
universität
**u**ulm

**Fakultät für**
**Mathematik und**
**Wirtschafts-**
**wissenschaften**

# Entwurf, Implementierung und Evaluierung von verschiedenen Machine Learning Algorithmen für das Strategielernen mit dem praktischen Versuch am Spiel Oware Abapa

**Masterarbeit bei der Firma SYSTECS Informationssysteme GmbH**

**Vorgelegt von:**
Daniela Hirsch
daniela.hirsch@uni-ulm.de

**Gutachter:**
Prof. Dr. Friedhelm Schwenker
Dr. Thomas Zurawka

2023

Fassung 28. März 2023

© 2023 Daniela Hirsch

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 3.0 License. To view a copy of this license, visit http://creativecommons.org/licenses/by-nc-sa/3.0/de/ or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.

Satz: PDF-LaTeX $2_\varepsilon$

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Danksagung

Ich möchte mich an dieser Stelle bei allen bedanken, die mich bei der Fertigstellung dieser Masterarbeit unterstützt haben.

Mein besonderer Dank gebührt dabei Prof. Dr. Friedhelm Schwenker von der Universität Ulm und Dr. Thomas Zurawka von der Firma SYSTECS Informationssysteme GmbH, die mich durch ihre kompetente Betreuung unterstützt und sich bereitgestellt haben als Gutachter zu fungieren.

Mein Dank geht auch an meine Familie, meinen Freund Magnus und meine Kommilitoninnen Jana und Pauline, die diese Arbeit gegengelesen haben und mir auch sonst im Studium gut zur Seite standen.

Ich danke auch allen Kollegen von SYSTECS, die mich gut in der Firma aufgenommen haben und mir ein Umfeld geschaffen haben, in welchem ich mich wohl fühlte.

# 1 Einleitung

Zwei-Personen-Brettspiele sind seit jeher ein auf der gesamten Welt verbreiteter Bestandteil der menschlichen Gesellschaft. Um diese Art von Spielen bestreiten zu können, ist ein würdiger Gegner von Nöten, welcher die teilweise sehr komplexen Spielzusammenhänge versteht, Züge vorausdenken kann und somit eine Herausforderung darstellt. Erst dann kann ein Spielspaß garantiert werden.
Eine dahingegen sehr junge Entwicklung der Menschheit stellt die Erforschung von künstlicher Intelligenz (KI) dar. Heutzutage wird sie in vielen Bereichen, wie zum Beispiel in der Medizin oder Automobilindustrie eingesetzt. Das Potenzial von KI und Machine Learning ist aber bei Weitem noch nicht ausgeschöpft. Da für Zwei-Personen-Spiele nicht immer ein würdiger menschlicher Gegner vorhanden ist, kann versucht werden, diesen durch einen Bot, der mit Hilfe von künstlicher Intelligenz das Spiel lernt, zu ersetzen.
Diese Arbeit beschäftigt sich mit der Frage, welche KI-Algorithmen für Zwei-Personen-Spiele geeignet sind, wie diese in der Theorie aufgebaut sind, wie sie sich implementieren lassen und wie sie im Vergleich zueinander abschneiden. Diese Problemstellung soll anhand des Spiels Oware Abapa erörtert werden. Dafür gliedert sich die Thesis in vier größere Abschnitte.

Zu Beginn der Arbeit, in **Kapitel 2**, erklären wir das Spiel Oware Abapa schematisch. Wir geben die Regeln des Spiels an, um klare Anforderungen an die KI- bzw. Machine Learning-Algorithmen stellen zu können.

In **Kapitel 3** stellen wir die Theorie der Algorithmen vor. Dabei betrachten wir einen Zufallsalgorithmus „Random“, Suchbaumverfahren wie Adversarial Search und Monte Carlo Tree Search, sowie ausgewählte Reinforcement Learning Algorithmen. Für den zuletzt genannten Punkt beschränken wir uns in dieser Arbeit auf die Q-Learning Methode und die Policy Gradient Optimierung.
Wir erklären Ansätze der Spieltheorie und wie diese im Bezug auf Oware Abapa zu verstehen sind. Mit Hilfe dieser theoretischen Betrachtungen geben wir eine umfangreiche Abschätzung hinsichtlich der Komplexität und Chancengleichheit der Spieler für die Adversarial Search Algorithmen an.

Um eine Vorstellung zu erhalten, wie sich die theoretisch dargestellten Algorithmen aus Kapitel 3 in der Praxis umsetzen lassen, erklären wir in **Kapitel 4**, wie eine Implementierung in Python durchgeführt werden kann. Dabei geben wir die Klassenstrukturen der einzelnen Programmbestandteile an, heben hervor, welche Python-Funktionen sich am besten eignen und legen dar, auf welche Art und Weise die Reinforcement Learning Algorithmen trainiert werden und wie das Trainingsergebnis erklärt werden kann.

Ein abschließender Vergleich der implementierten KI-Algorithmen in **Kapitel 5** soll zeigen, wie gut diese für das Zwei-Personen-Spiel Oware Abapa geeignet sind. Dabei untersuchen wir die Gewinnchancen, indem die Algorithmen gegeneinander getestet werden, aber auch wie viel Rechenzeit sie benötigen und wie schnell ein Algorithmus das Spiel für sich entscheidet.

# 2 Das Spiel Oware Abapa

In diesem Kapitel möchten wir eine kurze Einführung in das Regelwerk von Oware Abapa geben und spielbeschreibende Begriffe hervorheben, die in den nachfolgenden Kapiteln weiter gebraucht werden. Die Spielregeln, die wir nun präsentieren, finden sich in [17]. Oware Abapa stammt aus Afrika und ist eines der ältesten Brettspiele, das weltweit verbreitet ist.

Bei Oware Abapa handelt es sich um ein Spiel für zwei Personen. Das Spielbrett besteht aus 12 Vertiefungen, die groß genug sind, um sie mit Steinen zu befüllen. Wir nennen diese Vertiefungen *Felder*. Die 12 Felder sind in zwei *Reihen* zu je 6 Feldern angeordnet, von denen jeder Spieler eine Reihe erhält. Zu Beginn des Spiels liegen in jedem Feld 3 Steine. In diesem Punkt weichen wir von [17] ab, da dort von 4 Steinen pro Feld gesprochen wird. Insgesamt befinden sich also 36 Steine im Spiel. Eine schematische Darstellung findet sich in Abbildung 2.1.

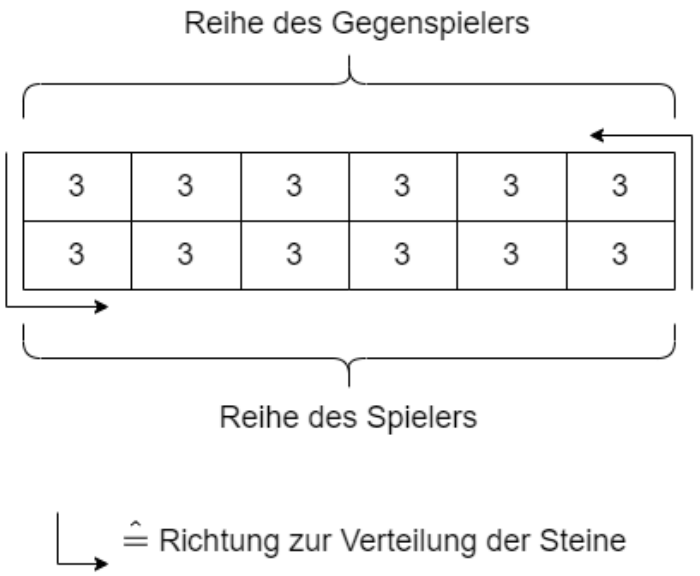


Abbildung 2.1: Aufbau und Zuordnung des Spielbrettes von Oware Abapa.

Ist ein Spieler am Zug, darf er aus seiner Reihe ein Feld bestimmen, welches sein *Startfeld* darstellt. Er nimmt dann aus diesem Startfeld alle sich darin befindenden Steine in die Hand und verteilt sie gegen den Uhrzeigersinn über beide Reihen hinweg. Ein Beispiel findet sich in Abbildung 2.2. Hier hat der Spieler das letzte Feld seiner Reihe als Startfeld gewählt und die Steine gegen den Uhrzeigersinn verteilt.

| | | | | | |
|---|---|---|---|---|---|
| 3 | 3 | 3 | 4 | 4 | 4 |
| 3 | 3 | 3 | 3 | 3 | |

Abbildung 2.2: Spielfeld nach dem ersten Zug des Spielers bei Wahl des letzten Feldes als Startfeld.

Ist die Wahl eines Spielers auf ein Startfeld gefallen, in dem sich mehr als 11 Steine befinden, und er diese somit einmal im Kreis verteilen würde, darf er beim Befüllen, wenn er wieder auf das Startfeld trifft, dieses überspringen.
Beinhaltet das Feld, in das der letzte Stein gefallen ist, nun zwei oder drei Steine und gehört das Feld dem Gegenspieler, so darf der Spieler diese Steine entnehmen. Man nennt diesen Prozess *Kapern*. Dann betrachtet man von diesem Feld aus der Reihe nach die direkt davor aufgefüllten Felder des Gegenspielers. Wurden diese auch auf zwei oder drei Steine aufgefüllt, dürfen diese Steine ebenfalls erbeutet werden. Stößt man allerdings auf ein Feld, bei dem das nicht zutrifft, bricht das Kapern ab. Dann ist die *Runde* des Spielers beendet und der Gegenspieler ist am Zug.
Das Ziel des Spiels ist es, am meisten Steine zu kapern. Wenn beide Spieler genau 18 Steine für sich beanspruchen konnten, liegt ein Unentschieden vor.
Das *Spielende* tritt ein, wenn ein Spieler mehr als 18 Steine kapern konnte und damit gewonnen hat, oder wenn ein Spieler keine Steine mehr in seinen Feldern hat und somit nicht mehr ziehen kann. Dann darf der Gegenspieler alle Steine in seiner Reihe kapern. Anschließend findet ein Auszählen der erbeutenden Steine und das Ermitteln des Gewinners statt.
Den Fall, dass ein Spieler keine Steine mehr in seiner Reihe liegen hat, kann man bewusst versuchen herbeizuführen. Dieses Verhalten nennt man *Aushungern* eines Spielers. Diese Regel weicht ebenfalls von [17] ab, verhindert aber, dass das Spiel in einer Art Endlosschleife feststeckt, wenn nur noch wenige Steine im Spiel sind und so ein Auffüllen eines Feldes zu 2 oder 3 Steinen nicht mehr möglich ist.

# 3 KI-Algorithmen in der Theorie

In diesem Kapitel werden wir verschiedene KI-Algorithmen vorstellen und mathematisch untersuchen. Diese Algorithmen sind die im Folgenden aufgelisteten:

(i) Random (Zufällige Spielzüge für den späteren Vergleich),

(ii) Adversarial Search mit dem Minimax- und Alpha-Beta-Pruning-Algorithmus,

(iii) Monte Carlo Tree Search

(iv) und insbesondere Algorithmen zum verstärkenden Lernen (Reinforcement Learning).

Anschließend stellen wir Neuronale Netze vor und erklären, wie die Reinforcement Learning-Algorithmen damit umgesetzt werden können.

## 3.1 Random

Um einen Vergleich für die strategielernenden Algorithmen zu erhalten, benötigen wir einen Algorithmus, welcher die Spielzüge zufällig wählt. Die KI-Algorithmen sollten gegenüber dieser Methode eine deutliche Verbesserung hinsichtlich der Gewinnchance erzielen.
Bei unserem Spiel wird also die Auswahl für das Startfeld der Runde zufällig bestimmt. Der weitere Verlauf der Runde ist dann deterministisch und nicht weiter beeinflussbar. Die Auswahl des Startfeldes erfolgt anhand einer Gleichverteilung. Leere Felder werden dabei nicht berücksichtigt.

## 3.2 Adversarial Search

In diesem Kapitel erklären wir Adversarial Suchalgorithmen anhand unseres Spiels Oware Abapa. Unser erster Algorithmus ist der Minimax-Algorithmus. Dabei orientieren wir uns an [19].
Darauf aufbauend untersuchen wir den Aufwand dieses Algorithmus für das Spiel Oware Abapa und geben geeignete Strategien für die spätere Implementierung an. Anschließend betrachten wir den Alpha-Beta-Pruning Algorithmus, welcher eine Verbesserung des Minimax-Algorithmus darstellt.

Ein Adversarial-Lernproblem kann wie in [5] formal beschrieben werden. Es seien Trainings- und Testdaten gegeben, anhand deren folgendes Klassifizierungsproblem gelöst werden soll. Es sei $X \subset \mathbb{R}^d$, wobei $d$ die Anzahl der Merkmale der Trainingsdaten darstellen. In diesem Klassifizierungsproblem wird eine Funktion

$$f : X \to \{-1, 1\}$$

berechnet, welche einer beliebigen Eingabe aus $X$ das richtige Label aus $\{-1, 1\}$ zuordnet. Das Adversarial-Lernproblem bringt nun einen Gegner (engl.: adversarial) ins Spiel, welcher die Eingabe $x \in X$ mit einer Abweichung $\delta$ in der Testphase stört, mit dem Ziel, dass $f(x) \neq f(x + \delta)$. Das Ziel dieses Lernproblems ist es nun eine robuste Funktion $f$ zu finden, sodass

$$P(f(x) \neq f(x + \delta)) < \epsilon$$

für ein beliebiges $\epsilon > 0$.
Diese abstrakte Betrachtung lässt sich auf die Theorie von Spielen anwenden. Das Adversarial-Problem bezeichnet hierbei das Handeln eines führenden Spielers und seines Gegners, bei dem der Spieler versucht, die durch die störenden Aktionen des Gegners entstandenen Verluste gering zu halten und seinen Gewinn zu maximieren. Dies führt uns zu unserem erstem Algorithmus.

### 3.2.1 Der Minimax-Algorithmus

Bei Oware Abapa handelt es sich um ein sogenanntes *Zwei-Personen-Null-Summen-Spiel* mit *vollständiger Information*. Die Bedeutung des Begriffes setzt sich aus folgenden Definitionen zusammen:

**Definition 3.2.1.1.** Der Ausdruck *Zwei-Personen* gibt an, dass es sich um ein Spiel handelt, bei dem zwei Parteien gegeneinander spielen. Diese Parteien sind der erste Spieler $p_1$ und sein Gegner $p_2$. Die Spieler $p_1$ und $p_2$ werden oft auch *MAX* und *MIN* genannt, da sie im späteren Verlauf die Berechnung des Maximums bzw. Minimums übernehmen. Außerdem muss das Spiel nach endlich vielen Zügen zum Ende kommen.

**Definition 3.2.1.2.** Unter dem Wort *Null-Summen* versteht man, dass der Gewinn des Spielers $p_1$ den Verlust des Gegners $p_2$ auf Null ausgleicht und umgekehrt. Wir nehmen für unser Beispiel an, dass der Gewinn mit der Anzahl gekaperter Steine belohnt wird, ein Verlust mit $(-1)\cdot$Anzahl der gekaperten Steine bestraft wird und ein Unentschieden $0$ ergibt.

**Definition 3.2.1.3.** Ein Spiel mit *vollständiger Information* ist ein Spiel, bei dem beide Parteien über den aktuellen Zustand des Spiels und alle Zugmöglichkeiten beider Spieler Bescheid wissen. Es gibt also keine zufälligen Ereignisse.
Da bei Oware Abapa ein offenes Spielfeld vorliegt und der Spieler sich in jeder Runde frei entscheiden kann, mit welchem Feld er anfängt, liegt diese Eigenschaft ebenfalls für unser Spiel vor.

Für Oware Abapa bezeichnet ein *Zustand* des Spiels die Informationen über das aktuelle Spielbrett und wie viele Steine jeder Spieler bereits gekapert hat. Eine *Aktion* bezeichnet die Wahl des Startfeldes, die ein Spieler trifft, wenn er an der Reihe ist. Vorab führen wir einige Definitionen ein, die ein Suchbaum-Spielverfahren beschreiben:

**Definition 3.2.1.4.**

(i) Sei $S$ die Menge aller möglichen Zustände des Spiels. Eine *Gewinnauszahlungsfunktion* oder *utility function*

$$f : S \to \mathbb{Z}$$

ist eine Funktion, die jedem Zustand $s \in S$ seinen Gewinn bzw. Verlust eines Spielers zuordnet.

(ii) Sei $A_s$ die Menge aller möglichen Aktionen, die ein Spieler zum Zustand $s$ des Spiels ausführen kann. Damit lässt sich eine Funktion

$$g_s : A_s \to S$$

definieren, welche jeder Aktion $a \in A_s$ einen Folgezustand in $S$ zuordnet.

(iii) Die Zustände und Aktionen lassen sich in einem *Spielbaum* darstellen. Jeder Knoten des Baumes bildet einen Zustand und jede Kante eine Aktion ab. Für den Minimax-Algorithmus ist ein Spielbaum wie in [19] beschrieben, ein Baum, dessen Ebenen sich zwischen *MAX* und *MIN*, also zwischen Spieler $p_1$ und $p_2$ abwechseln.

**Beispiel 3.2.1.5.** In Abbildung 3.1 ist ein Spielbaum dargestellt, welcher ein Spiel repräsentiert, das nach zwei Zügen endet. Für jeden Zustand $s_i,\ i = 0, ..., 3$ (also alle außer die Endzustände) gibt es drei mögliche Aktionen der Menge $A_{s_i}$.

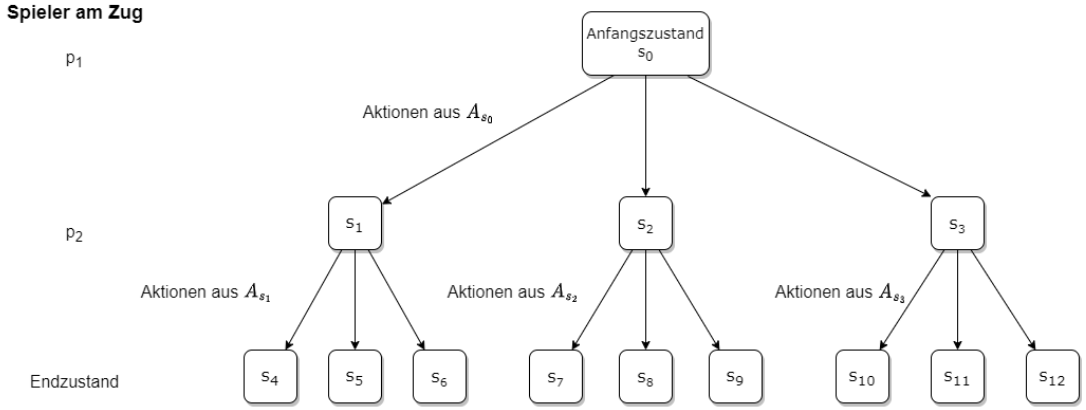


Abbildung 3.1: Spielbaum mit zwei Zügen und jeweils drei möglichen Aktionen pro Zustand

Um eine geeignete Spielstrategie für $p_1$ und $p_2$ zu finden, führen wir nun mit Hilfe dieser Definitionen den *Minimax-Wert* wie in [19] ein:

**Definition 3.2.1.6.** Der *Minimax-Wert* $v(s)$ für einen Zustand $s \in S$, dargestellt im Knoten $j$ des Baumes, ist gegeben durch

$$v(s) = \begin{cases} f(s), & \text{j ist ein Endknoten des Baumes} \\ \max\limits_{a \in A_S} v(g_s(a)), & \text{Spieler } p_1 \text{ ist am Zug} \\ \min\limits_{a \in A_S} v(g_s(a)), & \text{Spieler } p_2 \text{ ist am Zug} \end{cases} \tag{3.1}$$

Das Ziel des $p_1$-Spielers, der die *MAX*-Rolle übernimmt, ist es also, bei gleichzeitig

optimalem Gegenspiel, den Minimax-Wert zu maximieren. Das Ziel des Gegners $p_2$ ist es, diesen zu minimieren. Dementsprechend wählen die Spieler abwechselnd die Aktion $a \in A_s$, die zu dem größtmöglichen bzw. kleinstmöglichen Minimax-Wert für ihren Zustand $s$ führt.

**Beispiel 3.2.1.7.** Wir veranschaulichen dies nun mit einem konkreten Zahlenbeispiel. Gegeben seien die Werte der Gewinnauszahlungsfunktion in den Endzuständen, mit Hilfe derer wir die Minimax-Werte in den Knoten von unten nach oben berechnen können.

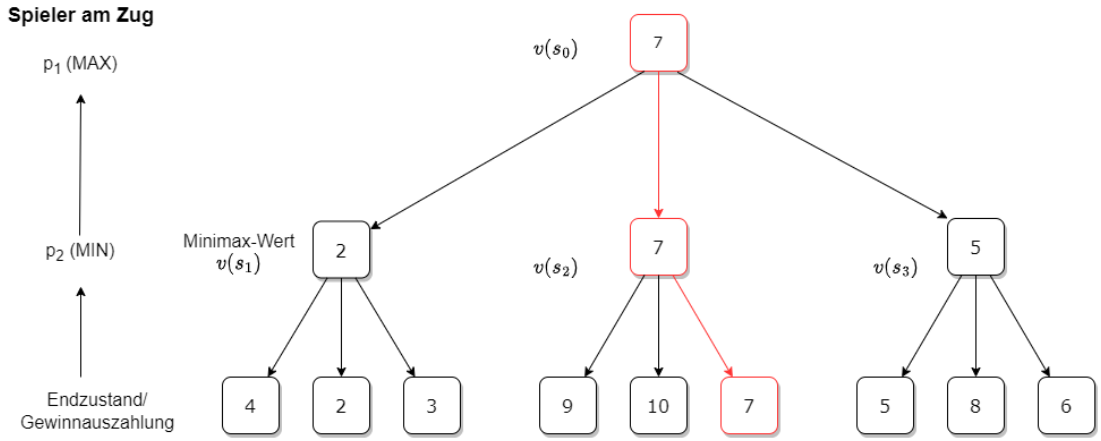


Abbildung 3.2: Spielbaum mit zwei Zügen und Berechnung des Minimaxwertes. Der rote Pfad gibt den gewählten Weg an.

