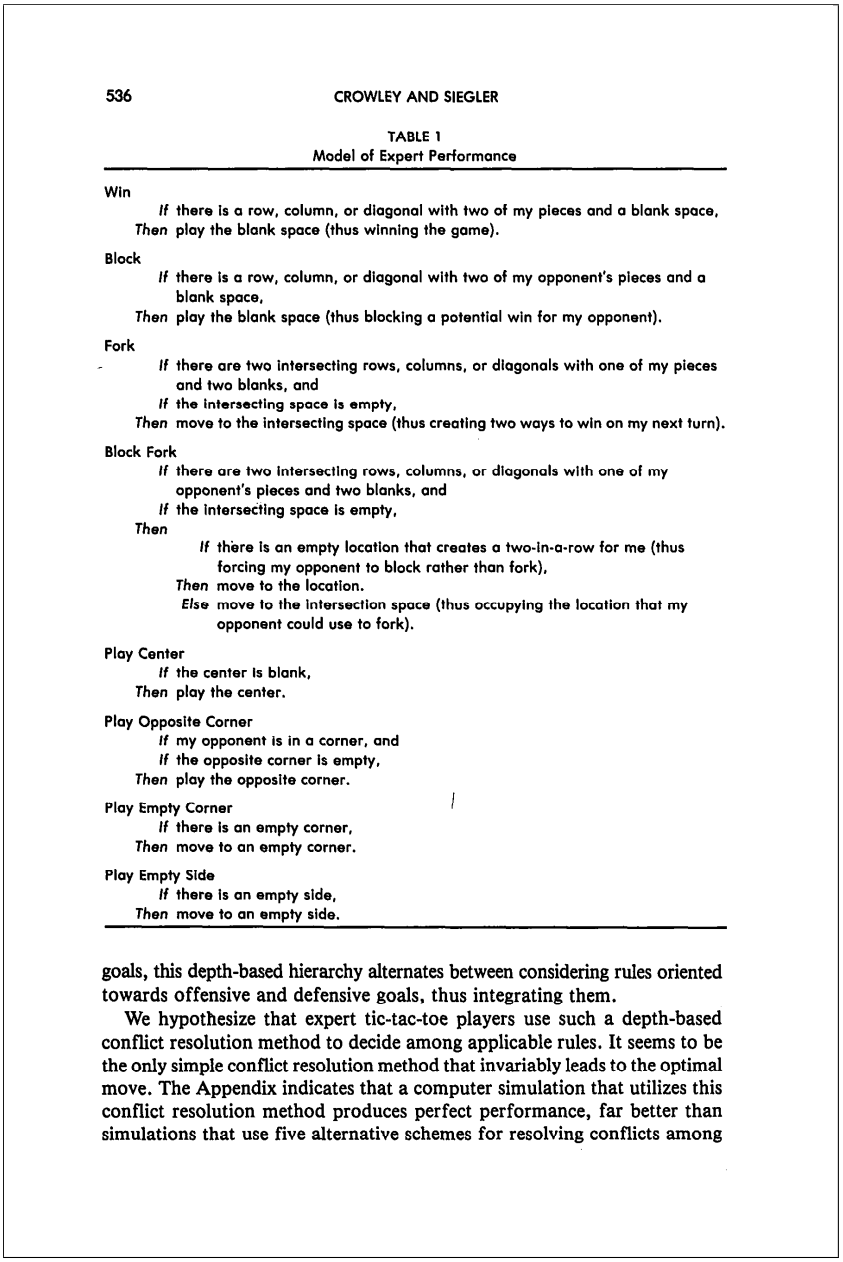
[38] W. Konen und T. Bartz–Beielstein, „Reinforcement Learning: Insights from Interesting Failures in Parameter Selection,“ in *Parallel Problem Solving from Nature – PPSN x*, Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, S. 478–487, ISBN: 978-3-540-87700-4.