Der Minimax-Wert für den Anfangszustand ist also

$$v(s_0) = \max\{\min\{4,2,3\}, \min\{9,10,7\}, \min\{5,8,6\}\} = \max\{2,7,5\} = 7.$$

**Algorithmus 3.2.1.8.** Im Folgenden möchten wir einen Pseudocode für die erarbeitete Spielstrategie angeben. Dabei orientieren wir uns wieder an [19].
Es handelt sich hierbei um einen tiefenorientierten Minimax-Algorithmus, das bedeutet, dass „erst dann ein neuer Knotennachfolger expandiert wird, wenn alle lexikographisch vorangehenden, das heißt in Abbildungen links eingezeichneten Knotennachfolger bis zur maximalen Suchtiefe expandiert worden sind.“ [19]
Da jede Aktion genau einen Zustand bedingt, ist die Berechnung des Baumes deterministisch.

**Algorithm** Der Minimax-Algorithmus

```
1: function MAXIMUM(s)
2:     if s ist ein Endzustand then
3:         return f(s)
4:     end if
5:     value ← −∞;
6:     for a ∈ A_s do
7:         value ← max(value, MINIMUM(g_s(a)));
8:     end for
9:     return value
10: end function
11:
12: function MINIMUM(s)
13:     if s ist ein Endzustand then
14:         return f(s)
15:     end if
16:     value ← +∞;
17:     for a ∈ A_s do
18:         value ← min(value, MAXIMUM(g_s(a)));
19:     end for
20:     return value
21: end function
```

Die Anzahl der Zustände des Spiels Oware Abapa lässt sich mit Hilfe der Kombinatorik [14] berechnen: Das Spielfeld besteht aus 12 Feldern, in welchen zu Beginn je 3 Steine liegen. Insgesamt sind also zu Beginn 36 Steine im Spiel. Die Anzahl der Zustände berechnet sich nun daraus, wie viele Möglichkeiten existieren, die Steine auf das Spielfeld zu verteilen.
Da die Steine jedes Mal neu verteilt werden und nicht unterscheidbar sind, handelt es sich hier um ein Urnen-Modell mit Zurücklegen und ohne Betrachtung der Reihenfolge. Für $n$ Steine und $k$ Felder existieren nach [11] für dieses Modell

$$\binom{n+k-1}{k} = \binom{n+k-1}{n-1}$$

Möglichkeiten. Da im Laufe des Spiels Steine weggenommen werden, addieren wir für jede Anzahl an Steinen im Spiel $n \in \{0, 1, 2, ..., 36\}$ die Anzahl der Möglichkeiten, wie diese auf dem Spielfeld verteilt sein können. Für $n = 0$ handelt es sich um ein leeres Spielfeld, das genau einen Zustand ergibt.

Insgesamt existieren somit bis zu

$$1 + \sum_{n=1}^{36} \binom{n-1+12}{n-1}$$

Zustände. Mit Hilfe einer Rechenregel für Binomialkoeffizienten $\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$ für $n, k \in \mathbb{N}$ lässt sich induktiv eine Formel für Summen wie in [14] herleiten:

$$\sum_{k=0}^{m} \binom{n+k}{k} = \binom{n+m+1}{n+1} \tag{3.2}$$

für $m, n, k \in \mathbb{N}$. Für unseren Fall ergeben sich also mit Hilfe eines Indexshifts

$$\begin{aligned} 1 + \sum_{n=1}^{36} \binom{n-1+12}{n-1} &= 1 + \sum_{n=0}^{35} \binom{n+12}{n} \\ &\overset{(3.2)}{=} 1 + \binom{12+35+1}{12+1} \\ &= 1 + \binom{48}{13} = 192.928.249.297, \end{aligned}$$

also etwa $1,93 \cdot 10^{11}$ mögliche Zustände.
Da dies eine extrem große Anzahl an Zuständen ist, wird es nicht möglich sein, den Spielbaum in angemessener Zeit komplett zu berechnen. Deshalb ändern wir den Minimax-Algorithmus zu unseren Gunsten, indem wir eine maximale Suchtiefe $t > 0$ vorgeben. Dieses Prinzip wird im nächsten Abschnitt aufgegriffen und näher erklärt.

**Bemerkung 3.2.1.9 (Einbettung des Algorithmus in das Gesamtprogramm).**
Mit dem Minimax-Algorithmus lassen sich sowohl Spiele eines Bots gegen einen menschlichen Spieler, als auch Spiele zweier Bots gegeneinander realisieren. Wir möchten beide Optionen implementieren.
Ist ein Bot an der Reihe, wird der Baum anhand des aktuellen Spielzustands als Anfangszustand bis zu der vorgegebenen Tiefe aufgebaut, die Gewinnauszahlungsfunktion berechnet sich dann anhand der erbeuteten bzw. verlorenen Steine. Anschließend wird der Strategieweg des Bots mit Hilfe des Minimax-Algorithmus gewählt und der daraufhin neu erreichte Zustand wird zum Anfangszustand für den Mensch oder den zweiten Bot.

Der Baum wird dabei nicht abgespeichert, sondern in jeder Runde für den entsprechenden Zustand bis zur vorgegebenen Tiefe mit Hilfe des Minimax-Algorithmus aufgebaut und die beste Aktion gewählt. Dies begründet sich durch die enorme Speicherkapazität, die von Nöten wäre, um einen Baum mit so vielen Zuständen abzuspeichern.
Die Bots übernehmen dabei immer die MAX-Rolle, unabhängig davon, wer das Spiel gestartet hat. Dies ist möglich, da in jeder Runde ein neuer Baum aufgebaut wird. Eine andere Möglichkeit wäre, die Rollen $p_1$ und $p_2$ fest zu verteilen, so dass auch ein Bot die MIN-Rolle übernehmen kann. Wir werden allerdings in Kapitel 3.2.2 sehen, dass dieses Vorgehen unter bestimmten Voraussetzungen nicht optimal ist.
Spielt ein Bot gegen einen weiteren Bot, kann der weitere Bot sowohl mit dem Minimax-Algorithmus antworten, als auch Aktionen auf Basis anderer Algorithmen, die wir noch vorstellen werden, wählen. Dies ist nützlich um einen Vergleich zwischen den Algorithmen herzustellen.

### 3.2.2 Spielerwahl

Wie in Bemerkung 3.2.1.9 beschrieben, gibt es neben der Option, dass jede KI ihren eigenen Suchbaum maximiert, die Möglichkeit den KI- bzw. menschlichen Spielern MAX und MIN fest zuzuordnen. Bei dieser Vorgehensweise stellt sich nun die Frage, ob es bezüglich der Gewinnchance eine Rolle spielt, ob man Spieler $p_1$ (MAX) oder Gegner $p_2$ (MIN) ist. Hierfür beschreiben wir zuerst die Strategien der beiden Spieler in einer kompakteren Form, der sogenannten Normalform. Dabei orientieren wir uns an [15] und passen die Beschreibungen an unsere Definitionen an.

**Definition 3.2.2.1.** Eine *Strategiematrix* $A$ ist eine $m \times n$ - Matrix, wobei $m$ die Anzahl der möglichen Strategien $b_1, ..., b_m \in A_{s_i}, i \in \mathbb{N}$ von $p_1$ und $n$ die Anzahl der möglichen Strategien $c_1, ..., c_n \in A_{s_{i+1}}, i \in \mathbb{N}$ von $p_2$ darstellt. Die Strategiematrix $A$ gibt für die jeweiligen Aktionen die entsprechende Gewinnauszahlung an, also

$$A = \begin{pmatrix} f(g_{g_{s_i}(b_1)}(c_1)) & f(g_{g_{s_i}(b_1)}(c_2)) & \dots & f(g_{g_{s_i}(b_1)}(c_n)) \\ f(g_{g_{s_i}(b_2)}(c_1)) & f(g_{g_{s_i}(b_2)}(c_2)) & \dots & f(g_{g_{s_i}(b_2)}(c_n)) \\ \vdots & \vdots & & \vdots \\ f(g_{g_{s_i}(b_m)}(c_1)) & f(g_{g_{s_i}(b_m)}(c_2)) & \dots & f(g_{g_{s_i}(b_m)}(c_n)) \end{pmatrix}.$$

**Satz 3.2.2.2.** Sei $A = (a_{ij})$ die Strategiematrix eines Zwei-Personen-Null-Summen-Spiels und

$$v_1 = \max_i \{\min_j a_{ij}\}$$

die sogenannte *Gewinnuntergrenze*, sowie

$$v_2 = \min_j \{\max_i a_{ij}\}$$

die *Verlustsobergrenze*. Dann gilt

$$v_1 \leq v_2.$$

*Beweis.* Für eine beliebige Zeile $i \in \{1, ..., m\}$ sei das Zeilenminimum

$$d_i := \min_j a_{ij},\ j \in \{1, ..., n\}.$$

Somit gilt $d_i \leq a_{ij}$ für alle $j = 1, ..., n$. Daraus folgt

$$\begin{aligned} &\Rightarrow \max_i d_i \leq \max_i a_{ij} \text{ für alle } j = 1, ..., n \\ &\Rightarrow \max_i d_i \leq \min_j \{\max_i a_{ij}\}. \end{aligned}$$

Also gilt $v_1 \leq v_2$. □

Wir sehen also, dass es bei dem Minimax-Algorithmus im Allgemeinen tatsächlich eine Rolle spielt, ob man $p_1$ oder $p_2$ ist, denn Satz 3.2.2.2 zeigt, dass es für beide Spieler am erfolgreichsten ist, wenn sie den letzten Zug des Spieles ausführen dürfen. Spieler $p_1$ erreicht somit $v_2$, was seinem Ziel, den Minimax-Wert zu maximieren, entgegenkommt, da $v_2 \geq v_1$. Spieler $p_2$ würde als letzter ziehender Spieler $v_1$ erreichen, was für diesen ebenfalls von Vorteil wäre.
Wir haben hier nur ein Spiel betrachtet, bei welchem jeder Spieler genau einmal am Zug ist. Dieses Schema lässt sich aber induktiv auf längere Partien übertragen, indem man die $v$-Werte ebenfalls in einer Strategiematrix darstellt und den Satz erneut darauf anwendet.
Wir sehen also, dass, wenn die Tiefe des vollständigen Minimax-Baumes fest und vorher bekannt ist, berechnet werden kann, ob es von Vorteil ist, der beginnende Spieler zu sein oder nicht. Falls der Spielbaum Endknoten unterschiedlicher Tiefe besitzt, lässt sich keine Aussage treffen.

### 3.2.3 Der Alpha-Beta-Pruning-Algorithmus

Aufbauend auf dem Minimax-Algorithmus betrachten wir nun den Alpha-Beta-Pruning-Algorithmus, welcher eine bessere Variante des Minimax-Algorithmus darstellt. Die Informationen hierzu finden sich in [16].
Es handelt sich um einen sehr verbreiteten Algorithmus für Spiele, bei dem die Rechenzeit ohne einen Verlust an Informationen verbessert wird. Es wird ebenfalls ein Minimax-Wert berechnet, allerdings werden Pfade des Spielbaums vernachlässigt, wenn anhand des Minimax-Wertes schon feststellbar ist, dass diese Pfade überhaupt nicht gewählt werden. Ein folgendes Beispiel soll dies veranschaulichen.

**Beispiel 3.2.3.1.** Wir betrachten den Spielbaum aus Abbildung 3.3. Zunächst wird dieser, wie bisher beschrieben, von unten links nach oben ausgefüllt und der erste Minimax-Wert für den Zustand des Knotens $a$, hier 10, berechnet.
Der nächste Endzustand $b$ besitzt den Wert 12 und da im vorherigen Zug der $p_1$-Spieler an der Reihe ist, wird der Minimax-Wert von Knoten $c$ auf jeden Fall größer oder gleich 12 sein. Da vor diesem aber $p_2$ am Zug ist und 10, der Wert von $a$ kleiner als 12 ist, wird der Weg zu $c$ gar nicht erst eingeschlagen. Wir ersparen uns hier also die Berechnung des Wertes von Knoten $d$.
Als Nächstes ist der Pfad zu Knoten $e$ von Interesse. Auch hier wird der Minimax-Wert für $e$, also 9, berechnet. Da im Knoten $f$ der $p_2$-Gegner am Zug ist, und den Minimax-Wert für seinen Zustand minimieren möchte, kann der Wert von $f$ somit nicht größer als 9 sein. Dies reicht aber nicht aus, um den $p_1$-Spieler von Knoten $g$ auf diesen Pfad zu lenken, da er den anderen Pfad zu dem Knoten mit Minimax-Wert 10 bevorzugen wird. Somit ist der Pfad ab Knoten $h$ vernachlässigbar.

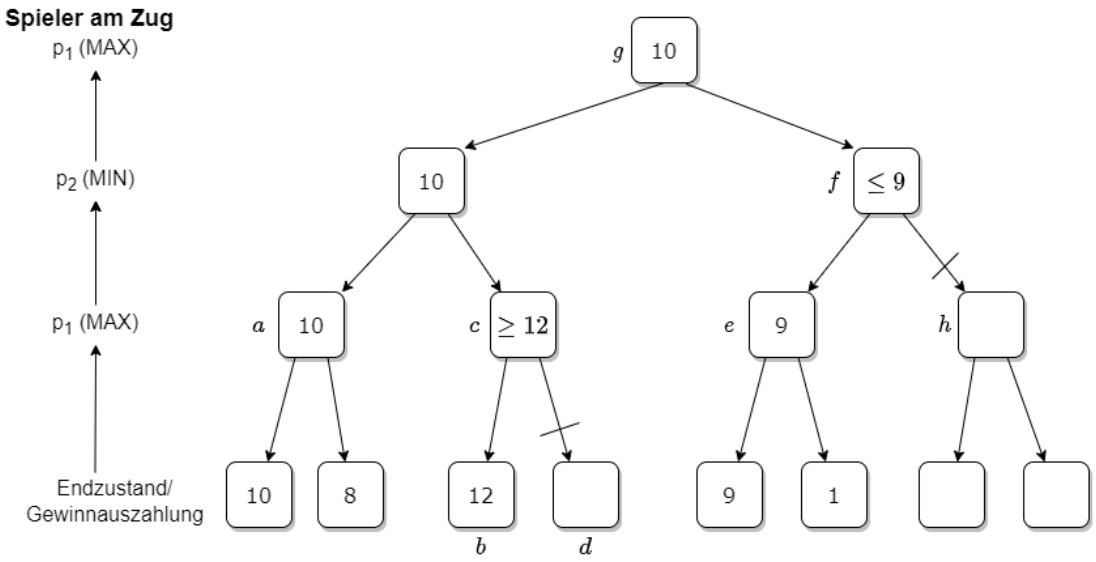


Abbildung 3.3: Spielbaum mit drei Zügen und Abtrennen der vernachlässigbaren Pfade zum Veranschaulichen des Alpha-Beta-Pruning Algorithmus

**Algorithmus 3.2.3.2.** Wir betrachten den Algorithmus aus [1] und passen ihn auf unsere Definitionen an.
Die Variable $\alpha$ soll die größten bisher erreichten Minimax-Werte abspeichern, die Variable $\beta$ die kleinsten. Sobald $\alpha \geq \beta$, lassen sich Wege ausschließen und es findet die in Beispiel 3.2.3.1 beschriebene Abtrennung (engl.: pruning) des Pfades statt, indem die weitere Berechnung gestoppt wird. Die Anfangswerte für $\alpha$ und $\beta$ lassen sich beliebig unterhalb bzw. oberhalb der möglichen Werte, welche die Gewinnfunktion annehmen kann, initialisieren. Wir setzen $\alpha := -\infty$ und $\beta := +\infty$, um sicher zu gehen, dass diese Bedingung erfüllt ist.
Mit der Rechenzeit, welche die hier abgeschnittenen Pfade im Minimax-Algorithmus benötigt haben, lassen sich stattdessen weiter bestehende Pfade tiefer erkunden.

**Bemerkung 3.2.3.3 (Umsetzung im Programm).** Obwohl der Baum nun weniger komplex ist und nicht so viele Zustände wie beim Minimax-Algorithmus abgespeichert werden müssen, ist es immer noch einfacher, den Baum jede Runde neu aufzubauen. Dementsprechend ist die Einbettung des Alpha-Beta-Pruning Algorithmus gleich wie die des Minimax-Algorithmus aus Bemerkung 3.2.1.9.

**Algorithm** Der Alpha-Beta-Pruning-Algorithmus

```
1: function MAXIMUM(s, α, β)
2:     if s ist ein Endzustand then
3:         return f(s)
4:     end if
5:     value ← −∞;
6:     for a ∈ A_s do
7:         value ← max(value, MINIMUM(g_s(a), α, β));
8:         α ← max(α, value);
9:         if α ≥ β then
10:            break
11:        end if
12:    end for
13:    return value
14: end function
15:
16: function MINIMUM(s, α, β)
17:    if s ist ein Endzustand then
18:        return f(s)
19:    end if
20:    value ← +∞;
21:    for a ∈ A_s do
22:        value ← min(value, MAXIMUM(g_s(a), α, β));
23:        β ← min(β, value);
24:        if β ≤ α then
25:            break
26:        end if
27:    end for
28:    return value
29: end function
```

**Bemerkung 3.2.3.4.** Um einen Vergleich zum Minimax-Algorithmus zu erhalten, sehen wir uns den *Verzweigungsfaktor* oder *branching factor* an. Dieser ist eine Angabe darüber, wie viele Nachfolger durchschnittlich für einen Knoten existieren, bzw. wie viele Pfade im Durchschnitt von jedem Knoten aus erkundet werden. Er lässt sich wie in [16] definieren:
Für einen Suchbaum mit Tiefe $t \in \mathbb{N}$ und Grad $n \in \mathbb{N}$, sei der Verzweigungsfaktor

$$\mathbb{B} := \lim_{t \to \infty} (N_{n,t})^{\frac{1}{t}}.$$

Hierbei gibt $N_{n,t}$ eine mittlere Anzahl von Endzuständen an, die während der Suche über einen Baum mit Tiefe $t$ und Grad $n$ betrachtet werden. Für einen vollständig aufgebauten Baum dieser Art, bei dem alle Endzustände für die Suche von Bedeutung sind, gilt $N_{n,t} = n^t$ für alle $t \in \mathbb{N}$ und damit $\mathbb{B} = n$.
Ein Resultat, das in [16] ausführlich bewiesen wird, sagt, dass der Verzweigungsfaktor für die Baumsuche mit Hilfe des Alpha-Beta-Pruning Algorithmus durch

$$\mathbb{B}_{\alpha-\beta} = \zeta_n/(1-\zeta_n)$$

angegeben werden kann. Dabei ist $\zeta_n$ eine positive Nullstelle des Polynoms

$$x^n + x - 1 = 0.$$

Dadurch besitzt der Alpha-Beta-Pruning Algorithmus nach [16] eine asymptotische Optimalität hinsichtlich der Komplexität gegenüber allen Suchbaumalgorithmen für Spiele.
Wir gehen davon aus, dass ein Spieler in jeder Runde des Spiels Oware Abapa von allen 6 Feldern aus ziehen kann. Da der Suchbaum des Minimax-Algorithmus immer vollständig aufgebaut wird, liegt hier also ein Verzweigungsfaktor von 6 vor.
Für den Verzweigungsfaktor des Alpha-Beta-Pruning Algorithmus berechnen wir nun die positive Nullstelle $\zeta_6$ des Polynoms $x^6+x-1$. Diese ist ungefähr $\zeta_6 = 0.778$. Also liegt hier ein Verzweigunsfaktor von $\mathbb{B}_{\alpha-\beta} = 0,778/(1-0,788) \approx 3,67$ vor, was etwas mehr als die Hälfte des Verzweigungsfaktors des Minimax-Algorithmus beträgt.

## 3.3 Monte Carlo Tree Search

Grundlegend für den Monte Carlo Tree Search Algorithmus ist das Markov Entscheidungsproblem (MEP), das wir kurz einführen möchten. Wir orientieren uns dabei an [4], [8] und [10]. In der Theorie von MEPs wird ein Spieler auch *Agent* genannt, welcher abhängig von seinem aktuellen Zustand, wie bei unserem Beispiel die Situation auf dem Spielbrett, verschiedene Aktionen ausführen kann. Wie im vorigen Kapitel beeinflussen diese Aktionen dann die nächsten Zustände. Allerdings werden nun Wahrscheinlichkeitsfunktionen mit ins Spiel gebracht, sodass für die Bestimmung einer guten Strategie keine eindeutige Zuordnung der Aktionen zu den Zuständen mehr erforderlich ist.

**Definition 3.3.1** Sei wie in Definition 3.2.1.4 $S$ die Menge aller möglichen Zustände und $A_s$ die Menge aller möglichen Aktionen im Zustand $s \in S$. Für die Endzustände gilt $A_s = \varnothing$. Weiter sei nun

(i) $f : S \times A_s \to \mathbb{Z}$ die Gewinnauszahlungsfunktion oder auch *reward*-Funktion, welche nun nicht mehr nur vom Zustand $s \in S$ abhängt. Sie bewertet nun das Ausführen einer Aktion $a \in A$ im Zustand $s \in S$.

(ii) $\delta : S \times A_s \times S \to [0, 1]$, wobei $\delta(s, a, s') := \delta(s'|a, s)$, das *Markov Transaktionsmodell*, welches die Wahrscheinlichkeit angibt zu Zustand $s' \in S$ zu gelangen, vorausgesetzt man führt Aktion $a$ in Zustand $s$ aus.

(iii) eine Startwertverteilung $\mu : S \to [0, 1]$, die jedem Zustand $s \in S$ eine Wahrscheinlichkeit zuordnet, mit der das MEP in diesem Zustand startet.

**Definiton 3.3.2 (Markov Entscheidungsproblem)** Das *Markov Entscheidungsproblem (MEP)* ist ein Tupel $(S, A_s, \delta, f, \mu)$.

Ziel des MEP ist es, den besten Gewinn zu erzielen und eine Strategie, eine sogenannte *policy*

$$\pi : S \to A_s$$

zu finden, welche jedem Zustand eine optimale Aktion zuweist, sodass dieses Ziel erfüllt ist.

**Bemerkung 3.3.3** Nicht immer ist es sinnvoll eine eindeutige Strategie zu wählen. Wie in [15] beschrieben, gibt es hierfür ein alternatives Konzept der *gemischten*

*Strategie* (engl.: *mixed strategy*). Dies stellt eine Wahrscheinlichkeitsverteilung dar, welche jeder möglichen Strategie des Spielers eine Wahrscheinlichkeit zuordnet, mit der die Strategie auszuführen ist. Im Allgemeinen lässt sich $\pi$ also auch als eine Funktion

$$\pi : S \times A_s \to [0,1] \text{ wobei } \pi(s,a) := \pi(a|s)$$

schreiben, welche die Wahrscheinlichkeit angibt im Zustand $s \in S$ die Aktion $a \in A_s$ auszuführen. Da es sich um eine Wahrscheinlichkeitsverteilung handelt, muss gelten:

$$\sum_{a \in A_s} \pi(a|s) = 1 \text{ und } \pi(a|s) \geq 0$$

**Definition 3.3.4** Die *beste* oder *optimale Strategie* $\pi^*$, ist die Strategie $\pi$, die den *erwarteten kumulativen Gewinn*

$$J(\pi) := \mathbb{E}_{a \sim \pi(\cdot|s), s' \sim \delta(\cdot|s,a)} \left[ \sum_{t=0}^{\infty} \gamma^t f(s_t, a_t) \right]$$

maximiert. Hierbei sind $s, a$ und $s'$ allgemeine Bezeichnungen für den Zustand, eine Aktion und den daraus resultierenden Folgezustand für alle Zeitpunkte $t \in \mathbb{N}$. Der Anfangszustand $s_0$ wird anhand der Startwertverteilung $\mu$ zufällig bestimmt. Weiter ist $\gamma \in [0,1]$ ein Diskontierungsfaktor, welcher den Einfluss von später stattfindenden Zügen dämpfen soll und $\mathbb{E}_{\mathcal{D}}(X)$ bezeichnet den Erwartungswert einer $\mathcal{D}$-verteilten Zufallsvariable $X$.

Wir möchten nun zeigen, dass dieses Optimierungsproblem eine eindeutige Lösung besitzt. Dafür betrachten wir zunächst folgende Definition.

**Definition 3.3.5.** Der erwartete kumulative Gewinn, gegeben man befindet sich zu Beginn in Zustand $s$, ist die *state-value*-Funktion

$$V^{\pi}(s) := \mathbb{E}_{a \sim \pi(\cdot|s), s' \sim \delta(\cdot|s,a)} \left[ \sum_{t=0}^{\infty} \gamma^t f(s_t, a_t) | s_0 = s \right].$$

Diese kann in Form einer Bellman-Gleichung geschrieben werden, denn

$$
\begin{aligned}
V^{\pi}(s) &= \mathbb{E}_{\underbrace{a \sim \pi(\cdot|s),\, s' \sim \delta(\cdot|s,a)}_{:=D}} \left[ \sum_{t=0}^{\infty} \gamma^t f(s_t, a_t) | s_0 = s \right] \\
&= \mathbb{E}_D(f(s_0, a_0) + \gamma f(s_1, a_1) + \gamma^2 f(s_2, a_2) + \cdots | s_0 = s) \\
&= \mathbb{E}_D(f(s_0, a_0) + \gamma \sum_{t=0}^{\infty} \gamma^t f(s_{t+1}, a_{t+1}) | s_0 = s) \\
&= \mathbb{E}_{a_0 \sim \pi(\cdot|s_0), s_1 \sim \delta(\cdot|s_0,a_0)} \left[ f(s_0, a_0) + \gamma \cdot \mathbb{E}_D(\sum_{t=0}^{\infty} \gamma^t f(s_{t+1}, a_{t+1})) | s_0 = s \right] \\
&= \mathbb{E}_{a_0 \sim \pi(\cdot|s_0), s_1 \sim \delta(\cdot|s_0,a_0)}(f(s_0, a_0) + \gamma \cdot \mathbb{E}_D(V^{\pi}(s_1)) | s_0 = s) \\
&= \mathbb{E}_D(f(s, a) + \gamma V^{\pi}(s')) \\
&= \mathbb{E}_D(f(s, a)) + \gamma \cdot \mathbb{E}_D(V^{\pi}(s')) \\
&= f^{\pi}(s, a) + \gamma \int_s \int_a \delta(s'|s, a) \pi(a|s) V^{\pi}(s')
\end{aligned}
\tag{3.3}
$$

Den letzten Gleichungsschritt erhält man, wenn die Definition des Erwartungswertes einsetzt. Dabei ist $f^{\pi}(a, s) = \int_a \pi(a|s) \cdot f(s, a)$.

**Satz 3.3.6** Das Maximierungsproblem der Bellmann-Gleichung (3.3) hat eine eindeutige Lösung.

*Beweis.* Der Einfachheit halber betrachten wir die Bellman-Gleichung in Form einer Summe, der Beweis für die allgemeine Gleichung in Definition 3.3.5 lässt sich dann darauf zurückführen. Der Beweis stammt aus [18]. Wir schreiben

$$
V^{\pi}(s) = f^{\pi}(s, a) + \gamma \sum_{\substack{s \in S, \\ a \in A_s}} \delta(s'|a, s) \pi(a|s) V^{\pi}(s') \tag{3.4}
$$

und fassen die Wahrscheinlichkeiten als $p_0(a, s, s') := \delta(s'|a, s)\pi(a|s) \in [0, 1]$ zusammen. Betrachtet man nun (3.4) als Abbildung

$$
T(V^{\pi}(s)) := f^{\pi} + \gamma \cdot \sum p_0 V^{\pi}(s'),
$$

kann man zeigen, dass dies eine Kontraktion ist:
Für jede Aktion $a \in A_s$ und Zustand $s \in S$ gibt es ein $\epsilon_0$, so dass $\gamma \cdot \|p_0\|_1 \leq \epsilon_0 < 1$. Hierbei ist $\| \cdot \|_1$ die $l_1$-Norm, das heißt $\|x\|_1 = \sum_{i=1}^{n} |x_i|$ für einen Vektor $x \in \mathbb{R}^n$.

Dann gilt für beliebige state-value-Funktionen $V_1$ und $V_2$

$$T(V_1) - T(V_2) = \gamma \cdot \sum_{s'} p_0(s')(V_1(s') - V_2(s')) \leq \gamma \cdot \sum_{s'} p_0(s') \|V_1 - V_2\|_\infty,$$

wobei $\|\cdot\|_\infty$ die $l_\infty$-Norm ist, also $\|x\|_\infty = \max_{i=1}^n |x_i|$ für einen Vektor $x \in \mathbb{R}^n$. Somit gilt also

$$T(V_1) - T(V_2) \leq \epsilon_0 \|V_1 - V_2\|_\infty,$$

das heißt $T$ ist eine Kontraktion. Dann folgt mit dem Fixpunktsatz von Banach die Aussage. □

### 3.3.1 Monte Carlo Tree Search (Algorithmus)

Monte Carlo Tree Search (MCTS) ist ein Algorithmus um MEPs zu lösen bzw. eine approximative Lösung bereitzustellen. MCTS bildet wie bei Adversarial Search einen Spielbaum, bei dem vielversprechende Pfade weiter expandiert werden als weniger erfolgreiche. Das Ziel ist es, einen Ausgleich zwischen der Ausschöpfung (*Exploitation*) bestehender Pfade, also die beste Strategie auf diesen Pfaden zu finden, und der weiteren Erkundung (*Exploration*) bisher unbekannter Pfade zu schaffen. Bei diesem Algorithmus werden vier Schritte wiederholt:

(i) *Selektion*: Im ersten Schritt geht der Algorithmus von der Wurzel des Baumes mit Hilfe einer Selektionsstrategie durch den Baum, bis ein Knoten erreicht wird, für den noch nicht alle Nachfolger existieren und welcher am ehesten expandiert werden sollte.

(ii) *Expansion*: Im nächsten Schritt wird an dieser Stelle für einen der in (i) erwähnten Nachfolger ein Knoten im Baum angelegt.

(iii) *Simulation*: Von diesem neuen Knoten aus werden zufällige Spielzüge ausgeführt, ein sogenanntes *rollout* findet statt. Dies geschieht, bis ein Spiel beendet oder die maximale Suchtiefe erreicht ist.

(iv) *Backpropagation*: Abschließend wird die Bewertung des Spielendes durch alle Knoten des selektierten Pfades zurückgegeben und die Werte dieser Knoten neu angepasst.

Eine Übersicht der vorgestellten Schritte findet sich in Abbildung 3.4.

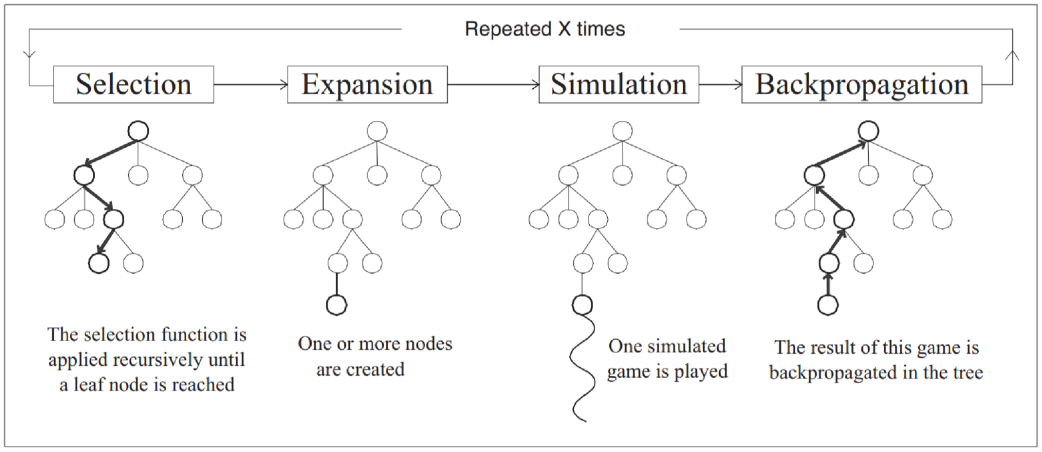


Abbildung 3.4: Aufbau von Monte Carlo Tree Search, Quelle: [6]

Wie in [2] beschrieben, entwickelt der MCTS-Algorithmus also zwei Strategien:

(i) *tree policy*: Beinhaltet die Schritte *Selektion* und *Expansion*

(ii) *default policy*: Beinhaltet den Schritt *Simulation*

Mit diesen Begrifflichkeiten lässt sich ein erster einfacher Algorithmus aufstellen:

**Algorithmus 3.3.1.1.** Dem Algorithmus wird der Startzustand $s_0$ für den Wurzelknoten $v_0$ übergeben und gibt die beste Aktion $a$ zurück. Die Funktion TREE POLICY gibt den Knoten $v_i$ zurück, der neu dazugekommen ist und für den die Simulation nun stattfinden soll. Die Variable $\Delta$ stellt die Bewertung des Simulationsschrittes dar.

**Algorithm** Der MCTS-Algorithmus

```
1: function MCTS(s_0)
2:     while Abbruchkriterium ist nicht erreicht do
3:         v_i ← TREE POLICY(v_0);
4:         Δ ← DEFAULT POLICY(v_i);
5:         BACKPROPAGATE(Δ, v_i)
6:     end while
7:     return π*(s_0)
8: end function
```

Der Nachfolgerknoten, den die *tree policy* festlegt, kann anhand von verschiedenen

Kriterien gewählt werden. Einige ausgewählte sind zum Beispiel, wie in [2] aufgelistet, der Knoten, dessen Zustand den höchsten Gewinn ergibt, der Knoten, der am öftesten durchlaufen wurde, eine Mischung daraus oder der Knoten, der eine obere Konfidenzgrenze maximiert, wie es in Kapitel 3.3.2 beschrieben wird.

### 3.3.2 Upper Confidence Bounds for Trees (UCT)

Die bekannteste Methode für die *tree policy* stellt der *UCT-Algorithmus* dar. Für jeden Knoten im Spielbaum, den man durchläuft, werden UCT-Werte für die Nachfolger berechnet, welche das Dilemma zwischen *Exploitation* und *Exploration* lösen sollen.
Der Baum wird anhand der höchsten UCT-Werte durchgegangen, bis ein Knoten gefunden wird, der noch nicht vollständig expandiert ist. Das bedeutet, dass für diesen Knoten noch nicht alle möglichen Nachfolger angelegt wurden. Der Nachfolger, der den höchsten UCT-Wert erhalten würde, wird im Expansionsschritt gebildet und ist der Knoten, welcher der *default policy* übergeben wird. Der UCT-Wert lässt sich wie in [2] definieren.

**Definition 3.3.2.1.** Sei $v$ der aktuelle Knoten, $n \in \mathbb{N}$ die Anzahl, wie oft dieser Knoten bisher durchlaufen wurde und $n_{v'} \in \mathbb{N}$ die Anzahl der Durchläufe für die Nachfolgerknoten $v'$. Sei weiterhin $c > 0$ eine Konstante, die den Grad der *Exploration* steuern soll und $\bar{X}_{v'} \in [0, 1]$ die Gewinnrate für den Knoten $v'$. Dann ist der *UCT*-Wert für Knoten $v'$ gegeben durch

$$UCT_{v'} = \begin{cases} \bar{X}_{v'} + 2c\sqrt{\frac{2 \ln n}{n_{v'}}}, & n_{v'} \neq 0 \\ +\infty, & n_{v'} = 0 \end{cases}.$$

**Algorithmus 3.3.2.2.** Auf der folgenden Seite findet sich der Pseudocode für den UCT-Algorithmus. Hierbei soll $\frac{Q(v')}{N(v')}$ eine Schätzung für $\bar{X}_{v'}$ sein. $Q(v')$ ist die Anzahl der bisherigen Gewinne für den Knoten $v'$ und $N(v')$ die Anzahl der bisher gespielten Spiele für den Knoten $v'$. Diese müssen in der BACKPROPAGATION-Funktion aktualisiert werden.
Die beste Aktion $\pi^*(s_0)$ lässt sich dann über den UCT-Wert berechnen, indem man $c$ auf Null setzt. Das bedeutet, der MCTS-Algorithmus gibt am Schluss BEST CHILD$(v_0, 0)$ zurück.

**Algorithm** Der UTC-Algorithmus

**function** TREE POLICY($v$)
  **while** $v$ ist kein Endknoten **do**
    **if** $v$ ist noch nicht vollständig expandiert **then**
      **return** EXPAND($v$);
    **else**
      $v \leftarrow$ BEST CHILD($v, c$);
    **end if**
  **end while**
  **return** $v$
**end function**

**function** EXPAND($v$)
  Wähle eine noch nicht verwendete Aktion $a \in A_s$ für den Zustand $s$ des Knotens $v$ aus.
  $v' \leftarrow$ Nachfolgerknoten von $v$ mit Zustand $g_s(a)$;
  **return** $v'$
**end function**

**function** BEST CHILD($v, c$)
  **return** $\underset{v' \text{ Nachfolger von } v}{\arg\max} \frac{Q(v')}{N(v')} + c\sqrt{\frac{2 \ln N(v)}{N(v')}}$
**end function**

**function** BACKPROPAGATE($v, \Delta$)
  **while** $v \neq 0$ **do**
    $N(v) \leftarrow N(v) + 1$;
    $Q(v) \leftarrow Q(v) + \Delta$;
    $v \leftarrow$ Elternknoten von $v$;
  **end while**
**end function**

**Beispiel 3.3.2.3.** Um den Algorithmus besser nachvollziehen zu können, möchten wir ein konkretes Zahlenbeispiel für die jeweiligen Schritte betrachten. Der Einfachheit halber gehen wir davon aus, dass in jedem Zustand nur zwei Aktionen möglich sind. Dieses Beispiel ist in Abbildung 3.5 schematisch dargestellt.
Wir beginnen mit dem Anfangszustand $s_0$. Da dieser zu Beginn noch keine Nachfolger besitzt, ist der Schritt der Selektion überflüssig. Wir wählen eine beliebige Aktion aus und bilden so den Nachfolgerknoten, für welchen dann der Simulationsschritt einsetzt. In diesem Beispiel wurde das Spiel gewonnen. Dies wird im Backpropagationschritt dann in allen durchlaufenen Knoten festgehalten. Hier wird also in beiden Knoten „1/1“ gespeichert, was bedeutet, dass in bisher einem gespielten Spiel genau ein Gewinn erzielt wurde.
Anschließend findet in der zweiten Reihe der Abbildung die Selektion des nächsten Knotens mit Hilfe des UCT-Wertes statt, der wie in Definition 3.3.2.1 berechnet wird. Da in Zustand $s_0$ noch eine weitere Aktion möglich ist und diese bisher noch nicht betrachtet wurde, hat der Knoten, zu dem diese Aktion führt, UCT-Wert $+\infty$. Dies ist größer als der UCT-Wert des Knotens der vorherigen Aktion (hier 1), sodass der neue Knoten im Expansionsschritt angelegt wird und die Simulation von diesem Knoten aus beginnt.
Nun wurde das Spiel verloren, was bedeutet, dass in dem Knoten „0/1“ gespeichert wird. Da der Knoten von $s_0$ ebenfalls durchlaufen wurde, ändert sich der darin gespeicherte Wert von „1/1“ zu „1/2“, da kein Gewinn hinzugekommen ist, aber ein weiteres Spiel gespielt wurde.
Im Selektionsschritt der darauf folgenden Zeile werden die UCT-Werte neu berechnet und mit Hilfe dieser Werte wird wieder ein Weg im Baum gewählt, bis ein Endknoten gefunden und ein neuer Nachfolger angelegt werden kann. Die beschriebenen Prozeduren wiederholen sich, bis ein vorab festgelegtes Ende erreicht wird. In der Praxis ist dies eine vorgegebene Anzahl an Zyklen, die der Algorithmus durchläuft.

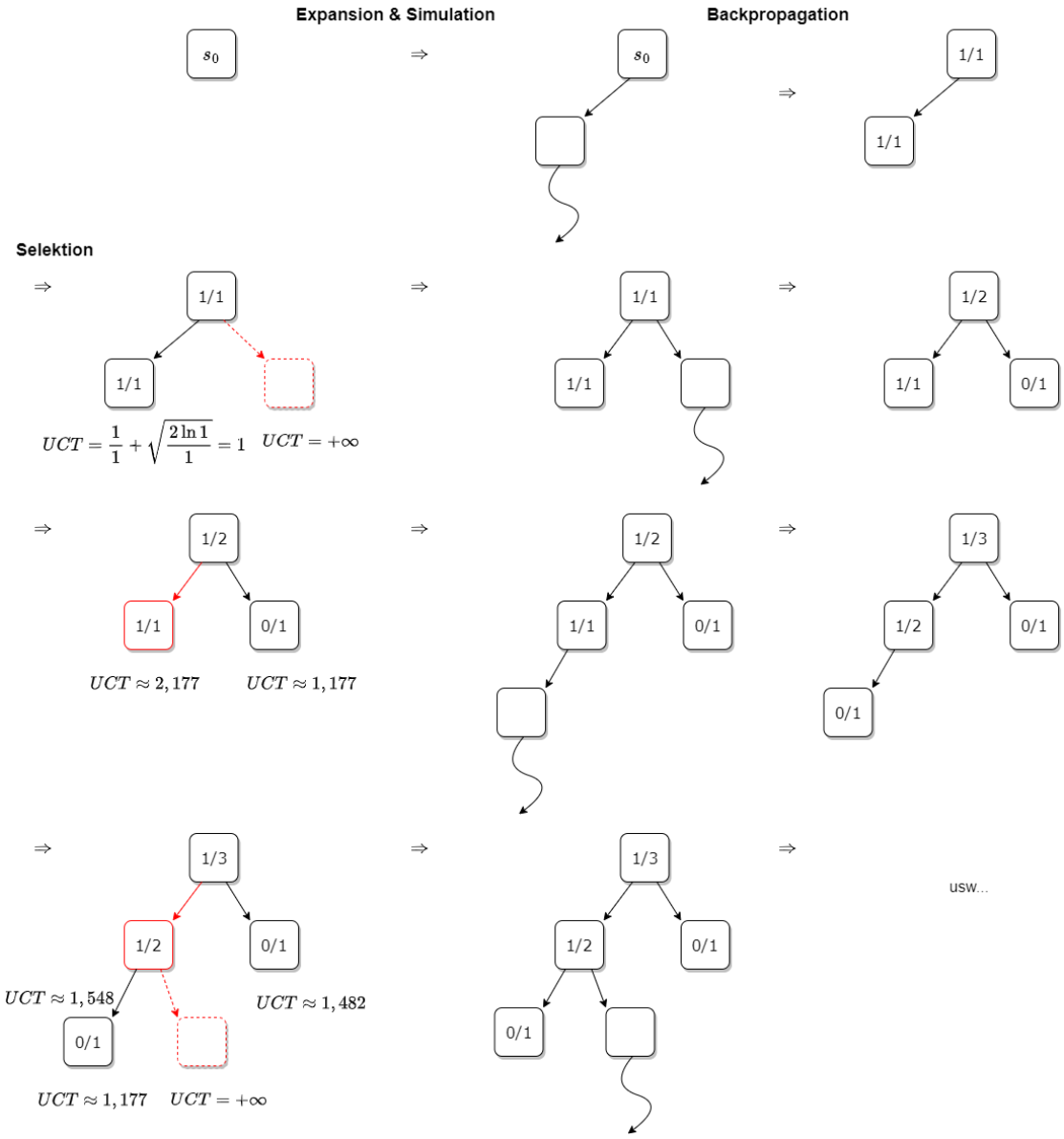


Abbildung 3.5: Aufbau eines Spielbaumes mit Hilfe des UCT-Algorithmus sowie die Darstellung der einzelnen Schritte Selektion, Expansion, Simulation und Backpropagation

**Bemerkung 3.3.2.4.** Bisher haben wir den Monte Carlo Tree Search Algorithmus als eine Methode betrachtet, die es nur einem Agenten ermöglicht eine optimale Strategie zu finden. Da es sich bei Oware Abapa aber um ein Zwei-Personen-Spiel handelt, muss der Algorithmus etwas verändert werden. Hierfür orientieren wir uns wieder an [2]. Ähnlich wie bei Adversarial Search repräsentiert abwechselnd eine

Ebene des Baumes die Handlungsmöglichkeiten des einen Spielers, die darauffolgende Ebene die des Gegenspielers. Gewinnt ein Spieler das Spiel, werden nun in der BACKPROPAGATION-Funktion nicht alle Gewinne in den Knoten des Pfades erhöht, sondern die des Gegenspielers verringert oder unverändert gelassen. Der folgende Algorithmus zeigt die neue BACKPROPAGATION-Funktion.