[39] G. Neto, „From Single-Agent to Multi-Agent Reinforcement Learning: Foundational Concepts and Methods,“ S. 61,

[40] S. P. Singh und R. S. Sutton, „Reinforcement learning with replacing eligibility traces,“ *Machine Learning*, Jg. 22, Nr. 1-3, S. 123–158, 1996, ISSN: 0885-6125, 1573-0565. DOI: 10.1007/BF00114726.

Anhangsverzeichnis

A. Model of Expert Performance



536 CROWLEY AND SIEGLER

TABLE 1
Model of Expert Performance

Win
If there is a row, column, or diagonal with two of my pieces and a blank space,
Then play the blank space (thus winning the game).

Block
If there is a row, column, or diagonal with two of my opponent's pieces and a blank space,
Then play the blank space (thus blocking a potential win for my opponent).

Fork
If there are two intersecting rows, columns, or diagonals with one of my pieces and two blanks, and
If the intersecting space is empty,
Then move to the intersecting space (thus creating two ways to win on my next turn).

Block Fork
If there are two intersecting rows, columns, or diagonals with one of my opponent's pieces and two blanks, and
If the intersecting space is empty,
Then
If there is an empty location that creates a two-in-a-row for me (thus forcing my opponent to block rather than fork),
Then move to the location.
Else move to the intersection space (thus occupying the location that my opponent could use to fork).

Play Center
If the center is blank,
Then play the center.

Play Opposite Corner
If my opponent is in a corner, and
If the opposite corner is empty,
Then play the opposite corner.

Play Empty Corner
If there is an empty corner,
Then move to an empty corner.

Play Empty Side
If there is an empty side,
Then move to an empty side.

goals, this depth-based hierarchy alternates between considering rules oriented towards offensive and defensive goals, thus integrating them.

We hypothesize that expert tic-tac-toe players use such a depth-based conflict resolution method to decide among applicable rules. It seems to be the only simple conflict resolution method that invariably leads to the optimal move. The Appendix indicates that a computer simulation that utilizes this conflict resolution method produces perfect performance, far better than simulations that use five alternative schemes for resolving conflicts among

Abbildung A.1.: Model of Expert Performance von Crowley und Siegler, Auszug aus [9, S. 536]

B. Beispiel für Meta-Log der Implementierung

Listing B.1.: Meta-Log der Implementierung

```
Algorithm: Q-Learning
Experience: Q-Table
depth penalty applied to reward: true
Training method: Normal Self-play
Number of training episodes: 150000

Alpha Change Mode: CONSTANT
  Value of Alpha: 0.1
Epsilon Change Mode: DEGRESSIVE_DECAY
  Initial Value of Epsilon: 1.0
  Final Value of Epsilon: 0.1
  Final value reached after 100000 episodes

Training using Self-play
Number of TRAIN episodes: 150000
Games won by X: 50736
Games won by O: 30002
Games ended in draw: 69262

Experience entries after training: 5478
Evaluation againstMinimax
Agent playing as: SYMBOL_X
Number of EVAL episodes: 10000
Games won by X: 0
Games won by O: 0
Games ended in draw: 10000

Evaluation againstRandom
Agent playing as: SYMBOL_X
Number of EVAL episodes: 10000
Games won by X: 9958
Games won by O: 0
Games ended in draw: 42

Evaluation againstMinimax
Agent playing as: SYMBOL_O
Number of EVAL episodes: 10000
Games won by X: 0
```

```
Games won by O: 0
Games ended in draw: 10000

Evaluation againstRandom
Agent playing as: SYMBOL_O
Number of EVAL episodes: 10000
Games won by X: 0
Games won by O: 9145
Games ended in draw: 855

Experience entries after evaluation: 5478
```

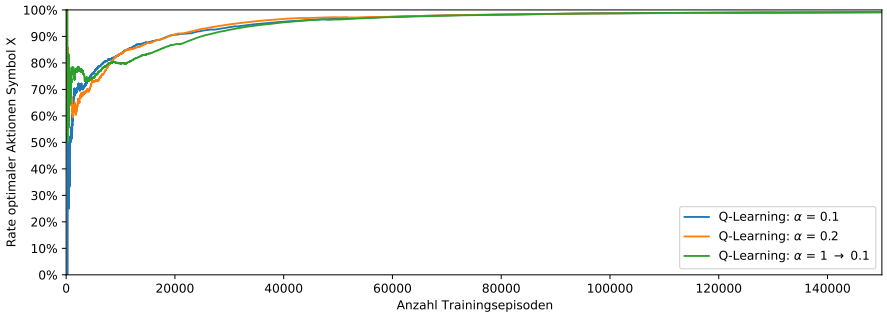
C. minimax-Methode

Listing C.1.: minimax-Methode der Klasse MinimaxAlgorithm

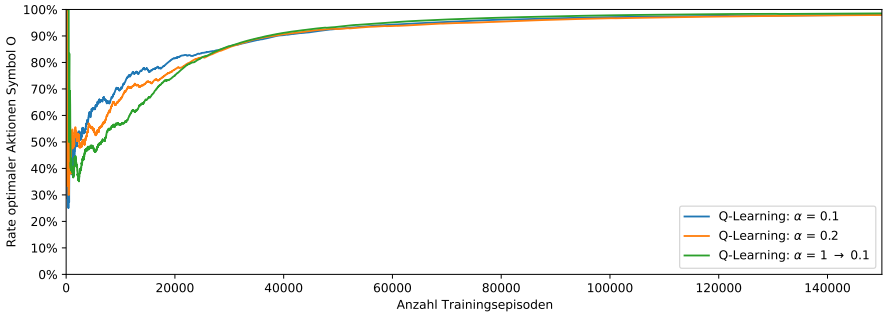
```
1  /**
2           * Implementation of the recursive minimax algorithm that constructs a complete
3           * gametree for the passed gamefield and derives the optimal moves for each
4           * player in every node The nodes are saved in the transposition table
5           * @param depth  current depth in the game tree of the node for the node to be
6           *               calculated
7           * @param symbol player that is allowed to act in this node
8           * @return Node in the gametree that was analyzed containing optimal actions and
9           *         expected score
10          */
11         private Node minimax(int depth, Symbol symbol) {
12                 int[] legalActions = this.internalGamefield.getLegalActions();
13                 if (legalActions.length == 0) {
14                         // if terminal state (leaf) create a leafnode, assign the score and
                           ↪  return
15                         Node leaf = new Node(symbol);
16                         leaf.setDepth(depth);
17                         leaf.setScore(this.calculateScore(depth));
18                         return leaf;
19                 }
20                 Node currentNode = new Node(symbol);
21
22                 int initialScore = symbol.isX() ? -1 : 1;
23                 currentNode.setScore(initialScore);
24
25                 Symbol nextSymbol = Symbol.getNextSymbol(symbol);
26
27                 for (int legalAction : legalActions) {
28                         this.internalGamefield.applyAction(symbol, legalAction);
29
30                         Node nextNode = this.transpositionTable.retrieveNode(this
                           ↪  .internalGamefield.getState());
31
32                         if (nextNode == null) {
33                                 nextNode = this.minimax(depth + 1, nextSymbol);
34                                 this.transpositionTable.putNode(this.internalGamefield
                                   ↪  .getState(),
                                   ↪  nextNode);
35                         }
36                         this.updateCurrentNode(currentNode, nextNode, legalAction, symbol);
37
38                         this.internalGamefield.undoAction(symbol, legalAction);
39                 }
40                 return currentNode;
41         }
```

D. Zusätzliche Auswertung zu Q-Learning

D.1. Konvergenz alternierendes Self-play



(a) Symbol X



(b) Sybmol O

Abbildung D.1.: Rate optimaler Aktionen von Q-Learning für verschiedene Lernraten α, alternierendes Self-play (a) Symbol X (b) Symbol O