**Algorithm** Der MCTS-Algorithmus für ein Zwei-Personen-Spiel

```
1: function BACKPROPAGATE(v, Δ)
2:     while v ≠ 0 do
3:         N(v) ← N(v) + 1;
4:         Q(v) ← Q(v) + Δ;
5:         Δ ← −Δ;
6:         v ← Elterknoten von v;
7:     end while
8: end function
```

**Bemerkung 3.3.2.5 (Konvergenz zu Minimax)** Es lässt sich beweisen, dass die Wahrscheinlichkeit eine Strategie zu wählen, die nicht der optimalen Strategie $\pi^*$ entspricht, gegen Null konvergiert. Das bedeutet also, dass wenn genug Zeit und Speicher vorhanden ist und somit ausreichend viele Durchläufe stattfinden, der Suchbaum des UCT-Algorithmus gegen den optimalen Baum des Minimax-Algorithmus konvergiert. Diese Informationen finden sich ebenfalls in [2].
Obwohl der Algorithmus gegen Minimax konvergiert, ist er aufgrund der zufälligen Simulationsschritte nicht deterministisch.

**Bemerkung 3.3.2.6 (Umsetzung im Programm).** Wie beim Minimax-Algorithmus implementieren wir eine KI mit dem MCTS-Algorithmus, die sowohl gegen einen Menschen als auch gegen eine weitere KI spielen kann.
Der Suchbaum wird ebenfalls in jeder Runde mit dem aktuellen Spielzustand als Zustand für den Wurzelknoten neu aufgebaut und nicht gespeichert, da eine solche Vielzahl an Bäumen eine zu hohe Speicherkapazität benötigen würde. Bei dem MCTS-Algorithmus wird keine Suchtiefe vorgegeben, sondern eine Zyklenanzahl, die angibt, wie oft der Algorithmus durchläuft. Als beste Aktion geben wir dann die Aktion zurück, welche zu dem Kindknoten der Wurzel führt, der den größten Quotienten $Q/N$ besitzt.
Im Gegensatz zum Minimax-Algorithmus, wird hier der Baum nicht rekursiv durchsucht, sondern aktiv über eine Liste aufgebaut.

## 3.4 Reinforcement Learning

Wir interessieren uns nun für verschiedene Verfahren des Reinforcement Learnings (RL) und stellen Überlegungen auf, wie sich diese für Oware Abapa anwenden lassen. In diesem Kapitel orientieren wir uns an [10].
Die Grundlage dieser Lernverfahren bildet die Interaktion eines Agenten mit der Umgebung. Es handelt ein Agent in Zustand $s_t$ mit einer bestimmten Aktion $a_t$ in einer Umgebung zur Zeit $t$. Dieses Handeln wird dann über eine Belohnungsfunktion $r_t := f(s_t, a_t)$ bewertet. Mit dieser Belohnung, auch *Reward* genannt, lernt der Agent, ob seine Aktion nützlich war und entwickelt daraus eine Strategie. Das Ausführen der Aktion $a_t$ in Zustand $s_t$ führt dann zum neuen Zustand $s_{t+1}$. Dieser Kreislauf ist in Abbildung 3.6 dargestellt. Das Ziel ist es, die bestmögliche Strategie $\pi^*$ zu finden, mit welcher der Agent auf jede mögliche Situation seiner Umwelt optimal reagieren kann.

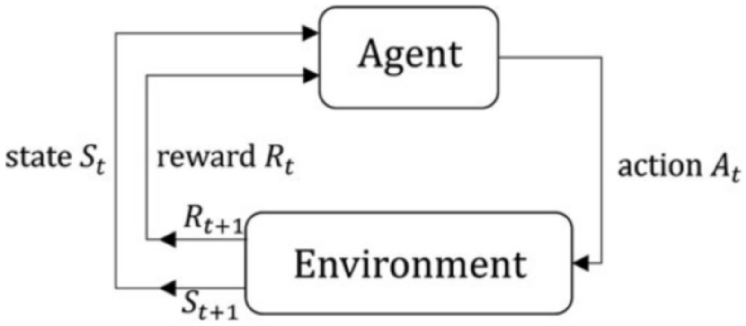


Abbildung 3.6: Grundlage des Reinforcement Learning: Interaktion eines Agenten mit der Umwelt, Quelle: [10].

Ähnlich wie in Kapitel 3.3.1 ist dabei ein Exploitation-Exploration-Dilemma zu lösen. Der Agent möchte sein Wissen über bestimmte Aktionen ausnützen (*Exploitation*) und sein Handeln hierbei verbessern, um einen möglichst guten Reward zu erzielen. Da es ungewiss ist, ob eine noch nicht erkundete Vorgehensweise einen besseren Reward erzielen würde, wird der Agent hauptsächlich auf den Exploitation-Weg setzen, man nennt dieses Handhaben *greedy*-Strategie. Hierbei werden womöglich nur kleine Verbesserungen erzielt, es schützt aber vor Verlusten.
Da die Erkundung neuer Strategien (*Exploration*) dennoch wichtig ist, um eine womöglich wesentlich lukrativere Strategie nicht zu übersehen, existieren verschiedene Verfahren um einen angebrachten Mittelweg zu finden. Eines davon ist die in

Kapitel 3.3.2 vorgestellte Berechnung des UCT-Wertes.
Ein weiteres Verfahren ist die $\epsilon$-*greedy* Methode, welche mit vorgegebener Wahrscheinlichkeit $\epsilon \geq 0$ die Exploration in Betracht zieht, ansonsten den Exploitation-Weg bevorzugt. Der Nachteil ist hier allerdings, dass alle bisher nicht erkundeten Strategien für die Exploration zur Wahl stehen und keine Gewichtung, wie bei der Methode über den UCT-Wert, stattfindet.

Die meisten RL-Algorithmen beruhen auf der Lösung des Markov Entscheidungsproblems, wie es in Kapitel 3.3 beschrieben wurde. Das Ziel dieser Algorithmen ist es den erwarteten kumulativen Gewinn

$$J(\pi) = \mathbb{E}_{a\sim\pi(\cdot|s),s'\sim\delta(\cdot|s,a)}\left[\sum_{t=0}^{\infty}\gamma^t f(s_t,a_t)\right]$$

zu maximieren, indem die optimale Strategie $\pi^*$ gesucht wird. Wir verwenden in Zukunft nur noch $\mathbb{E}_{a\sim\pi(\cdot|s)}(\cdot)$, da das Transaktionsmodell $\delta$ ebenfalls von $\pi$ abhängt und es sich bei Oware Abapa um ein Spiel handelt, bei dem sich nach einer gewählten Aktion ein eindeutiger Folgezustand ergibt. Wir benötigen noch einige Ergänzungen, die wir nun vorstellen möchten. Die Theorie dazu entnehmen wir aus [26] und [10].

**Definition 3.4.1.** Wie in Kapitel 3.3 sei für eine beliebige Strategie $\pi$ der erwartete Gewinn, gegeben man befindet sich in Zustand $s$, die *state-value*-Funktion

$$V^{\pi}(s) = E_{a_t\sim\pi(\cdot|s_t)}(\sum_{t=0}^{\infty}\gamma^t f(s_t,a_t)|s_0 = s).$$

Weiter definieren wir für eine beliebige Strategie $\pi$ die *action-value*-Funktion

$$Q^{\pi}(s,a) := E_{a_t\sim\pi(\cdot|s_t)}(\sum_{t=0}^{\infty}\gamma^t f(s_t,a_t)|s_0 = s, a_0 = a),$$

welche nun zusätzlich von der Aktion $a$ abhängt. Der Zusammenhang der beiden Funktionen kann über $V^{\pi}(s) = E_{a\sim\pi}(Q^{\pi}(s,a))$ beschrieben werden.

### 3.4.1 Modellbasierte und modellfreie Algorithmen

In modellbasierten Algorithmen wird davon ausgegangen, dass über die Elemente des Tupels $(S, A_s, \delta, f, \mu)$ des MEPs Kenntnis herrscht. Oft sind die Gewinnfunktion $f$ und die Transaktionsfunktion $\delta$ allerdings unbekannt. Einen Ausweg bietet hierbei die Option, durch genügend Beobachtungen von $(s, a, f(a, s), s')$, wobei $s$ einen Zustand, $a$ eine Aktion und $s'$ den daraus resultierenden Folgezustand bezeichnet, die unbekannten Funktionen $f$ und $\delta$ vorherzusagen. Man modelliert hier also die Umgebung des Agenten. Dieses Verfahren wird beispielsweise bei dem MCTS-Algorithmus aus Kapitel 3.3.1 angewendet.
Da eine Modellierung der Umgebung sehr schwierig sein kann und wir nicht alle Parameter des MEPs angewandt auf unser Spiel Oware Abapa kennen, beschäftigen wir uns mit den modellfreien Algorithmen.
Im Gegensatz zu den modellbasierten Algorithmen ist hier kein Wissen über alle Umgebungsparameter von Nöten. Hierbei wird die Umgebung auch nicht modelliert, sondern die optimale Strategie direkt berechnet. Dadurch lassen sich die Algorithmen leichter implementieren. Wir betrachten hier das *Q-Learning* und die *policy gradient*-Methode. Weitere modellfreie Algorithmen, die wir nicht betrachten, sind zum Beispiel Temporal Difference Learning und SARSA. Die Theorie zu diesen Methoden findet sich in [10].

### 3.4.2 Q-Learning

In diesem Abschnitt orientieren wir uns wieder an [10]. Das Ziel des Q-Learnings ist es, anhand von approximierten Qualitätswerten (*Q-Werte*) $Q(s, a)$ die beste Strategie $\pi^*$ zu finden, sodass $Q^{\pi^*}$ maximal wird. Für jeden Zustand und jede Aktion wird ein solcher Q-Wert angelegt und während dem Lernprozess fortlaufend angepasst. Diese Werte lassen sich in einer Q-Tabelle speichern, wobei die Zeilen die Zustände und die Spalten die Aktionen der Q-Werte festlegen. Nach ausreichend vergangener Trainingszeit lässt sich dann anhand der Q-Tabelle eine optimale Strategie ableiten, indem für jeden Zustand die Aktion ausgewählt wird, für die der Q-Wert in der Tabelle am höchsten ist.
Q-Learning ist eine *off-policy*-Methode, das bedeutet, dass keine Strategie vorliegen muss, die evaluiert oder verbessert werden soll. Der Algorithmus arbeitet ausschließlich mit Hilfe der Q-Werte.

**Definition 3.4.2.1.** Für $n \in \mathbb{N}$ ist der $n-$te überarbeitete Q-Wert für den Zustand $s_t$ und Aktion $a_t$ gegeben durch

$$Q_n(s_t, a_t) := Q_{n-1}(s_t, a_t) + \alpha_n(f(s_t, a_t) + \gamma \cdot \max_{a_{t+1} \in A_{s_{t+1}}} Q_{n-1}(s_{t+1}, a_{t+1}) - Q_{n-1}(s_t, a_t)).$$

Hierbei ist $\alpha_n \in (0, 1]$ die Lernrate des Approximationsschrittes $n \in N$.

**Bemerkung 3.4.2.2.** Man kann leicht die Struktur der Lernregel erkennen. Diese verläuft nach folgendem Muster:

$$Q(neu) = Q(alt) + Lernrate \cdot \underbrace{(Ziel - Q(alt))}_{Fehler}$$

Ähnlich wie in Kapitel 3.3 lässt sich beweisen, dass die approximierten Q-Werte gegen die Q-Werte der besten Strategie $\pi^*$ konvergieren. Die Konvergenzgeschwindigkeit hängt von der Wahl der Lernrate $\alpha$ ab. Dies wird in [9] im Detail ausgeführt.

**Algorithmus 3.4.2.3.** Wir wollen mit dieser Lernregel nun einen Algorithmus für das Q-Learning angeben. Dieser soll die oben beschriebene Q-Tabelle berechnen.
Die initialen Q-Werte $Q_0(s_t, a_t)$ für alle Zustande und Aktionen müssen vorab definiert werden, beispielsweise setzt man diese auf Null. Im Algorithmus entspricht $s = s_t$ und $s' = s_{t+1}$. Um nicht immer dieselben Strategien zu erkunden, verwenden wir hier die $\epsilon$-greedy Methode. Eine Episode bezeichnet eine Reihe von aufeinanderfolgenden Spielzügen, deren Länge vorgegeben ist. Die Anzahl der Episoden muss ebenfalls festgelegt werden. Dies bestimmt die Intensität des Trainings.

**Algorithm** Q-Learning

1: $Q(s, a) \leftarrow 0$ für alle Zustände und Aktionen;
2: **for** jede Episode **do**
3: Initialisiere Anfangszustand $s$;
4: **for** jeden Schritt der Episode **do**
5: Wähle anhand der Q-Werte die beste Aktion $a$ des Zustands $s$ oder eine zufällige Aktion ($\epsilon$-greedy);
6: $Q(s, a) \leftarrow Q(s, a) + \alpha_n(f(s, a) + \gamma \cdot \max_{a' \in A_{s'}} Q(s', a') - Q(a, s))$;
7: $s \leftarrow s'$;
8: **end for**
9: **end for**

**Beispiel 3.4.2.4.** Um diesen Algorithmus besser zu verstehen, möchten wir ein Beispiel angeben. Dieses bezieht sich auf unser Spiel Oware Abapa. Der Einfachheit halber ignorieren wir den Gegenspieler und betrachten die Spielsituation für nur einen Agenten, welcher konstant am Zug ist. Die Erläuterung zu Q-Learning für Zwei-Personen-Nullsummen-Spiele folgt später. Wir betrachten die fünf verschiedenen Spielzustände aus Abbildung 3.7. Jeder Zustand zeigt ein Spielfeld mit 12 Feldern. Die Zahlen in den Feldern beschreiben, wie viele Steine dort liegen. Ein Feld ohne Zahl beinhaltet keine Steine. Der Spieler darf aus seinen eigenen Feldern ein Feld auswählen, von welchem er zieht. Diese sechs möglichen Aktionen $a_i$ sind im Spielzustand $s_0$ gekennzeichnet und gelten für alle Zustände.

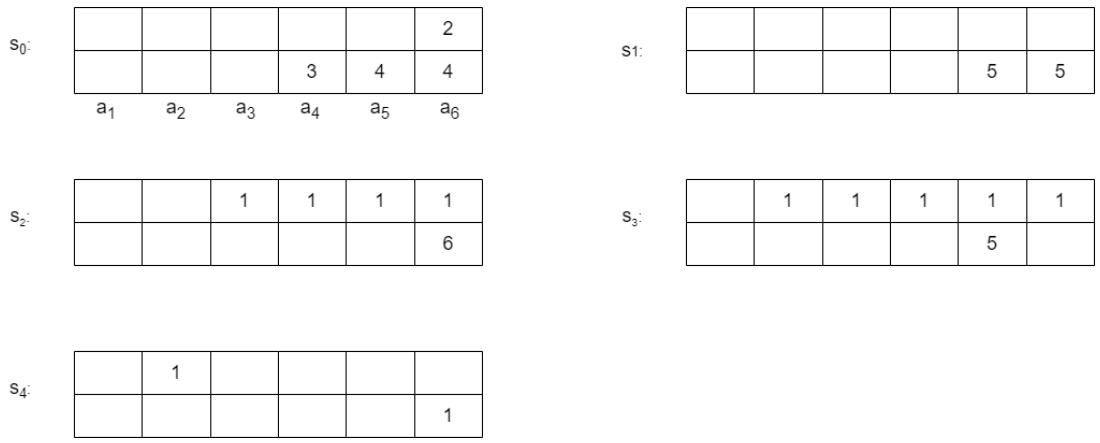


Abbildung 3.7: Fünf Zustände des Spiels Oware Abapa für das Beispiel des Q-Learning Algorithmus.

Zu Beginn wird die Q-Tabelle für alle Zustände und Aktionen mit Null initialisiert. Wir verwenden Lernrate $\alpha = 1$.

| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ |
|---|---|---|---|---|---|---|
| $s_0$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_1$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_2$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_3$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_4$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $\vdots$ | 0 | 0 | 0 | 0 | 0 | 0 |

In der ersten Episode wird Zustand $s_0$ als Anfangszustand gewählt. Da die Q-Werte bisher noch alle Null sind, kann die Aktion zufällig ausgesucht werden. Wir wählen

Aktion $a_4$, was uns zum nächsten Zustand $s_1$ führt und einen Gewinn von 3 auszahlt, da 3 Steine erbeutet wurden. Folgen wir diesem Schema, erhalten wir für die Q-Werte dieser Episode:

$$\begin{aligned} Q(s_0, a_4) &= Q(s_0, a_4) + \alpha \cdot (f(s_0, a_4) + \gamma \cdot \max_a Q(s_1, a) - Q(s_0, a_4)) \\ &= 0 + 1(3 + \gamma \cdot 0 - 0) = 3 \\ Q(s_1, a_5) &= 0 + 1(0 + \gamma \cdot 0 - 0) = 0 \\ Q(s_2, a_6) &= 0 + 1(0 + \gamma \cdot 0 - 0) = 0 \end{aligned}$$

Die Q-Tabelle ändert sich nun wie folgt.

| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ |
|---|---|---|---|---|---|---|
| $s_0$ | 0 | 0 | 0 | 3 | 0 | 0 |
| $s_1$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_2$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_3$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_4$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $\vdots$ | 0 | 0 | 0 | 0 | 0 | 0 |

Wir beginnen die nächste Episode. Wieder ist $s_0$ der Anfangszustand. In diesem Beispiel vernachlässigen wir $\epsilon$-greedy, so dass die nächste Aktion nur aufgrund der Q-Werte gewählt wird. Hier siegt also $a_4$ mit Q-Wert 3. Im zweiten Schritt wird dieses mal nicht Aktion $a_5$, sondern Aktion $a_6$ gewählt. Die neuen Q-Werte sind nun

$$\begin{aligned} Q(s_0, a_4) &= 3 + 1(3 + \gamma \cdot 0 - 3) = 3 \\ Q(s_1, a_6) &= 0 + 1(0 + \gamma \cdot 0 - 0) = 0 \\ Q(s_3, a_5) &= 0 + 1(8 + \gamma \cdot 0 - 0) = 8 \end{aligned}$$

| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ |
|---|---|---|---|---|---|---|
| $s_0$ | 0 | 0 | 0 | 3 | 0 | 0 |
| $s_1$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_2$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_3$ | 0 | 0 | 0 | 0 | 8 | 0 |
| $s_4$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $\vdots$ | 0 | 0 | 0 | 0 | 0 | 0 |

In der nächsten Episode sehen wir nun, wie sich die Bewertung der Aktion $a_5$ im Zustand $s_3$ auf die vorigen Schritte mit dem Dämpfungsfaktor $\gamma$ positiv auswirkt.

$$Q(s_0, a_4) = 3 + 1(3 + \gamma \cdot 0 - 3) = 3$$
$$Q(s_1, a_6) = 0 + 1(0 + \gamma \cdot 8 - 0) = 8\gamma$$
$$\dots$$

| | $a_1$ | $a_2$ | $a_3$ | $a_4$ | $a_5$ | $a_6$ |
|---|---|---|---|---|---|---|
| $s_0$ | 0 | 0 | 0 | 3 | 0 | 0 |
| $s_1$ | 0 | 0 | 0 | 0 | 0 | $8\gamma$ |
| $s_2$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $s_3$ | 0 | 0 | 0 | 0 | 8 | 0 |
| $s_4$ | 0 | 0 | 0 | 0 | 0 | 0 |
| $\vdots$ | 0 | 0 | 0 | 0 | 0 | 0 |

Dies beeinflusst in der nächsten Episode ebenfalls den Q-Wert für den Anfangszustand, denn

$$Q(s_0, a_4) = 3 + 1(3 + \gamma \cdot \gamma \cdot 8 - 3) = 3 + 8\gamma^2.$$

**Bemerkung 3.4.2.5.** Bisher haben wir die Theorie des Q-Learnings nur für einen Agenten, der wiederholt Züge ausführt, erklärt. Da Oware Abapa aber ein Zwei-Personen-Spiel ist, müssen wir die Aktionen des Gegners miteinbeziehen. Der einfachste Weg ist hierbei den Gegner, wie in [13] beschrieben, als Teil der Umgebung zu sehen. Das bedeutet, wenn der Agent einen Zug in Zustand $s_t$ ausführt, ist der neue Zustand $s_{t+1}$ der Zustand, der erreicht wird, nachdem der Gegner am Zug war. Die Züge des Gegners sind fix, das bedeutet, sie folgen einem festgelegten Muster. Wir verwenden dafür folgende Möglichkeiten:

(i) *Random:* Die Spielzüge des Gegners werden zufällig gewählt.