D.2. Spielergebnismatritzen klassisches Self-play

Tabelle D.1.: Spielergebnismatrix für Q-Learning mit Lernrate $\alpha = 0,2$, klassisches Self-play

	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	98,91% ± 0,31%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	1,09% ± 0,31%
O Q-Learning	X Minimax:	0,00% ± 0,00%	X Random:	0,00% ± 0,00%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	90,84% ± 0,49%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	9,16% ± 0,49%

Tabelle D.2.: Spielergebnismatrix für Q-Learning mit abnehmender Lernrate, klassisches Self-play

	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	98,91% ± 0,24%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	1,09% ± 0,24%
O Q-Learning	X Minimax:	0,00% ± 0,00%	X Random:	0,00% ± 0,00%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	91,49% ± 0,35%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	8,51% ± 0,35%

D.3. Spielergebnismatritzen alternierendes Self-play

Tabelle D.3.: Spielergebnismatrix für Q-Learning mit Lernrate $\alpha = 0,1$, alternierendes Self-play

	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	95,12% ± 1,75%
	O Minimax:	0,00% ± 0,00%	O Random:	0,22% ± 0,16%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	4,66% ± 1,64%
O Q-Learning	X Minimax:	0,00% ± 0,00%	X Random:	0,96% ± 0,50%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	81,96% ± 2,49%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	17,08% ± 2,59%

Tabelle D.4.: Spielergebnismatrix für Q-Learning mit Lernrate $\alpha = 0,2$, alternierendes Self-play

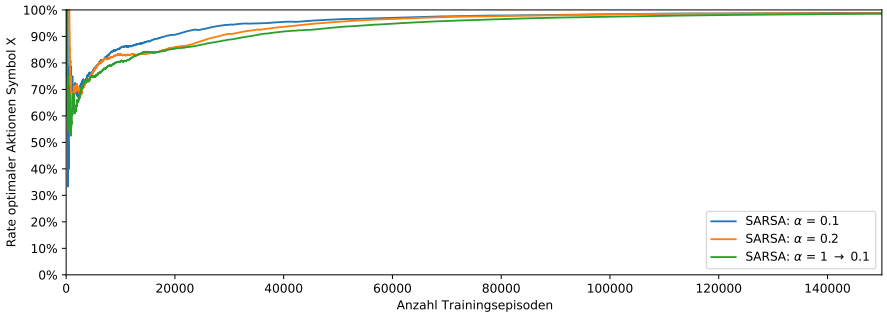
	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	96,52% ± 2,72%
	O Minimax:	0,00% ± 0,00%	O Random:	0,07% ± 0,09%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	3,41% ± 2,63%
O Q-Learning	X Minimax:	0,04% ± 0,09%	X Random:	0,47% ± 0,18%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	84,37% ± 1,94%
	Unentschieden:	99,96% ± 0,09%	Unentschieden:	15,17% ± 1,93%

Tabelle D.5.: Spielergebnismatrix für Q-Learning mit abnehmender Lernrate, alternierendes Self-play

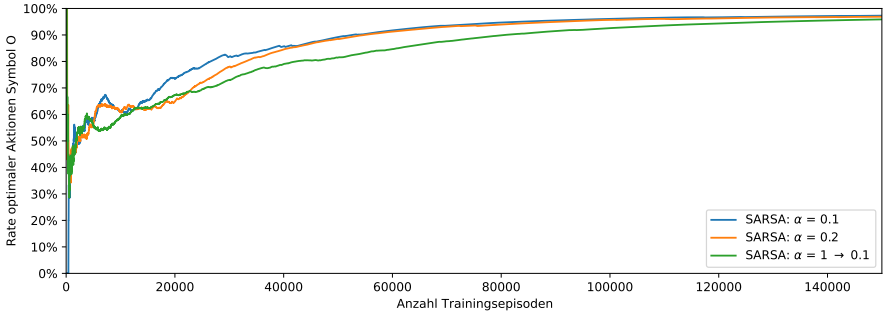
	Minimax		**Random**	
X Q-Learning	X Q-Learning:	0,00% ± 0,00%	X Q-Learning:	93,85% ± 0,57%
	O Minimax:	0,00% ± 0,00%	O Random:	0,21% ± 0,11%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	5,94% ± 0,47%
O Q-Learning	X Minimax:	0,40% ± 0,80%	X Random:	0,28% ± 0,37%
	O Q-Learning:	0,00% ± 0,00%	O Q-Learning:	79,32% ± 3,14%
	Unentschieden:	99,60% ± 0,80%	Unentschieden:	20,40% ± 2,96%

E. Zusätzliche Auswertung zu Sarsa

E.1. Konvergenz alternierendes Self-play



(a) Symbol X



(b) Sybmol O

Abbildung E.1.: Rate optimaler Aktionen von Sarsa für verschiedene Lernraten α, alternierendes Self-play (a) Symbol X (b) Symbol O

E.2. Spielergebnismatritzen klassisches Self-play

Tabelle E.1.: Spielergebnismatrix für Sarsa mit Lernrate $\alpha = 0,2$, klassisches Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	98,92% ± 0,09%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	1,08% ± 0,09%
O Sarsa	X Minimax:	0,85% ± 1,05%	X Random:	0,18% ± 0,16%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	87,25% ± 1,37%
	Unentschieden:	99,15% ± 1,05%	Unentschieden:	12,57% ± 1,21%

Tabelle E.2.: Spielergebnismatrix für Sarsa mit abnehmender Lernrate, klassisches Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	98,68% ± 0,50%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	1,32% ± 0,50%
O Sarsa	X Minimax:	0,73% ± 1,18%	X Random:	0,14% ± 0,18%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	87,19% ± 2,22%
	Unentschieden:	99,27% ± 1,18%	Unentschieden:	12,67% ± 2,12%

E.3. Spielergebnismatritzen alternierendes Self-play

Tabelle E.3.: Spielergebnismatrix für Sarsa mit Lernrate $\alpha = 0,1$, alternierendes Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	95,97% ± 2,93%
	O Minimax:	8,81% ± 15,25%	O Random:	1,38% ± 2,02%
	Unentschieden:	91,19% ± 15,25%	Unentschieden:	2,65% ± 1,03%
O Sarsa	X Minimax:	3,29% ± 4,41%	X Random:	2,48% ± 0,51%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	84,36% ± 1,09%
	Unentschieden:	96,71% ± 4,41%	Unentschieden:	13,15% ± 0,96%

Tabelle E.4.: Spielergebnismatrix für Sarsa mit Lernrate $\alpha = 0,2$, alternierendes Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	97,15% ± 1,42%
	O Minimax:	0,00% ± 0,00%	O Random:	0,60% ± 0,63%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	2,25% ± 0,84%
O Sarsa	X Minimax:	3,29% ± 4,17%	X Random:	1,54% ± 0,62%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	85,22% ± 1,19%
	Unentschieden:	96,71% ± 4,17%	Unentschieden:	13,24% ± 1,19%

Tabelle E.5.: Spielergebnismatrix für Sarsa mit abnehmender Lernrate, alternierendes Self-play

	Minimax		**Random**	
X Sarsa	X Sarsa:	0,00% ± 0,00%	X Sarsa:	98,80% ± 0,10%
	O Minimax:	0,00% ± 0,00%	O Random:	0,00% ± 0,00%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	1,20% ± 0,10%
O Sarsa	X Minimax:	0,00% ± 0,00%	X Random:	0,38% ± 0,20%
	O Sarsa:	0,00% ± 0,00%	O Sarsa:	86,83% ± 0,54%
	Unentschieden:	100,00% ± 0,00%	Unentschieden:	12,79% ± 0,37%