(ii) *Perfekt:* Es wird vom bestmöglichen Spielzug des Gegners ausgegangen. Dieser lässt sich z.B. anhand des Minimax-Baumes aus Kapitel 3.2.1 ableiten.

(iii) *MCTS*: Die Spielzüge des Gegners werden nach dem MCTS-Baum gewählt.

Die Tatsache, dass der Gegner als Teil der Umgebung gesehen wird, hat signifikante Auswirkungen auf die Spielchancen des Bots. Dies werden wir in einem späteren Teil dieser Arbeit sehen.
Da wir in Kapitel 3.2.1 gesehen haben, dass das Spiel Oware Abapa eine Anzahl

von Zuständen der Größenordnung $10^{11}$ besitzt, wird es nicht möglich sein, eine so große Tabelle von Zustands-Aktions-Tupeln zu erstellen und zu aktualisieren. Aus diesem Grund muss die hier vorgestellte Theorie mit Hilfe eines neuronalen Netzes umgesetzt werden. Dieses Vorgehen werden wir in Kapitel 3.5 näher beschreiben.

### 3.4.3 Policy Gradient Optimierung

Bisher haben wir eine *wertbasierte* Lernmethode betrachtet, welche die action-value-Funktion schätzt und damit eine optimale Strategie ermittelt. Neben dieser Methodik gibt es auch Verfahren, welche die Strategie direkt optimieren, ohne die action-value-Funktion zu berechnen. Man nennt diese Verfahren *strategiebasiert*. Es gibt gradientenbasierte und gradientenfreie strategiebasierte Algorithmen. Wir betrachten hier die gradientenbasierten Algorithmen (*policy gradient*), da diese auch in hochdimensionalen und nicht-diskreten Aktionsräumen gut angewandt werden können. Die policy gradient-Algorithmen haben gegenüber den wertbasierten Algorithmen außerdem den Vorteil, dass sie stärkere Konvergenzeigenschaften besitzen und auch gemischte Strategien $\pi(\cdot|\cdot)$ berechnen können. Wir beziehen die Informationen zu diesem Unterkapitel aus [10].
Unser Ziel ist es wieder $J(\pi)$ zu maximieren. Sei $\theta$ ein Überbegriff für die Parameter von denen die Strategie $\pi_\theta$ abhängt. Wir wollen $\theta$ so anpassen, dass $\pi_\theta = \pi^*$, wobei $\pi^*$ die Strategie ist, welche $J$ maximiert. Mit Hilfe des Gradientenaufstiegsverfahrens lässt sich das Optimierungsproblem schrittweise über

$$\theta_{t+1} = \theta_t + \alpha \nabla_{\theta_t} J(\pi_{\theta_t})$$

lösen. Hierbei bezeichnet $\alpha$ wieder die Lernrate und $t$ den aktuellen Schritt. Es ist jedoch sehr schwierig $\nabla_\theta J(\pi_\theta)$ direkt zu berechnen. Aus diesem Grund betrachten wir den folgenden Satz aus [10].

**Satz 3.4.3.1 (Policy Gradient Theorem).** Sei $\tau = (s_0, a_0, f(s_0, a_0), ..., s_T, a_T, f(s_T, a_T))$ eine Sequenz an Zuständen, Aktionen und Gewinnen, die sich mit der Strategie $\pi_\theta$ ergeben. Es gilt

$$\nabla_\theta J(\pi_\theta) = E_{\tau \sim \pi_\theta}(\sum_{t=0}^{T} \nabla_\theta(\log\ \pi_\theta(a_t|s_t)) Q^{\pi_\theta}(s_t, a_t))$$

*Beweis.* Es ist

$$\nabla_\theta J(\pi_\theta) = \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}(\sum_{t=0}^{T} \gamma^t f(s_t, a_t)) = \sum_{t=0}^{T} \nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}(\gamma^t f(s_t, a_t)),$$

da der Erwartungswert linear ist. Es folgt mit der Definition des Erwartungswertes und den Notationen aus Definition 3.3.1, dass

$$\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}(\gamma^t f(s_t, a_t)) = \nabla_\theta \int_{\tau_t} \gamma^t f(s_t, a_t) \underbrace{\mu(s_0) \prod_{j=0}^{t} \delta(s_{j+1}|a_j, s_j)\pi_\theta(a_j, s_j)}_{=:p(\tau_t|\pi_\theta)} d\tau_t, \tag{3.5}$$

wobei $\tau_t = (s_0, a_0, f(s_0, a_0), \cdots, s_t, a_t, f(s_t, a_t), s_{t+1})$ eine Sequenz an Zuständen, Aktionen und Gewinnen ist, die bis zum Zeitpunkt $t$ durchgegangen wird. Weiter gilt

$$\nabla_\theta p(\tau|\pi_\theta) = p(\tau|\pi_\theta) \cdot \frac{\nabla_\theta p(\tau|\pi_\theta)}{p(\tau|\pi_\theta)} = p(\tau|\pi_\theta)\nabla_\theta \log p(\tau|\pi_\theta). \tag{3.6}$$

Es folgt also durch das Vertauschen von Integral und Gradient, sowie dem Einsetzen von (3.6), dass

$$\begin{aligned}
\nabla_\theta \mathbb{E}_{\tau\sim\pi_\theta}(\gamma^t f(s_t, a_t)) &\overset{(3.5)}{=} \nabla_\theta \int_{\tau_t} \gamma^t f(s_t, a_t) p(\tau_t|\pi_\theta) d\tau_t = \int_{\tau_t} \gamma^t f(s_t, a_t) \nabla_\theta p(\tau_t|\pi_\theta) d\tau_t \\
&\overset{(3.6)}{=} \int_{\tau_t} \gamma^t f(s_t, a_t) p(\tau_t|\pi_\theta) \nabla_\theta \log p(\tau_t|\pi_\theta) d\tau_t \\
&= \mathbb{E}_{\tau\sim\pi_\theta}(\gamma^t f(s_t, a_t) \nabla_\theta \log p(\tau_t|\pi_\theta)).
\end{aligned}$$

Also gilt $\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau\sim\pi_\theta}(\sum_{t=0}^{T} \gamma^t f(s_t, a_t) \nabla_\theta \log p(\tau_t|\pi_\theta))$. Im letzten Teil des Beweises betrachten wir den Term $\nabla_\theta \log p(\tau_t|\pi_\theta)$ näher. Es ist

$$\begin{aligned}
\nabla_\theta \log p(\tau_t|\pi_\theta) &\overset{(3.5)}{=} \underbrace{\nabla_\theta \log \mu(s_0)}_{=0} + \sum_{j=0}^{t} (\underbrace{\nabla_\theta \delta(s_{j+1}|a_j, s_j)}_{=0} + \nabla_\theta \log \pi_\theta(a_j, s_j)) \\
&= \sum_{j=0}^{t} \nabla_\theta \log \pi_\theta(a_j, s_j) = \nabla_\theta \sum_{j=0}^{t} \log \pi_\theta(a_j, s_j).
\end{aligned}$$

Damit lässt sich nun die Aussage zeigen, denn

$$\begin{aligned}\nabla_\theta J(\pi_\theta) &= \mathbb{E}_{\tau\sim\pi_\theta}(\sum_{t=0}^{T}\gamma^t f(s_t,a_t)\nabla_\theta\sum_{j=0}^{t}\log\pi_\theta(a_j,s_j))\\ &= \mathbb{E}_{\tau\sim\pi_\theta}(\sum_{j=0}^{T}\nabla_\theta\log\pi_\theta(a_j,s_j)\sum_{t=j}^{T}\gamma^t f(s_t,a_t))\end{aligned}$$

Den letzten Gleichungsschritt erhält man durch einfaches Umstellen der beiden Summen. Dies wird klar, wenn man den Ausdruck der Summen ausschreibt. Dann lässt sich $\sum_{t=j}^{T}\gamma^t f(s_t,a_t)$ durch $Q^{\pi_\theta}(s_j,a_j)$ ersetzen, denn mit der Eigenschaft $\mathbb{E}(\mathbb{E}(x|y)) = \mathbb{E}(x)$ von bedingten Erwartungswerten gilt

$$\begin{aligned}\mathbb{E}_{\tau\sim\pi_\theta}(Q^{\pi_\theta}(s_j,a_j)) &= \mathbb{E}_{\tau\sim\pi_\theta}(\mathbb{E}_{a_t\sim\pi_\theta(\cdot|s_t)}(\sum_{t=0}^{\infty}\gamma^t f(s_t,a_t)|s_0=s_j,a_0=a_j))\\ &\overset{a_t\sim\pi_\theta(\cdot|s_t)\hat{=}\tau\sim\pi_\theta}{=} \mathbb{E}_{\tau\sim\pi_\theta}(\sum_{t=j}^{T}\gamma^t f(s_t,a_t))\end{aligned}$$

und somit ist $\sum_{t=j}^{T}\gamma^t f(s_t,a_t)$ ein guter Schätzer für $Q^{\pi_\theta}(s_j,a_j)$. □

**Bemerkung 3.4.3.2.** Mit Hilfe des Policy Gradient Theorems lässt sich ein Schätzer für den Gradienten wie folgt formulieren:

$$\hat{g} = \frac{1}{|\mathcal{D}|}\sum_{\tau\in\mathcal{D}}\sum_{j=0}^{T}\nabla_\theta\log\pi_\theta(a_j,s_j)\sum_{t=j}^{T}\gamma^t f(s_t,a_t) \tag{3.7}$$

Hierbei ist $\mathcal{D} = \{\tau_i\}_{i=1,...,N}$ eine Menge von Durchlaufsequenzen der Länge $T$, die zustande kommen, wenn der Agent in der vorgegebenen Umgebung der Strategie $\pi_\theta$ folgt. Der Term $\sum_{t=j}^{T}\gamma^t f(s_t,a_t)$ lässt sich zu $\gamma^j\sum_{t=j}^{T}\gamma^{t-j}f(s_t,a_t)$ umformen. Der Dämpfungsfaktor $\gamma$ ist nützlich um die hohen Schwankungen des Gradientenschätzers zu auszugleichen. In der Praxis wird $\gamma^j$ jedoch weggelassen, um ein „Überbetonen“ [10] der ersten Schritte zu vermeiden. Die letzte Summe in (3.7) wird also durch $\sum_{t=j}^{T}\gamma^{t-j}f(s_t,a_t)$ ersetzt.

**Algorithmus 3.4.3.3.** Mit diesen Erkenntnissen lässt sich nun ein einfacher Algorithmus aufstellen. Hierbei bezeichnen $a_{d,j}$ und $s_{d,j}$ die $j$-te Aktion und den $j$-ten Zustand der Sequenz $d \in \{1,...,D\}$.

**Algorithm** REINFORCE

1: Initialisiere $\theta$;
2: **for** k = 1,2,... **do**
3: Sammle $D$ Durchlaufsequenzen der Länge $T$, die mit Strategie $\pi_\theta$ erreicht werden;
4: $\nabla J(\theta) \leftarrow \frac{1}{D} \sum_{d=1}^{D} \sum_{j=0}^{T} \nabla_\theta \log \pi_\theta(a_{d,j}, s_{d,j}) \sum_{t=j}^{T} \gamma^{t-j} f(s_{d,t}, a_{d,t})$;
5: $\theta \leftarrow \theta + \alpha \nabla J(\theta)$;
6: **end for**

**Lemma 3.4.3.4 (Expected Grad-Log-Prob).** Sei $x$ eine Zufallsvariable, die $p_\theta$-verteilt ist. Dann gilt

$$\mathbb{E}(\nabla_\theta \log p_\theta(x)) = 0.$$

*Beweis.* Da $p_\theta$ eine Wahrscheinlichkeitsdichte ist, gilt $\int_x p_\theta(x) = 1$. Daraus folgt

$$\nabla_\theta \int_x p_\theta(x) = \nabla_\theta 1 = 0.$$

Mit dem selben Trick wie (3.6) in dem Beweis von Satz 3.4.3.1 erhalten wir

$$0 = \nabla_\theta \int_x p_\theta(x) = \int_x \nabla_\theta p_\theta(x) = \int_x p_\theta(x) \nabla_\theta \log p_\theta(x) = \mathbb{E}(\nabla_\theta \log p_\theta(x)). \quad \square$$

**Bemerkung 3.4.3.5.** Der einfache REINFORCE-Algorithmus hat den Nachteil, dass die Varianz des Gradientenschätzers sehr groß wird. Aus diesem Grund führen wir im Folgenden das Konzept einer *Baseline* ein, welche diese hohe Varianz reduzieren soll. Das Prinzip der Varianzreduktion ist im Anhang erklärt. Die Baseline $b(s_t)$ hängt nur vom Zustand $s_t$ ab und ist von den weiteren Aktionen und Zuständen, die mit der Strategie $\pi_\theta$ ausgeführt und erreicht werden, unabhängig. Mit Lemma 3.4.3.4 sieht man deshalb leicht, dass

$$\mathbb{E}(\nabla_\theta \log \pi_\theta(a_j, s_j) b(s_j)) = \mathbb{E}(b(s_j) \mathbb{E}(\nabla_\theta \log \pi_\theta(a_j, s_j) | s_j)) = 0.$$

Dementsprechend kann man den Gradienten auch in der Form

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{\tau \sim \pi_\theta}(\sum_{j=0}^{T} \nabla_\theta \log \pi_\theta(a_j, s_j)(\sum_{t=j}^{T} \gamma^t f(s_t, a_t) - b(s_j)))$$

schreiben, denn

$$\begin{aligned}
&\mathbb{E}_{\tau\sim\pi_\theta}(\sum_{j=0}^{T}\nabla_\theta \log \pi_\theta(a_j,s_j)(\sum_{t=j}^{T}\gamma^t f(s_t,a_t)-b(s_j)))\\
&=\mathbb{E}_{\tau\sim\pi_\theta}(\sum_{j=0}^{T}\nabla_\theta \log \pi_\theta(a_j,s_j)\sum_{t=j}^{T}\gamma^t f(s_t,a_t))-\underbrace{\mathbb{E}_{\tau\sim\pi_\theta}(\sum_{j=0}^{T}\nabla_\theta \log \pi_\theta(a_j,s_j)b(s_j))}_{=0}\\
&=\nabla_\theta J(\pi_\theta).
\end{aligned}$$

Der REINFORCE-Algorithmus lässt sich dann dementsprechend anpassen:

**Algorithm** REINFORCE mit Baseline

1: Initialisiere $\theta$;
2: **for** k = 1,2,... **do**
3: Sammle $D$ Durchlaufsequenzen der Länge $T$, die mit Strategie $\pi_\theta$ erreicht werden;
4: $\nabla J(\theta) \leftarrow \frac{1}{D}\sum_{d=1}^{D}\sum_{j=0}^{T}\nabla_\theta \log \pi_\theta(a_{d,j},s_{d,j})(\sum_{t=j}^{T}\gamma^{t-j}f(s_{d,t},a_{d,t})-b(s_{d,t}))$;
5: $\theta \leftarrow \theta+\alpha\nabla J(\theta)$;
6: Passe $b(s_{d,t})$ über $\{s_{d,t},a_{d,t},f(s_{d,t},a_{d,t})\}$ an;
7: **end for**

Eine gute Wahl für die Baseline $b(s_t)$ ist die state-value-Funktion $V(s_t)$. Die Werte $V(s_t)$ liegen nicht vor und müssen geschätzt oder approximiert werden. Dies kann beispielsweise ebenfalls über das Gradientenverfahren geschehen. Sei hierfür, wie in [22], $\hat{V}(s,\omega)$ eine Funktion, welche die wahre state-value-Funktion $V(s)$ approximieren soll, das bedeutet $J(\omega) := \frac{1}{2}(V(s)-\hat{V}(s,\omega))^2$ soll minimiert werden. Dann wird der Parameter $\omega$ über die Lernregel des Gradientenabstiegsverfahren

$$\begin{aligned}
\omega &= \omega-\alpha_\omega\nabla J(\omega)\\
&= \omega-\alpha_\omega\nabla\frac{1}{2}(V(s)-\hat{V}(s,\omega))^2\\
&= \omega+\alpha_\omega(V(s)-\hat{V}(s,\omega))\nabla\hat{V}(s,\omega)
\end{aligned} \tag{3.8}$$

optimiert. Hierbei ist $\alpha_\omega$ die Lernrate. Ein Konzept, welches dieses Verfahren aufgreift, ist die *Actor-Critic*-Methode (*AC*). Wir stellen diese vor und orientieren uns dabei an [22].

AC ist eine Mischung aus einer strategiebasierten und wertebasierten Lernmethode. Hierbei gibt es den sogenannten *actor*, welcher versucht die Strategie $\pi_\theta$ zu optimieren und das Prinzip von *critic*, welches die state-value-Funktion $V^\pi(s_t)$ approximiert. Das Zusammenspiel dieser beiden Prinzipien soll die policy gradient-Lernmethodik verbessern.
Wir stellen hier den episodischen AC-Algorithmus aus [22] vor, in welchem nicht ganze Episoden gesammelt und die Parameter damit anpasst werden (Monte-Carlo-AC), sondern die Parameterberechnung für jeden Schritt der Episode geschieht. Dies hat den Vorteil, dass über das critic-Prinzip die Wahl der nächsten Aktion durch den actor beeinflusst wird und ein schnelleres Lernverhalten beobachtet werden kann.
Im AC-Algorithmus wird

$$\sum_{t=j}^{T} \gamma^t f(s_t, a_t) - b(s_j)$$

durch den *Temporal-Difference*-Fehler (*TD*)

$$f(s_j, a_j) + \gamma V^\pi(s_{j+1}) - V^\pi(s_j)$$

der state-value-Funktion ersetzt. Der TD-Fehler kann für die state-value-Funktion analog zu dem Fehler des Q-Learnings aus Bemerkung 3.4.2.2 verstanden werden. Das bedeutet, wir minimieren den quadratischen TD-Fehler

$$J(\omega) := (f(s_j, a_j) + \gamma V^\pi(s_{j+1}, \omega) - V^\pi(s_j, \omega))^2,$$

indem wir $\omega$ über die Lernregel $\omega = \omega - \alpha_\omega \cdot \nabla J(\omega)$ wie in (3.8) anpassen.

**Algorithmus 3.4.3.6** Der Actor-Critic-Algorithmus lässt sich wie nachfolgend aufstellen. Da wir den Algorithmus in Kapitel 3.5 mit Approximation von $\pi_\theta$ und $V(s_t)$ über neuronale Netze beschreiben, werden wir nur diese, noch folgende Variante implementieren. Der hier aufgestellte Algorithmus soll die theoretischen Prinzipien von Actor-Critic verdeutlichen und als Basis für den AC-Algorithmus mit neuronalen Netzen dienen.

**Algorithm** Actor-Critic

Initialisiere $\theta$ und $\omega$, sowie $I := 1$;
**for** t = 0,1,2,... **do**
    **if** $s_t$ ist ein Endzustand **then**
        $t = 0, s_t = s_0, I = 1$;
    **end if**
    Wende Strategie $\pi_\theta$ auf $s_t$ an und berechne $\{s_t, a_t, f(s_t, a_t), s_{t+1}\}$;
    $A_t \leftarrow f(s_t, a_t) + \gamma \hat{V}(s_{t+1}, \omega) - \hat{V}(s_t, \omega)$;
    $J(\theta) \leftarrow \log \pi_\theta(a_t, s_t) A_t$;
    $\omega \leftarrow \omega + \alpha_\omega \cdot A_t \nabla \hat{V}(s_t, \omega)$;
    $\theta \leftarrow \theta + \alpha_\theta \cdot I \cdot \nabla J(\theta)$;
    $I \leftarrow I \cdot \gamma$;
**end for**

Weiterhin müssen wir noch festlegen, welche Funktion $\pi_\theta$ wir als Strategie optimieren möchten. Dafür stellen wir die *softmax*-policy wie in [21] vor. Diese eignet sich für diskrete Aktionsräume gut und ist somit für das Spiel Oware Abapa von Nutzen, da hier die Aktionen aus $\{a_1, a_2, \cdots, a_6\}$ gewählt werden.

**Definition 3.4.3.7.** Sei $\Phi(s, a)$ ein Vektor, der Zustand und Aktion speichert. Beispielsweise lässt sich der Zustand des Spielbretts von Oware Abapa als Vektor mit 14 Einträgen, bestehend aus der Anzahl der Steine in den 12 Feldern und die Anzahl der gekaperten Steine beider Partien, schreiben. Die Aktion kann als Zahl aus $\{1, 2, 3, 4, 5, 6\}$ oder als 6-komponentiger Vektor bestehend aus 0 für „nicht gewählte Aktion“ und 1 für „gewählte Aktion“ an den Zustandsvektor angehängt werden. Dann ist $\theta = (\theta_1, ..., \theta_n)^T$ der zu optimierende Parametervektor, der dieselbe Dimension wie $\Phi$ besitzt. Die *softmax*-policy ist definiert durch

$$\pi_\theta(s, a) := \frac{e^{\Phi(s,a)^T \theta}}{\sum_{b \in A_s} e^{\Phi(s,b)^T \theta}}. \tag{3.9}$$

**Bemerkung 3.4.3.8.** Mit (3.9) lässt sich $\nabla_\theta \log \pi_\theta(a, s)$ mit Hilfe der Logarithmus-Formel $\log(x/y) = \log(x) - \log(y)$ wie in [21] vereinfachen:

$$
\begin{aligned}
\nabla_\theta \log \pi_\theta(a,s) &= \nabla_\theta \log(e^{\Phi(s,a)^T\theta}) - \nabla_\theta \log(\sum_{b \in A_s} e^{\Phi(s,b)^T\theta}) \\
&= \nabla_\theta(\Phi(s,a)^T\theta) - \frac{\nabla_\theta \sum_b e^{\Phi(s,b)^T\theta}}{\sum_b e^{\Phi(s,b)^T\theta}} \\
&= \Phi(s,a) - \frac{\sum_b \Phi(s,b) e^{\Phi(s,b)^T\theta}}{\sum_b e^{\Phi(s,b)^T\theta}} \\
&= \Phi(s,a) - \sum_b \pi_\theta(s,b)\Phi(s,b) = \Phi(s,a) - \mathbb{E}(\Phi(s,a))
\end{aligned}
$$

## 3.5 Neuronale Netze

Neuronale Netze sind ein zentraler Bestandteil der künstlichen Intelligenz und heutzutage nicht mehr wegzudenken. Sie können eine Vielzahl von Aufgaben übernehmen, wie zum Beispiel Sprach- und Bilderkennung, das Einschätzen von aktuellen Situationen auf dem Finanzmarkt, die Kontrollierung von Hochgeschwindigkeitszügen oder das Entwickeln von passenden Spielstrategien. Zu Beginn dieses Kapitels möchten wir eine Einführung in neuronale Netze geben und diese mathematisch definieren. Dabei orientieren wir uns an [7],[12] und [20]. Schließlich finden Überlegungen statt, wie neuronale Netze für die Berechnung einer Spielstrategie eingesetzt werden können.

### 3.5.1 Biologischer Hintergrund

Die Idee der neuronalen Netze beruht auf der biologischen Betrachtung eines neuronalen Nervensystems. In diesem existieren eine Vielzahl von Neuronen, welche über Synapsen miteinander verbunden sind und über elektrische Spannungen kommunizieren. Sobald eine Konzentration an $Na^+$- und $Ka^+$-Ionen vorhanden ist, die einen bestimmten Schwellenwert überschreitet, wird das Neuron aktiviert und dazu veranlasst, die eigenen Informationen an das nächste Neuron weiterzugeben. Die synaptische Übertragung ist gewichtet, sodass die Neuronen unterschiedliche Funktionalitäten haben können.
Die Neuronen schalten parallel, sodass das Gehirn fähig ist, schnell denken und lernen zu können. Das neuronale Nervensystem eines menschlichen Gehirns ist sehr komplex, es wird geschätzt, dass es in etwa $10^{11}$ Neuronen enthält.
Das Ziel von neuronalen Netzen ist es, anhand dieses biologischen Vorbilds ein künstliches Netzwerk zu erschaffen, welches ebenfalls lernfähig ist.

### 3.5.2 Künstliche neuronale Netze

Im Folgenden führen wir die Architektur eines künstlichen neuronalen Netzwerkes ein. Dazu betrachten wir folgende Definitionen.

**Definition 3.5.2.1.**

(i) Der *Input-* oder *Eingabevektor* $x = (x_1, ..., x_n) \in \mathbb{R}^n$ gibt die Werte an, die das Neuron von außen erhält. Dieser ist gewöhnlich genormt um eine effizientere Berechnung zu ermöglichen.

(ii) Die *synaptischen Gewichte* $w_1, ..., w_n$ in Form eines *Gewichtsvektors* $w \in \mathbb{R}^n$ gewichten die Eingaben des Neurons je nach Funktionalität.

(iii) Eine *Transferfunktion* $f$ gibt an, wie Eingaben mit den Gewichten verrechnet werden, meistens handelt es sich um das Skalarprodukt $f(x, w) = \sum_{i=1}^{n} x_i \cdot w_i =< x, w >$.

(iv) Der *Bias* $v$ ist eine Variable, die auf das Ergebnis der Transferfunktion addiert wird, um eine mögliche Aktivierung des Neurons zu erreichen.

(v) Die *Aktivierungsfunktion* $\sigma : \mathbb{R} \to \mathbb{R}^+$ bestimmt, ob das Neuron aktiviert wird und die Informationen weitergegeben werden. Dies soll geschehen, wenn der Input des Neurons wichtig für die Ausgabe des Modells ist.

(vi) Der Variable $y \in \mathbb{R}$ bezeichnet die *Ausgabe* oder *Output* des Neurons.

Ein einschichtiges neuronales Netz kann also durch den Ausdruck

$$y = \sigma(f(x, w) + v) = \sigma(\sum_{i=1}^{n} x_i \cdot w_i + v)$$

dargestellt werden. Eine Veranschaulichung findet sich in Abbildung 3.8:

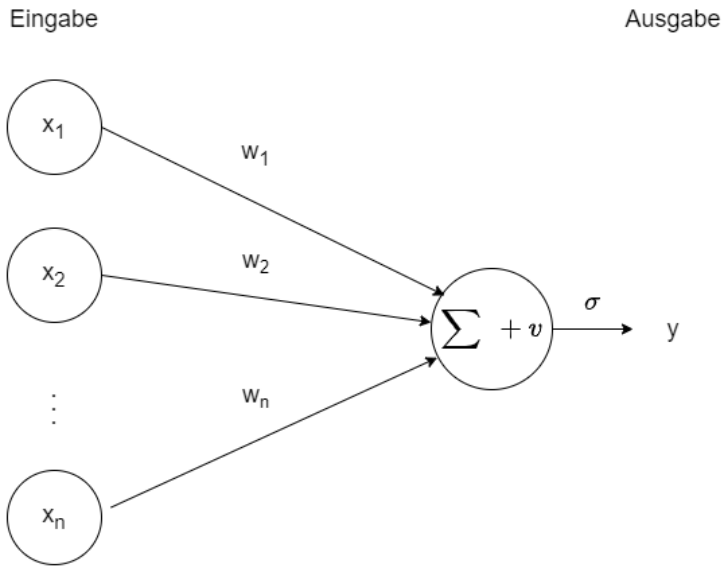


Abbildung 3.8: Einschichtiges neuronales Netz mit $n \in \mathbb{N}$ Eingabeneuronen und einem Ausgabeneuron

**Definition 3.5.2.2.** Einige ausgewählte bekannte Aktivierungsfunktionen sind

(i) die *lineare* Aktivierungsfunktion $\sigma(u) = u$. Diese eignet sich allerdings nicht um tiefe vielschichtige neuronale Netzwerke zu entwerfen oder nichtlineare Zusammenhänge darzustellen.

(ii) die *Schwellenwertfunktion* $\sigma(u) = \begin{cases} 1, & u \geq 0 \\ 0, & u < 0 \end{cases} \in [0,1]$,

(iii) die *sigmoidale* oder *logistische* Funktion $\sigma(u) = \frac{1}{1+e^{-u}} \in (0,1)$. Diese ist im Gegensatz zu (ii) differenzierbar.

(iv) die *ReLU*-Funktion (*Rectified Linear Unit*) $\sigma(u) = \begin{cases} u, & u \geq 0 \\ 0, & u < 0 \end{cases}$ ist eine sehr bekannte Aktivierungsfunktion, die sich gut für tiefe neuronale Netze eignet. Ein Problem ist allerdings, dass sie negative Eingaben sofort ausschaltet und somit eine direkte Datenverarbeitung nicht immer möglich ist.

(v) die *Softmax*-Funktion $\sigma(u_i^l) = \frac{e^{u_i^l}}{\sum_{j=1}^{m} e^{u_j^l}}$, welche $m$ Neuronen $u_1, ..., u_m$ der Schicht $l$ normalisiert, sodass eine Wahrscheinlichkeitsverteilung entsteht.

Ein neuronales Netzwerk definiert sich nicht nur durch eine Neuronenschicht, es kann aus einer Vielzahl von Schichten bestehen, die unterschiedlich viele Neuronen enthalten und unterschiedliche Aktivierungsfunktionen verwenden. Es gibt drei Arten von Neuronen-Schichten: Die *Eingabeschicht* oder *input layer* repräsentiert die erste Schicht, in welcher das Netzwerk die Eingabewerte $x_1, ..., x_n$ erhält.
Darauf folgen eine beliebige Anzahl von *Zwischenschichten* oder *hidden layers*, welche die weitere Verarbeitung vornehmen und für die Datenanalyse verantwortlich sind. Oft wird hier als Aktivierungsfunktion die ReLU-Funktion verwendet.
Die letzte Schicht ist die *Ausgabeschicht* oder *output layer*, welche die Ausgabe $y = (y_1, ..., y_m) \in \mathbb{R}^m$ für die $m \in N$ enthaltenen Neuronen erzeugt. Die Aktivierungsfunktion hierfür ist für gewöhnlich die Softmax-Funktion.
Ein mehrschichtiges neuronales Netzwerk liegt vor, wenn eine oder mehr Zwischenschichten existieren. Es berechnet sich aus einer Komposition der Ausdrücke aus Definition 3.5.2.1 und wird im Folgenden erklärt.

**Definition 3.5.2.3.** Sei $L$ die Anzahl der Schichten des Netzwerkes, sei $o_i^l$ die Ausgabe des $i$-ten Neurons der Schicht $l \in \{1, ..., L\}$ und sei $\sigma^l$ die Aktivierungsfunktion der Schicht $l$. Weiter sei $w_{ij}^l$ das Gewicht der $j$-ten Eingabe für das $i$-te Neuron

der Schicht $l$ und $v_i^l$ der Bias des gleichen Neurons. Dann lässt sich ein mehrschichtiges *neuronales Netzwerk* berechnen durch

$$o_i^l = \sigma(u_i^l)$$

$$u_i^l = \sum_{i=1}^{m} w_{ij} o_j^{l-1} + v_i^l.$$

Man bezeichnet $u_i^l$ als *propagation function*, welche die aktuelle Eingabe des Neurons $i$ aus Schicht $l$ angibt. Es ist also $y_i = o_i^L$ der $i$-te Output des Netzwerkes und $o_j^1 = x_j$ der $j$-te Input der Eingabeschicht.

**Beispiel 3.5.2.4.** Abbildung 3.9 zeigt ein solches Netzwerk, das zwei Zwischenschichten enthält. Es befinden sich $n$ Neuronen in der Eingabeschicht, $l_1$ Neuronen in der ersten und $l_2$ Neuronen in der zweiten Zwischenschicht, sowie $m$ Neuronen in der Ausgabeschicht.

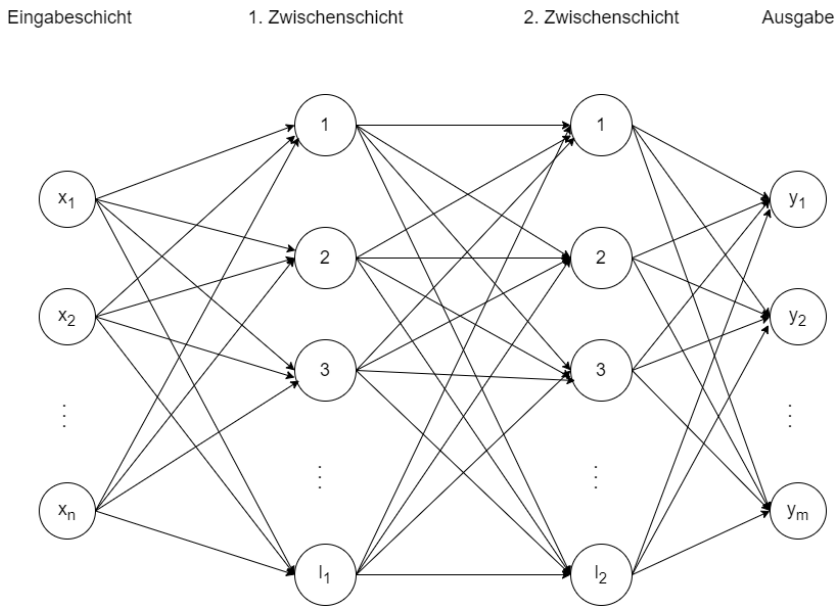


Abbildung 3.9: Schematische Darstellung eines Mehrschichtnetzes mit zwei Zwischenschichten.

Um ein Netzwerk zu trainieren, benötigt man *Loss-Funktionen*. Sie beschreiben den Fehler zwischen der Ausgabe des Netzwerkes und der wahren Ausgabe, welche für das Training vorgegeben ist. Damit lassen sich dann die Parameter des

Netzwerkes, bestehend aus Gewichten und Bias, mit Hilfe des Gradientenabstiegsverfahrens (*stochastic gradient descent* (*SGD*)) optimieren und das Netzwerk so trainieren. Das Ziel ist, die Parameter so zu wählen, dass der Fehler möglichst klein ist. Wir geben einige wichtige Loss-Funktionen an. Diese finden sich in [10].

**Definition 3.5.2.5.** Es sei $y$ die wahre Ausgabe, also das Ziel, das das neuronale Netzwerk approximieren soll (Sollwert), und $\hat{y}$ die Ausgabe des Netzwerkes (Istwert). Diese Ausgaben liegen in Vektorform mit Dimension $N \in \mathbb{N}$ vor.

(i) *$L_p$-Norm:* $L_p(y, \hat{y}) := ||y - \hat{y}||_p^p = \sum_{i=1}^{N} |y_i - \hat{y}_i|^p$

(ii) *Mittlerer quadratischer Fehler:* $L(y, \hat{y}) := \frac{1}{N}||y - \hat{y}||_2^2 = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2$

(iii) *Mittlerer absoluter Fehler:* $L(y, \hat{y}) := \frac{1}{N}\sum_{i=1}^{N} |y_i - \hat{y}_i|$

(iv) *Hubert-Loss:* $L_\delta(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2, & |y - \hat{y}| \leq \delta \\ \delta \cdot (|y - \hat{y}| - \frac{1}{2}\delta), & \text{sonst} \end{cases}$

### 3.5.3 Supervised Learning

Ein neuronales Netzwerk lässt sich auf verschiedene Arten trainieren. Die Informationen hierzu finden sich ebenfalls in [7]. Die Art des Trainings hängt immer von der Ausgangssituation des Lernproblems ab.
Bei *supervised learning*, dem überwachten Lernen, existieren Trainingsdaten, die aus Eingabedaten, z.B mögliche Risikofaktoren einer Krankheit, sowie aus den zugehörigen Ausgabewerten, z.B. Krankheitsstatus (ja/nein), zusammengesetzt sind. Anhand dieser vorgegebenen „Regeln“ lässt sich das Netzwerk trainieren, indem die Gewichte und der Bias so angepasst werden, dass der Output der neuronalen Netze möglichst wenig von den Ausgabewerten der Trainingsdaten abweicht. Das neuronale Netz wird so lange trainiert, bis der Fehler unter einer vorgegebenen Grenze liegt. Für unser Spiel Oware Abapa würde das also bedeuten, dass wir Daten benötigen, die vorgeben, welche Aktionen in welchen Zuständen gute Strategien darstellen. Dies erfordert Wissen von Expertenspielern, das uns leider nicht vorliegt.

### 3.5.4 Deep Q-Learning

Eine weitere Möglichkeit, neuronale Netze zu trainieren, bietet das Reinforcement Learning. Wir erklären nun, wie sich das in Kapitel 3.4.2 vorgestellte Q-Learning

mit neuronalen Netzen umsetzen lässt. Dies ist im Vergleich zu dem ursprünglichen Algorithmus des Q-Learnings effizienter, denn hier wird keine Q-Tabelle mehr benötigt. Wir orientieren uns hierbei an [10].
Wie in Bemerkung 3.4.2.2 beschrieben, werden die Q-Werte $Q(s_t, a_t)$ anhand ihres Fehlers, in Form der Abweichung zum Ziel $f(s_t, a_t) + \gamma \cdot Q(s_{t+1}, a_{t+1})$, aktualisiert. Dies lässt sich für das Trainieren eines neuronalen Netzes zu Nutzen machen. Wir fassen die zu trainierenden Gewichte und den Bias im Parameter $\theta$ zusammen. Dann lässt sich das neuronale Netz trainieren, indem der Parameter $\theta$ so angepasst wird, dass der Fehler

$$L(Q(s_t, a_t, \theta), f(s_t, a_t) + \gamma Q(s_{t+1}, a_{t+1}, \theta))$$

mit Hilfe des Gradientenabstiegsverfahrens minimiert wird. Der Ausdruck $Q(s, a, \theta)$ bezeichnet den Output eines Netzwerkes mit Parameter $\theta$, das als Input den Zustand $s$ erhält und die Q-Werte für die möglichen Aktionen berechnet.

**Algorithmus 3.5.4.1.** Wie in [10] beschrieben, ist es sinnvoll einen *replay buffer* mit in das Training einzubringen, in welchem die gewonnenen Erkenntnisse in Form von Tupeln $(s_t, a_t, f(s_t, a_t), s_{t+1})$ für mehrere Zeitschritte $t$ gespeichert werden. Aus diesem Buffer wird ein sogenanntes Minibatch zufällig gezogen, auf das das Q-Learning mit neuronalen Netzen angewandt wird. Ein Minibatch ist eine Auswahl an Trainingsdaten, die die Gesamtheit der vorhandenen Daten repräsentieren soll. Die Hinzunahme eines replay buffers hat den Vorteil, dass aus bisherigen Erfahrungen gelernt wird und die Daten somit effizienter genutzt werden. Ohne diesen Buffer würde außerdem ein Minibatch angelegt werden, das aus nacheinander folgenden Tupeln besteht, die miteinander korrelieren und somit das Lernresultat verschlechtern. Um Speicher zu sparen, werden nur die letzten $N \in \mathbb{N}$ Tupels gespeichert.
Eine weitere Verbesserung lässt sich erzielen, wenn man für die Berechnung der Zielwerte ein weiteres Q-Netzwerk anlegt und dieses nur nach einer festgelegten Anzahl von Schritten dem ursprünglichen, in jedem Schritt aktualisierten Q-Netzwerk anpasst, indem die Parameter übernommen werden. Das bedeutet, dass das Lernen durch die Verwendung von alten Q-Funktionen etwas verzögert wird, was das Oszillieren um den richtigen Parameter $\theta$ etwas reduzieren und eine Überschätzung vermeiden soll. Da das Ziel-Q-Netzwerk nicht jedes mal neu angepasst wird, bleibt es außerdem stabil.
Der Algorithmus, der die genannten Prinzipien zusammenführt, sieht dann wie folgt aus:

**Algorithm** Deep Q-Learning

Initialisiere den replay buffer $D$, Anfangsparameter $\theta$ für das action-value-Funktion-Netzwerk $Q$ und $\hat{\theta} = \theta$ für das Ziel-Netzwerk $\hat{Q}$ ;
**for** jede Episode **do**
    Initialisiere Anfangszustand $s_0$;
    **for** $t = 0, 1, 2...$ **do**
        Wähle die beste Aktion $a \in \arg\max_{a_t} Q(s_t, a_t, \theta)$ oder eine zufällige Aktion ($\epsilon$-greedy);
        Führe diese Aktion aus und beobachte $f(s_t, a_t)$ und $s_{t+1}$;
        Speicher Tupel $(s_t, a_t, f(s_t, a_t), s_{t+1})$ in $D$;
        Ziehe ein zufälliges Minibatch an Tupeln $(s_i, a_i, f(s_i, a_i), s_{i+1})$ aus $D$;
        **if** $s_{i+1}$ ist das Ende einer Episode **then**
            $y_i \leftarrow f(s_i, a_i)$;
        **else**
            $y_i \leftarrow f(s_i, a_i) + \gamma \cdot \max_a \hat{Q}(s_{i+1}, a, \hat{\theta})$;
        **end if**
        Passe $\theta$ mit dem Gradientenabstiegsverfahren über $L(y_i, Q(s_i, a_i, \theta))$ an;
        Synchronisiere $\hat{Q}$ nach endlich vielen Schritten;
    **end for**
**end for**

### 3.5.5 Actor-Critic mit neuronalen Netzen

Das Actor-Critic-Verfahren aus Kapitel 3.4.3 lässt sich ebenfalls mit neuronalen Netzen umsetzen. Der vorgestellte Algorithmus 3.4.3.6 beschreibt den episodischen AC-Algorithmus, welcher in jedem Schritt der Epsiode die Optimierungsparameter $\theta$ und $\omega$ anpasst. Da dieses Lernverfahren in der Praxis jedoch instabil werden kann, implementieren wir den Monte-Carlo-AC-Algorithmus mit neuronalen Netzen. Monte-Carlo-AC verwendet das gleiche Prinzip, optimiert die Parameter allerdings nach einer teilweise oder komplett durchlaufenen Episode.
Hierfür stellen wir sowohl für *actor* als auch für *critic* ein neuronales Netz auf und optimieren diese Netze nach jedem Durchlauf einer Episode der Länge $T$ wie in [25]: Sei

$$G = G_t = \sum_{t'=t}^{T} \gamma^{t'-t} f(s_{t'}, a_{t'})$$

der erwartete kummulative Reward für den Zeitschritt $t$. Dann ist $L_{actor}$ die zu minimierende Loss-Funktion des actor-Netzwerks und durch

$$L_{actor} := -\sum_{t=1}^{T} \log \pi_\theta(s_t, a_t)(G(s_t, a_t) - \hat{V}(s_t, w))$$

definiert. Die Loss-Funktion für das critic-Netzwerk $L_{critic}$ ist durch

$$L_{critic} := L_\delta(G, \hat{V}(s, w))$$

definiert. Hierbei ist $L_\delta$ der Hubert-Loss, welcher robuster gegenüber fehlerhaften Werten ist als der mittlere quadratische Fehler.
Wir optimieren die Softmax-Strategie, was bedeutet, dass die Aktivierungsfunktion der Output-Schicht des actor-Netzwerks der Softmax-Funktion aus Definition 3.5.2.2 entspricht. Abbildung 3.11 zeigt die Umsetzung des AC-Algorithmus mit neuronalen Netzen strukturell.

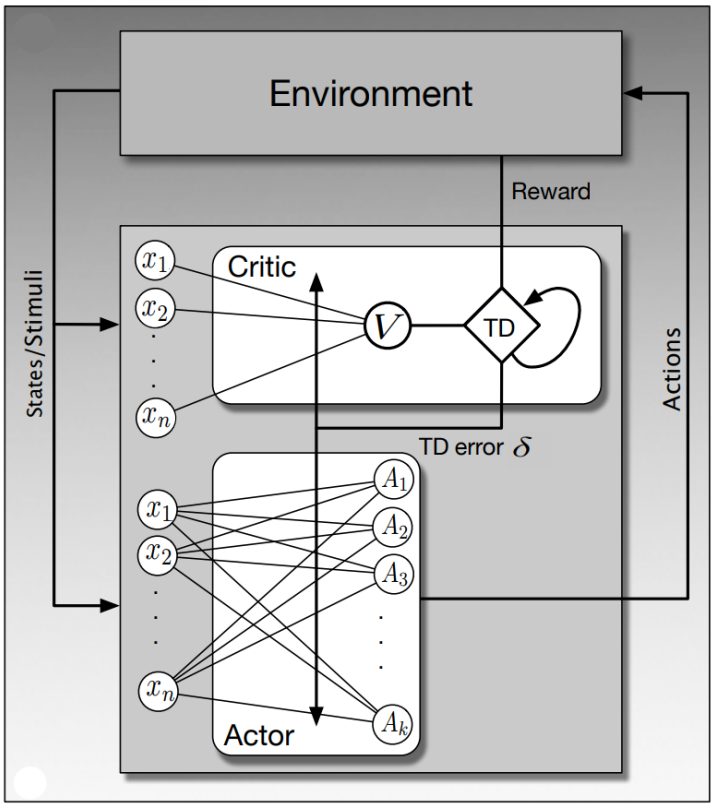


Abbildung 3.10: Actor-Critic mit neuronalen Netzen, Quelle: [22]

Mit diesem Lernverfahren erhalten wir über das actor-Netzwerk eine Strategie, nach der die Bot-Spieler ihre Züge wählen können. Es lassen sich sowohl Spiele zweier Bots gegeneinander, als auch Spiele eines Bots gegen einen Menschen realisieren. Die approximierte optimale Strategie $\pi_\theta$ in Form des actor-Netzwerks muss nur einmal berechnet werden und kann dann für beliebig viele Spiele genutzt werden.
Da dieser Algorithmus ebenfalls für nur einen lernenden Agenten konzipiert ist, werden die Züge des Gegenspielers als Teil der Umwelt gesehen. Die fixen Strategien des Gegenspielers werden wie in Bemerkung 3.4.2.5 des Ein-Agenten-Q-Learnings gewählt.

# 4 Implementierung

In diesem Kapitel erklären wir, wie die Algorithmen implementiert werden können, wie eine Spielumgebung für Oware Abapa geschaffen werden kann und wie man eine Schnittstelle aufbaut, die sowohl Bot-Bot-Spiele, Mensch-Mensch-Spiele, als auch Mensch-Bot-Spiele ermöglicht.
Alle Algorithmen werden mit Python 3.10 implementiert. Dies liegt daran, dass die aktuellste Python-Version 3.11 (Stand Januar 2023) noch nicht von der Bibliothek *tensorflow* unterstützt wird und diese für die Implementierung der Reinforcement Learning-Algorithmen benötigt wird.

## 4.1 Klassenstruktur

Für die Implementierung der KI-Algorithmen des Spiels Oware Abapa benötigen wir die Klassen *game*, *board*, *api*, sowie eine Klasse für jeden Algorithmus. Die Zusammenhänge dieser Klassen sind in Abbildung 4.1 ersichtlich und diese möchten wir nun erklären.
Grundlegend für die Spielsimulation ist die Klasse *board*, die das Spielfeld als Array beschreibt und angibt, wie viele Steine die beiden Spieler jeweils schon gekapert haben. Die aktuelle Spielsituation lässt sich ausgeben (*draw*), auf Grundlage dieser ein Zug ausführen (*move*), sowie Steine kapern (*capture*). Ob ein Spieler ausgehungert ist oder das Spiel noch nicht regulär vorbei ist, lässt sich mit *starve* und *IsNotEnd* prüfen.
Die Klasse *game* stellt die Spielstruktur für die KI-Algorithmen auf. Die Methode *play* legt fest, wann der neue Zug der KI berechnet, ausgeführt und übermittelt werden soll. Für die Berechnung des neuen Zuges ist die Methode *action_KI* angelegt. Jede Algorithmusklasse (*random*, *minimax*, *mcts*,...) erbt von der Klasse *game* und überschreibt die Methode *action_KI* mit ihrer jeweiligen Berechnung des besten Zuges.
Die Klasse *api* ist nötig um eine Schnittstelle für Bot-Bot-Spiele, Mensch-Bot-Spiele,

sowie Mensch-Mensch-Spiele bereitzustellen und wird im folgendem Kapitel 4.2 genauer erklärt.

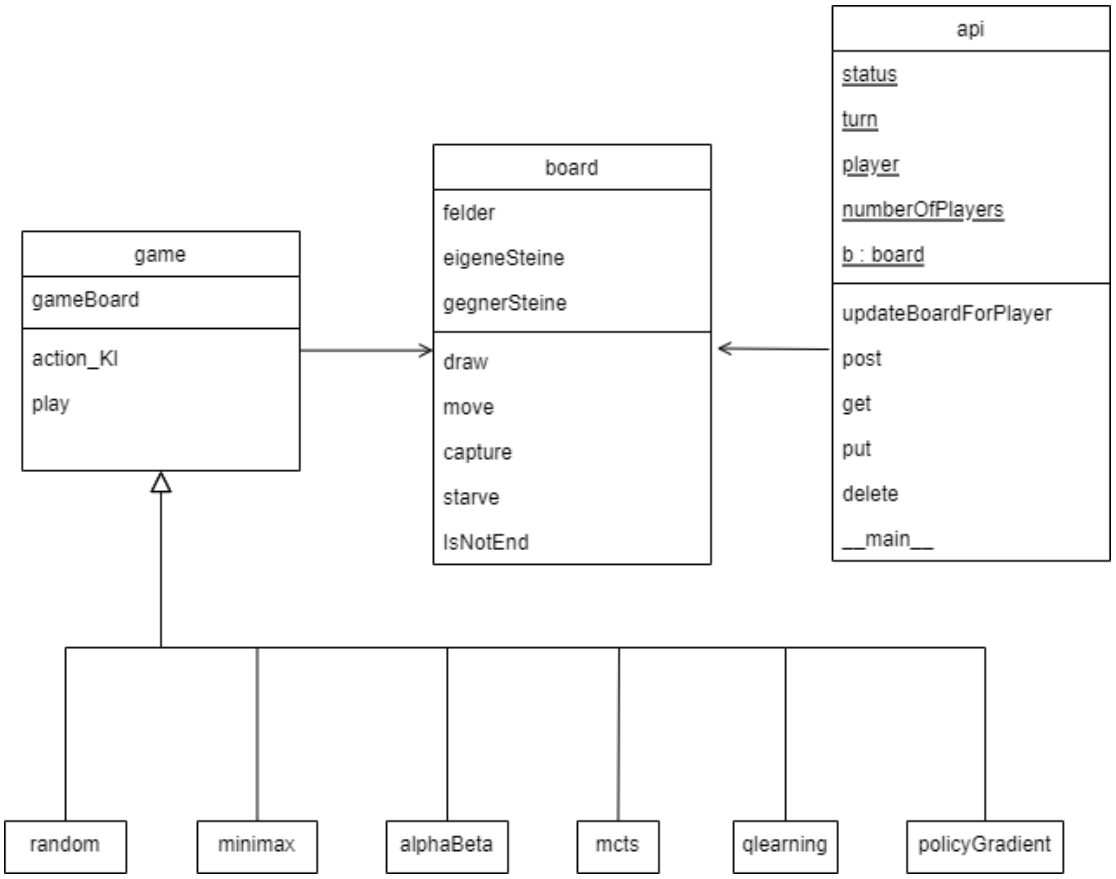


Abbildung 4.1: Klassendiagramm der Gesamtstruktur für die Implementierung

## 4.2 Anwendungsschnittstelle

Um ein Spiel zwischen zwei Parteien bereitstellen zu können, wird eine Anwendungsschnittstelle (API, engl.: application programming interface) benötigt, über welche die Spieler kommunizieren. Wir verwenden hierfür eine REST-API (Representational State Transfer), welche das Spiel als einen Webservice bereitstellt. Um dies zu ermöglichen, wird das von Python bereitgestellte Webframework *Flask* benötigt.
Über diese Schnittstelle werden spielerelevante Daten übertragen: Die Variable *status* speichert die wichtigsten Informationen des Spiels für die jeweiligen Spieler im JSON-Format. Diese Liste besteht aus der ID des Spielers, der Information, ob der Spieler gerade an der Reihe ist, dem aktuellen Spielbrett, sowie den gekaperten Steinen und den gekaperten Steinen des Gegners. Wird vom Spieler eine GET-Anfrage an die API gesendet, werden diese Informationen übermittelt. Die für Oware Abapa eigene Implementierungen für die HTTP-Anfragen GET, PUT, POST und DELETE sind in den Methoden *get*, *put*, *post* und *delete* in der Klasse *api* umgesetzt.
Um an einem Spiel teilzunehmen, muss der Spieler (egal ob Mensch oder Bot) sich über die Anfrage POST anmelden. Er übermittelt einen Namen und erhält im Anschluss seine ID und, falls vorhanden, den Namen des ebenfalls bereits angemeldeten Gegners.
Das Spielgeschehen erfolgt dann über abwechselnde PUT- und GET-Anfragen. Der Spieler übermittelt über PUT seine ID und das ausgewählte Startfeld für seinen Zug. Die Klasse *api* simuliert damit dann das Spiel und gibt die neue Statusliste zurück. Möchte man den nachfolgenden Zug des Gegners und den daraus resultierenden Spielzustand erfahren, ruft man diese Informationen wieder mit GET ab.
Die statischen Variablen *turn*, *player*, *numberOfPlayers*, *board* und die Methode *updateBoardForPlayer* sind für die Simulation des Spiels von Nutzen und werden hier nicht weiter erklärt.
Ist ein Spiel beendet oder möchte ein Spieler dieses vorzeitig abbrechen, wird von diesem eine DELETE-Anfrage gesendet und hierbei die ID des Spielers übermittelt. Diese ID wird anschließend gelöscht. Ist keine ID mehr vorhanden, was bedeutet, dass beide Spieler abgebrochen haben oder das Spiel regulär vorbei ist, werden die Informationen in *status* und sonstige spielrelevante Variablen zurückgesetzt. Ein neues Spiel kann nun wieder über eine Anmeldung mit POST begonnen werden.

## 4.3 Minimax- und Alpha-Beta-Pruning-Algorithmus

Da der Alpha-Beta-Pruning-Algorithmus, wie in 3.2.3 beschrieben, nur eine Verbesserung hinsichtlich des Rechenaufwands des Minimax-Algorithmus ist und die beiden Algorithmen sich nur durch die Hinzunahme der Abbruchvariablen *Alpha* und *Beta* unterscheiden, erklären wir hier die Implementierung für beide Algorithmen anhand von Minimax.
Da der Minimax-Suchbaum, wie in Bemerkung 3.2.1.9 beschrieben, nicht vollständig, sondern nur bis zu einer gewissen Tiefe aufgebaut wird, fügen wir dem Algorithmus eine Klassenvariable *tiefe* hinzu, die bei jedem Aufruf der Methode MAXIMUM bzw. MINIMUM übergeben und um Eins verringert wird. Ist die Tiefe bei Wert Null oder ist das Spiel vorbei, findet die Berechnung der Gewinnfunktion statt.
Um den Zustand des aktuellen Spielbrettes nicht zu überschreiben, wird in jedem Aufruf von MAXIMUM und MINIMUM eine Kopie des aktuellen Spielbrettobjektes *board* erstellt und anhand einer Liste *aktionsliste*, in der die Aktionen bis zum jeweiligen Knoten der rekursiven Tiefensuche gespeichert sind, durchgespielt. Die Liste wird wie folgt erstellt. Zu Beginn wird der MAXIMUM-Funktion eine leere Liste übergeben. Nachdem eine Aktion ausgewählt wurde, wird diese der Liste hinzugefügt und beim Aufruf der MINIMUM-Funktion übergeben. Anschließend wird die Aktion aus der Liste wieder entfernt, damit sie für einen neuen Pfad wiederverwendet werden kann. In der Methode MINIMUM findet dieses Prinzip analog statt. So ist es möglich, den Spielverlauf für jeden Pfad zu simulieren, ohne dass die Information des originalen Spielbretts, also das Spielbrett mit dem aktuellen Zustand, verloren geht.
Um den besten Zug, also den Zug, der zu dem höchsten Minimax-Wert führt, herauszufinden, wird eine Variable *bestAction* mit eingebracht. Sobald in der MAXIMUM-Methode ein Pfad eingeschlagen wird, der einen höheren Minimax-Wert erzielt, wird die Variable *bestAction* auf die Aktion gesetzt, welche auf diesen Pfad führt. In der MINIMUM-Funktion ist dies nicht von Nöten, da der Bot, welcher den Algorithmus verwendet, immer der MAX-Spieler ist. Dies ist auch der Fall, wenn zwei Bots gegeneinander spielen. Der Minimax-Algorithmus liefert am Schluss die Variable *bestAction* zurück, sodass man den bestmöglichen Zug, den man mit der Berechnung dieses Algorithmus erhält, ausführen kann. Die möglichen Werte von *bestAction* sind Integerwerte zwischen 0 und 5, da diese die sechs möglichen Startfelder des Spielbrettarrays indizieren.
Eine Übersicht der Klassenvariablen und Klassenmethoden für den Minimax- und Alpha-Beta-Pruning-Algorithmus findet sich in Abbildung 4.2.

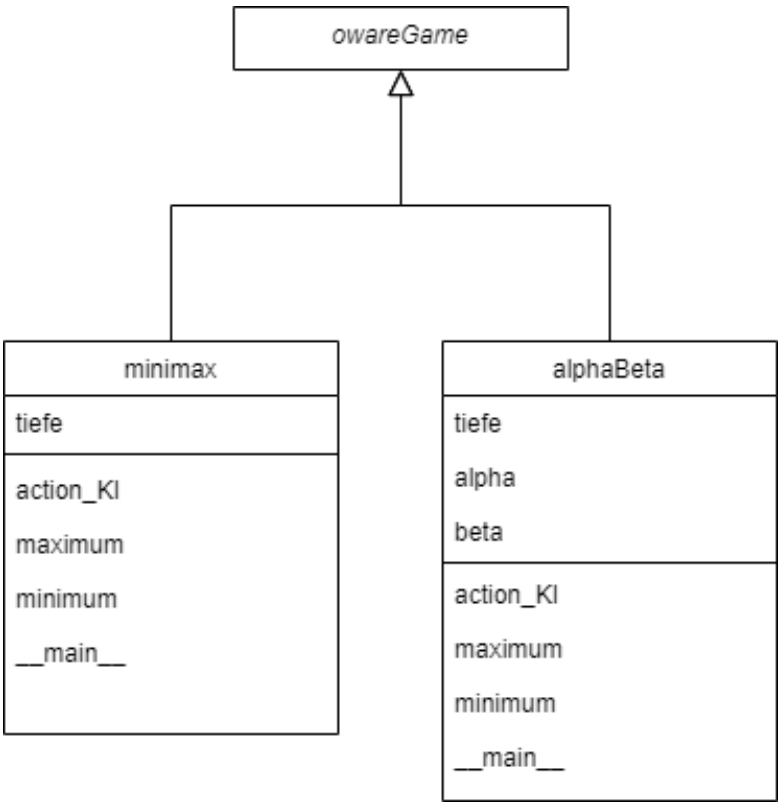


Abbildung 4.2: Klassendiagramm für die Klassen, die den Minimax-Algorithmus und den Alpha-Beta-Pruning-Algorithmus umsetzen

## 4.4 Monte Carlo Tree Search

Für den MCTS-Algorithmus implementieren wir eine Klasse *mcts*, welche die Grundschritte des Algorithmus umsetzt. Eine weitere Klasse *node* wird benötigt um die Knoten im Suchbaum zu verwalten. Da es sich hier nicht um eine rekursive Tiefensuche handelt, muss der Baum explizit berechnet werden.
Die Klasse *node* erzeugt ein Knotenobjekt, das durch folgende Attribute charakterisiert wird: Die Variable *state* hält den aktuellen Spielbrettzustand des Knotens als Array fest, die Variablen *N* und *Q* geben die Gesamtzahl der Spieldurchläufe und die gewonnen Spiele des Knotens an. Die Vernetzung der Knoten erfolgt über die Variablen *parent*, welche das Elternknotenobjekt speichert und *children*, welche die Nachfolgerknoten des Knotens in einer Liste speichert. Die Variable *actions* hält alle

Aktionen, die abhängig vom Spielzustand *state* möglich sind, in einer Liste fest und *numberActions* gibt die Länge dieser Liste an. Um die Spielpfade auch anhand der gewählten Aktionen nachvollziehen zu können, wird in *actionToNode* die Aktion (Integerwert zwischen 0 und 5) festgehalten, die vom Elternkoten zu dem eigentlichen Knoten führt. Die Methode *bestChild* berechnet, wie in Kapitel 3.3.2 beschrieben, den Nachfolgerknoten mit dem besten UCT-Wert, und *expand* legt einen neuen, bisher noch nicht existenten Nachfolgerknoten an.
Die Klasse *mcts* führt den eigentlichen MCTS-Algorithmus durch. Die Klassenvariablen *c* und *cycles* legen den Exploration-Parameter $c > 0$ des UCT-Algorithmus, sowie eine Anzahl an Zyklen, die der MCTS-Algorithmus durchlaufen soll, fest. Wir werden diese Parameter später in der Evaluierung unterschiedlich besetzen und die Performance des Algorithmus mit den verschiedenen Startwerten vergleichen. Des Weiteren werden die Methoden *TreePolicy*, *DefaultPolicy* und *backpropagate*, wie in Abschnitt 3.3.2 dargestellt, implementiert. Die statische Methode *allAction* berechnet aus dem Spielbrettzustand eines Knotens alle möglichen Zugaktionen und speichert sie in *actions*.

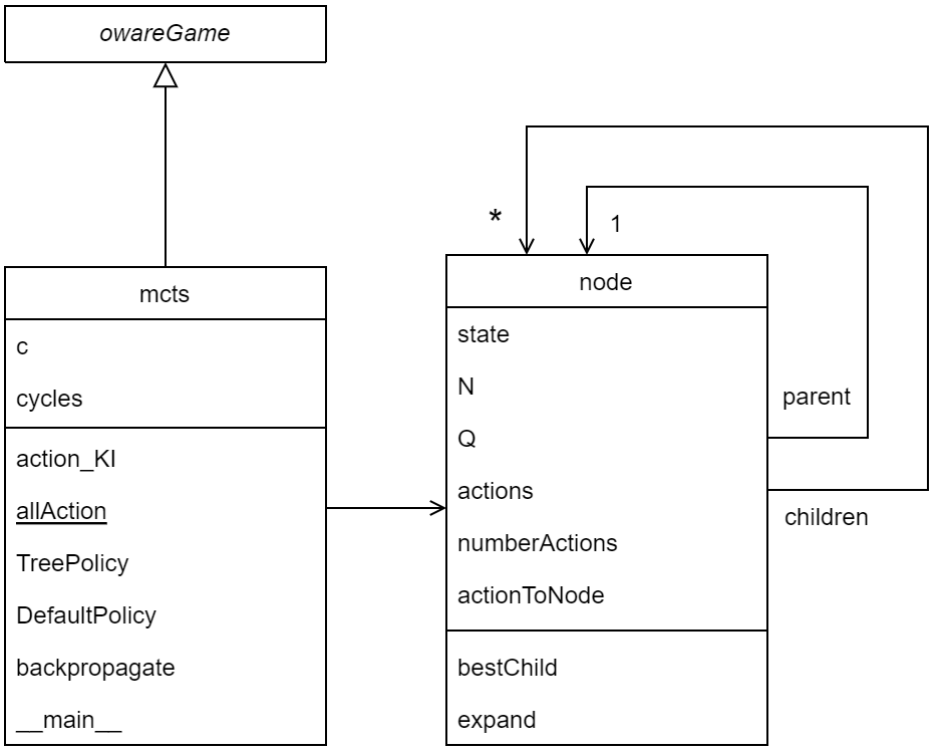


Abbildung 4.3: Klassendiagramm für den Monte Carlo Tree Search Algorithmus

## 4.5 Reinforcement Learning: Deep Q-Learning

Um Machine Learning Probleme mit Python lösen zu können, existiert das Framework *Tensorflow*. Für das Reinforcement Learning bietet Tensorflow die Bibliothek *tf-agents* an. Diese erleichtert das Verwalten der Umgebung (environment), ermöglicht eine einfache Implementierung des Agenten und stellt Replay Buffer, sowie deren Treiber bereit. Wir verwenden die Version 0.15.0. Da die Zusammenhänge der einzelnen Bestandteile (Environment, Replay Buffer, Treiber für den Replay Buffer, Agent) nicht einfach zu durchschauen sind, bietet Tensorflow Tutorials für die Anwendung von *tf-agents* an. Wir implementieren unsere Spielanwendung in Anlehnung an das Tutorial für das Trainieren eines Deep Q-Netzwerkes [24].

### 4.5.1 Environment

Um das Spielgeschehen, das Handeln des Agenten, sowie die erhaltenen Rewards zu simulieren, muss eine Klasse für die Umgebung (*environment*) aufgesetzt werden. Tensorflow bietet einige schon bestehende Umgebungen für bekanntere Spiele an, Oware Abapa ist allerdings nicht darunter. Aus diesem Grund muss eine eigene Klasse erstellt werden, die von der Standard-Pythonenvironment *PyEnvironment* erbt und die *PyEnvironment*-Methoden und Attribute nach eigenen Vorgaben überschreibt. Bei diesem Vorgehen orientieren wir uns an [23]. Die Python-Environment muss dann anschließend zu einer Tensorflow-Environment konvertiert werden, um von dem *tf-agents*-DQN-Agenten genutzt werden zu können.
Die Klassenvariablen *_action_spec* und *_observation_spec* legen das Format für die Aktionen und die zu beobachtenden Spielzustände fest. Hierbei ist zu beachten, dass die Dimension und der Datentyp korrekt angegeben werden müssen, da eine falsche Festlegung zu massiven Problemen beim Training des Agenten führen kann und diese Probleme dann nicht mehr der Environment zugeordnet werden können. Beispielsweise ist das Format für die Aktion bei unserer Anwendung durch ein Skalar des Datentyps Integer mit 32 Bits (da die Aktion nur eine Zahl zwischen 0 und 5 ist) gekennzeichnet.
Die Variable *_state* speichert den aktuellen Zustand, indem die beiden Zeilen des Arrays des Spielbretts, sowie die Anzahl der gekaperten Steine beider Spieler in einem eindimensionalen 14-elementigen Array festgehalten wird. Diese Darstellung als eindimensionales Array ist von Nöten um später als Input für das Deep Q-Netzwerk fungieren zu können. Das *_state*-Array normieren wir zusätzlich, um zu vermeiden, dass die Berechnung der Lossfunktion einen zu großen und damit

nicht verarbeitbaren Wert für den Computer erzeugt. Dies ist möglich, da wir als Loss-Funktion den mittleren quadratischen Fehler verwenden und dieser nicht robust gegenüber Ausreißern ist. Weiter ist zu beachten, dass *_state* dem Format von *_observation_spec* entspricht.
Eine weitere Klassenvariable *_episode_ended* gibt an, ob die Episode beendet ist. Dies liegt vor, wenn das Spielende eintritt oder der Agent eine Aktion wählt, die nicht gültig ist. Wir fügen außerdem noch eine Klassenvariable *b* hinzu, die ein Objekt der Klasse board erzeugt und das Spielgeschehen simulieren soll, sodass wir die *_state*-Variable immer aktualisieren können.
Die Methoden *_reset* und *_step* sind statische Methoden, die wir für unsere Anwendung ebenfalls überschreiben müssen. Die Methode *_reset* wird nach dem Ende einer Episode aufgerufen und ist eine Methode, die das Zurücksetzen der Variablen auf die Anfangswerte übernimmt. Die *_step*-Methode führt eine Aktion des Agenten durch und bewertet diese anschließend, indem die Belohnung (Reward) berechnet und als Teil der Episode gespeichert wird. Da es sich hierbei um Ein-Agenten Reinforcement Learning handelt, wird die Aktion des Gegners direkt nach der Aktion des Agenten ausgeführt, erst danach findet die Berechnung des Rewards und die Aktualisierung des Zustands *_state* statt. Eine Ausnahme gibt es für die Situation, wenn die Episode nach der Aktion des Agenten vorbei ist. Dann wird der letzte Reward ohne Beachtung des Gegners berechnet. Wir belohnen einen Gewinn mit 100, ein Unentschieden mit 0, bestrafen das Verlieren des Spiels mit -100 und bestrafen das Ausführen eines ungültigen Zuges ebenfalls mit -100. Kapert ein Spieler Steine, belohnen bzw. bestrafen wir die gewählte Aktion des Agenten mit der Anzahl der gewonnenen Steine minus der Anzahl der Steine, die der Gegner gekapert hat. Dieser Reward wird aber nur in dem entsprechenden Zug verteilt. Kapert beispielsweise zu Beginn des Spiels der Agent 5 Steine, erhält er einen Reward von 5. Verändert sich in den nächsten Runden die Anzahl der gekaperten Steine nicht weiter, erhält der Agent keine weiteren Rewards von 5 mehr.
Wir setzen drei verschiedene Umgebungen auf. Diese unterscheiden sich durch den verwendeten Gegner. Die drei Gegner wählen ihre Züge anhand eines Zufallsalgorithmus (Diese Variante bildet die Klasse *environmentRandom*.), des Minimax-Algorithmus (*environmentPerfect*), sowie des MCTS-Algorithmus (*environmentMcts*).

### 4.5.2 Trainieren des Agenten

Nachdem wir die Environment aufgesetzt haben, können wir den Agenten trainieren. Hierfür benötigen wir ein neuronales Netz, welches die Q-Werte berechnen

soll. Für diesen Zweck stellt *tf-agents* die Funktion *QNetwork* zur Verfügung, welche ein neuronales Netz erstellt und anhand der Environment die Dimension der Input- und Outputlayer anpasst. Die Anzahl der Zwischenschichten sowie die Anzahl ihrer Neuronen können individuell festgelegt werden. Wir verwenden 3 Zwischenschichten mit 100, 100 und 50 Neuronen und trainieren das Netzwerk mit Stochastic Gradient Descent und einer Lernrate von 0,0005 für *environmentRandom*, 0,00005 für *environmentPerfekt* und 0,000001 für *environmentMcts*. Die Lernrate ist ein entscheidender Parameter und darf nicht zu groß gewählt werden, da das Gradientenverfahren sonst über das gesuchte Minimum hinaus rechnet und das Netzwerk dadurch instabil wird, aber auch nicht zu klein gewählt werden, da sonst kein Lernfortschritt erfolgt. Die optimalen Parameter für die Lernrate wurden hier durch empirische Methodik bestimmt.
Mit diesem aufgestellten Q-Netzwerk lässt sich nun der Deep Q-Learning Agent initialisieren. Tensorflow bietet hierfür die Bibliothek *dqn_agent* unter *tf-agents* an.
Steht das Q-Netzwerk und der Agent, kann der Replay Buffer aufgebaut werden. Hierbei weichen wir von der offiziellen Tensorflowbeschreibung aus [24] ab, da ein Replay Buffer der Bibliothek *reverb* verwendet wird, die nicht von Windows unterstützt wird [3]. Wir verwenden stattdessen den *TFUniformReplayBuffer* von *tf-agents*. Auf diesen Replay Buffer angepasst, verwenden wir, wieder abweichend von [24], die Funktion *DynamicStepDriver* aus der Bibliothek *tf-agents*. Dies ist ein Treiber, welcher die Schritte des Agenten in der Environment ausführt und die gesammelten Erkenntnisse, bestehend aus Zustand, Aktion, Reward und Folgezustand in dem Replay Buffer speichert.
Das Ziel-Q-Netzwerk wird automatisch mit der Initialiserung des Agenten erstellt und beim Trainieren synchronisiert. Wir verwenden hierbei die default-Einstellungen von *dqn_agent*. Wir trainieren den Agenten wie folgt:

(i) 150.000 Iterationen mit *environmentRandom*

(ii) 100.000 Iterationen mit *environmentRandom* + 50.000 Iterationen mit *environmentPerfect* (Minimax mit Tiefe 2, 4 und 6)

(iii) 50.000 Iterationen mit *environmentPerfect* (Tiefe = 2) aus Schritt (ii) + 50.000 Iterationen mit *environmentMcts* (MCTS mit c=1 und 100 Zyklen)

Diese Festlegung begründet sich wie folgt: Das Trainieren des Agenten mit der *environmentRandom*-Klasse bietet den Vorteil, dass weniger Rechenoperationen nötig sind und das Trainieren somit schneller erfolgt als mit den anderen Umgebungen. Da der Agent zu Beginn des Trainings noch keiner Strategie folgt und immer eine Auswahl an Aktionen 0 bis 5 besitzt, kommt es in den anfänglichen Iterationen oft

vor, dass er sich für einen nicht erlaubten Zug entscheidet. Ein Beispiel für einen nicht erlaubten Zug wäre die Auswahl eines Startfeldes, das keine Steine beinhaltet. Da dieses Handeln mit einem Reward von -100 bestraft wird und zum sofortigen Ende einer Episode führt, hat der Agent zu Beginn keine Chance eine Gewinnstrategie zu erlernen. Der Agent lernt zuerst keine fehlerhaften Züge auszuführen. Da er dies mit *environmentRandom* als Umgebung ebenso lernt wie mit den anderen rechenintensiveren Umgebungen, verwenden wir einen Agenten mit 100.000 Iterationen der *environmentRandom*-Klasse als Basis für das Trainieren des Agenten mit *environmentPerfect*. Als Basis für *environmentMcts* dient dann das trainierte Q-Netzwerk mit *environmentPerfect* (Tiefe = 2) aus Schritt (ii).
Um eine Übersicht über das Lernverhalten zu bekommen, implementieren wir wie in [24] eine Methode *compute_avg_return*, welche nach jeder 100. Iteration aufgerufen wird und mit der aktuellen Strategie des Agenten 10 Episoden spielt. Anschließend wird der durchschnittliche kumulative Reward dieser Episoden zurückgegeben. Wir halten die Ergebnisse für die Trainingsschritte (i) bis (iii) in den folgenden Diagrammen der Abbildungen 4.4 - 4.6 fest.

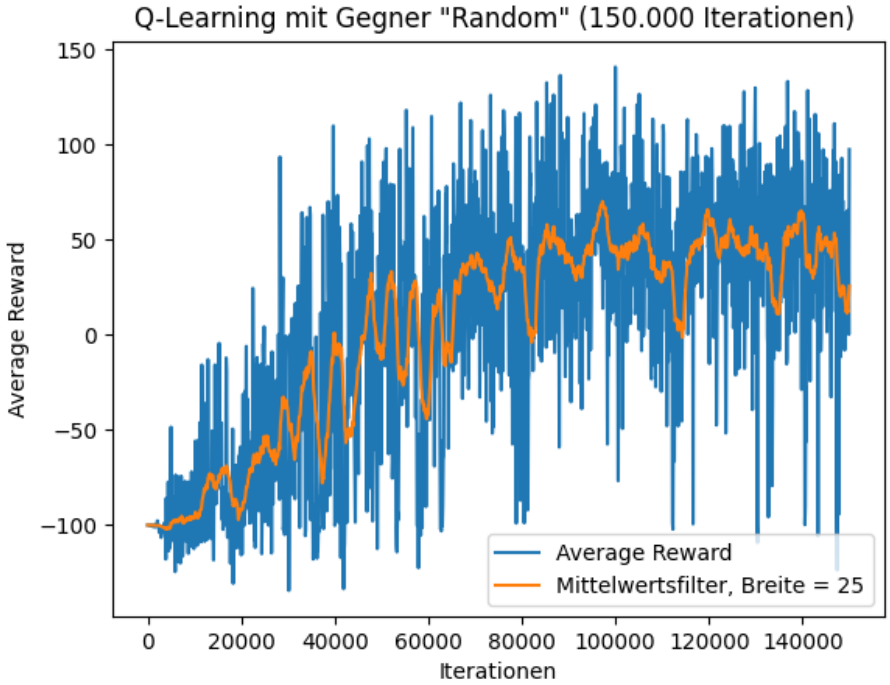


Abbildung 4.4: Deep Q-Learning, Trainingsschritt (i): 150.000 Iterationen mit *environmentRandom*

Wir sehen in Abbildung 4.4. das Lernverhalten des Deep Q-Learning-Agenten mit einem Gegner, der zufällige Züge wählt, über 150.000 Iterationen. Zu Beginn ist der mittlere Reward niedrig, er nimmt Werte um -100 an. Das geschieht, da der Agent zu Beginn noch viele nicht erlaubte Züge unternimmt. Mit zunehmenden Iterationen lernt er dieses Verhalten zu vermeiden, der mittlere Reward steigt. Dennoch ist ein starkes Oszillieren des mittleren Rewards zu erkennen. Dies lässt sich mit der hohen Anzahl an möglichen Spielzuständen ($10^{11}$, siehe Kapitel 3.2.1) erklären. Da der Gegner zufällige Aktionen wählt, gelangt der Agent zu vielen, ihm bisher noch unbekannten Zuständen, für die er erst noch eine geeignete Aktion herausfinden muss. Um eine bessere Übersicht zu erhalten, haben wir einen Mittelwertsfilter der Breite 25 auf den mittleren Reward angewendet.

Für Trainingsschritt (ii) geben wir der Übersicht halber nur ein Diagramm der letzten 50.000 Iterationen für das Deep Q-Learning mit einem Gegner, der nach dem Minimax-Algorithmus Aktionen auswählt, an. Wie beschrieben, liegt diesem Agenten aber ein Training gegen einen Zufallsalgorithmus (100.000 Iterationen) zu Grunde. Für die Abbildung 4.5 verwenden wir Tiefe 2, die Diagramme mit Tiefe 4 und 6 haben eine ähnliche Struktur und finden sich im Anhang.

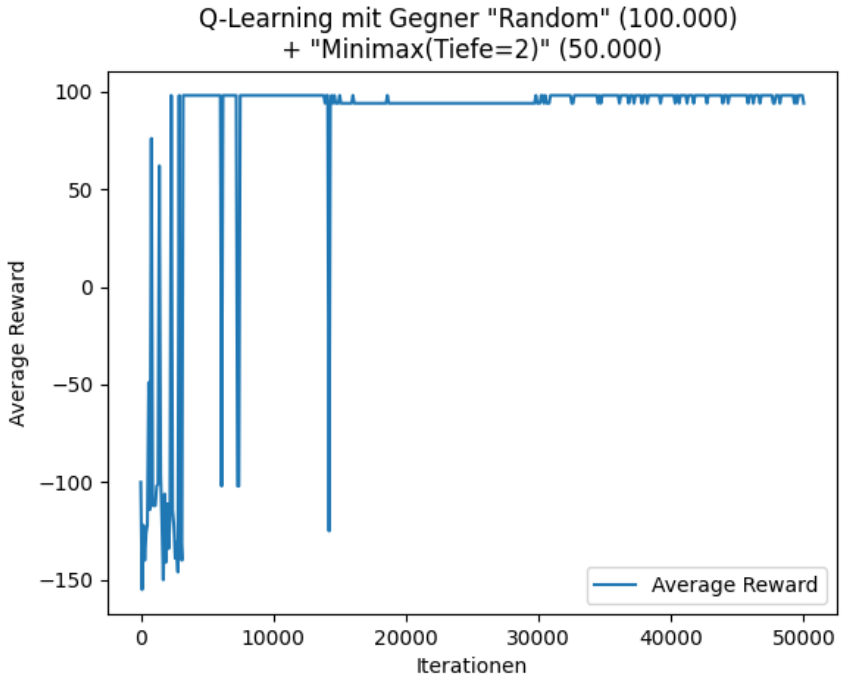


Abbildung 4.5: Deep Q-Learning, Trainingsschritt (ii): Zusätzliche 50.000 Iterationen mit *environmentPerfect* (Tiefe = 2)

Wir beobachten in den anfänglichen Iterationen wieder einen mittleren Reward von -100. Dies ist dadurch begründet, dass der zu Beginn strategielose Agent gegen einen Gegner mit Strategie (Minimax mit Tiefe 2) spielt und somit oft verliert. Nach 15.000 Iterationen hat der Agent gelernt auf seinen Gegner angemessen zu reagieren und ein mittlerer Reward von 100 stellt sich ein. Dies zeigt, dass der Agent nun ausreichend trainiert wurde, da das Gewinnen des Spiels mit einem Reward von 100 belohnt wird. Da der Minimax-Algorithmus ein deterministischer Algorithmus ist, gibt es nach 15.000 Iterationen kaum noch Schwankungen, der Agent sieht nicht viele neue Spielzustände und kann dadurch schneller lernen. Kleine Abweichungen sind durch das $\epsilon$-greedy Verfahren zu erklären.

Für den letzten Trainingsschritt (iii) geben wir ebenfalls ein Diagramm der 50.000 Iterationen, die mit *environmentMcts* durchgeführt wurden, an. Als Basis dient das in Trainingsschritt (ii) aufgestellte Deep Q-Netzwerk für *environmentPerfect* mit Tiefe 2.

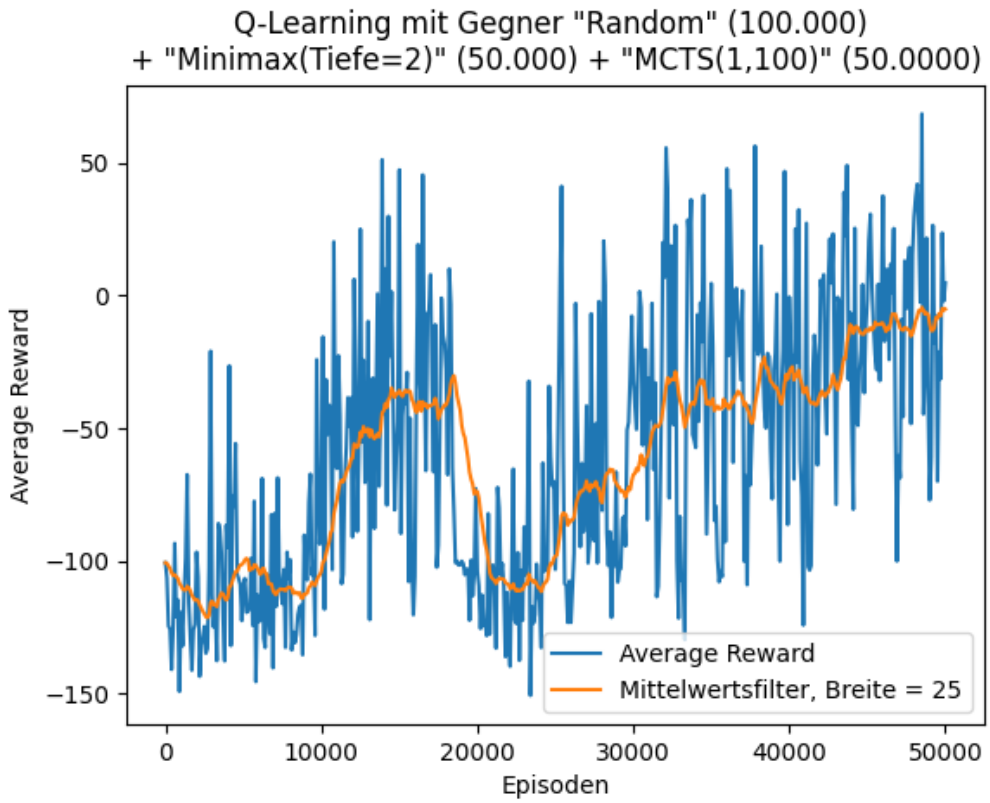


Abbildung 4.6: Deep Q-Learning, Trainingsschritt (iii): Zusätzliche 50.000 Iterationen mit *environmentMcts* (c = 1, Zyklen = 100)

Wir erkennen wieder einen aufsteigenden Verlauf des Rewards. In den anfänglichen Iterationen nimmt die Kurve einen Reward um -100 an, dieses Verhalten erklären wir dadurch, dass das durch *environmentPerfect* trainierte Netz noch nicht auf eine neue Strategie (MCTS) angemessen reagieren kann.
Bei der 20.000sten Iteration ist ein starkes Abfallen erkennbar. Dies kann dadurch erklärt werden, dass es sich bei MCTS um einen nicht-deterministischen Algorithmus handelt und so neue, bisher noch nicht betrachtete Spielzustände auftreten. Hat der Agent diese Zustände beim Training gegen *environmentRandom* noch nicht erlernt oder wurden die Erkenntnisse durch das Trainieren gegen *environmentPerfect* überschrieben, finden hier wieder vermehrt nicht gültige Züge statt, die mit einem Reward von -100 bestraft werden. Das nicht-deterministische Verhalten von MCTS erklärt auch das starke Oszillieren des Rewards im Gesamten.
Innerhalb der 50.000 Iterationen ist eine Steigerung des mittleren Rewards bis zu 0 erkennbar. Um eine Strategie zu entwickeln, welche Rewards von 100 erzeugt, müssten wesentlich mehr Iterationen stattfinden. Da der MCTS-Algorithmus aber ein sehr rechenintensiver Algorithmus ist und somit das Trainieren eines Deep Q-Netzwerks mit *environmentMcts* ebenfalls sehr viel Zeit in Anspruch nimmt (hier über 24 Stunden), sehen wir von dieser Methodik ab und betrachten dieses Verfahren nicht weiter. Eine Möglichkeit, um das Problem der langen Trainingsdauer zu verbessern, wäre die Verwendung der GPU (Graphical Processing Unit) für Berechnungsaufgaben anstelle der CPU (Central Prozessing Unit).

### 4.5.3 Nutzung der trainierten Agenten und Klassenstruktur

Um die trainierten Agenten weiter verwenden zu können, speichern wir diese nach dem erfolgten Training mittels *Checkpointer* aus der Bibliothek *tf_agents* ab. Diese Funktion erstellt einen Ordner, der die Gewichts- und Biasparameter des Q-Netzwerks, sowie weitere Parameter des Agenten in einer eigenen Datei abspeichert. Bei erneutem Aufstellen eines Q-Netzwerkes werden nach dem Aufrufen dieses Ordners die Parameter übertragen und das Netzwerk initialisiert.
Um den Agenten als Bot nutzen zu können, erstellen wir eine Klasse *qlearning*, in welcher das Deep Q-Netzwerk des Agenten initialisiert wird. Beim Aufruf der Methode *action_KI* wird die Aktion, die den besten Q-Wert des Netzwerks erzielt, gewählt und als Zugaktion für den Bot genutzt.
Das Zusammenspiel der Umgebung, des Trainings des Agenten sowie die Berechnung der besten Aktion für den Bot ist in Abbildung 4.7 als Klassendiagramm dargestellt.

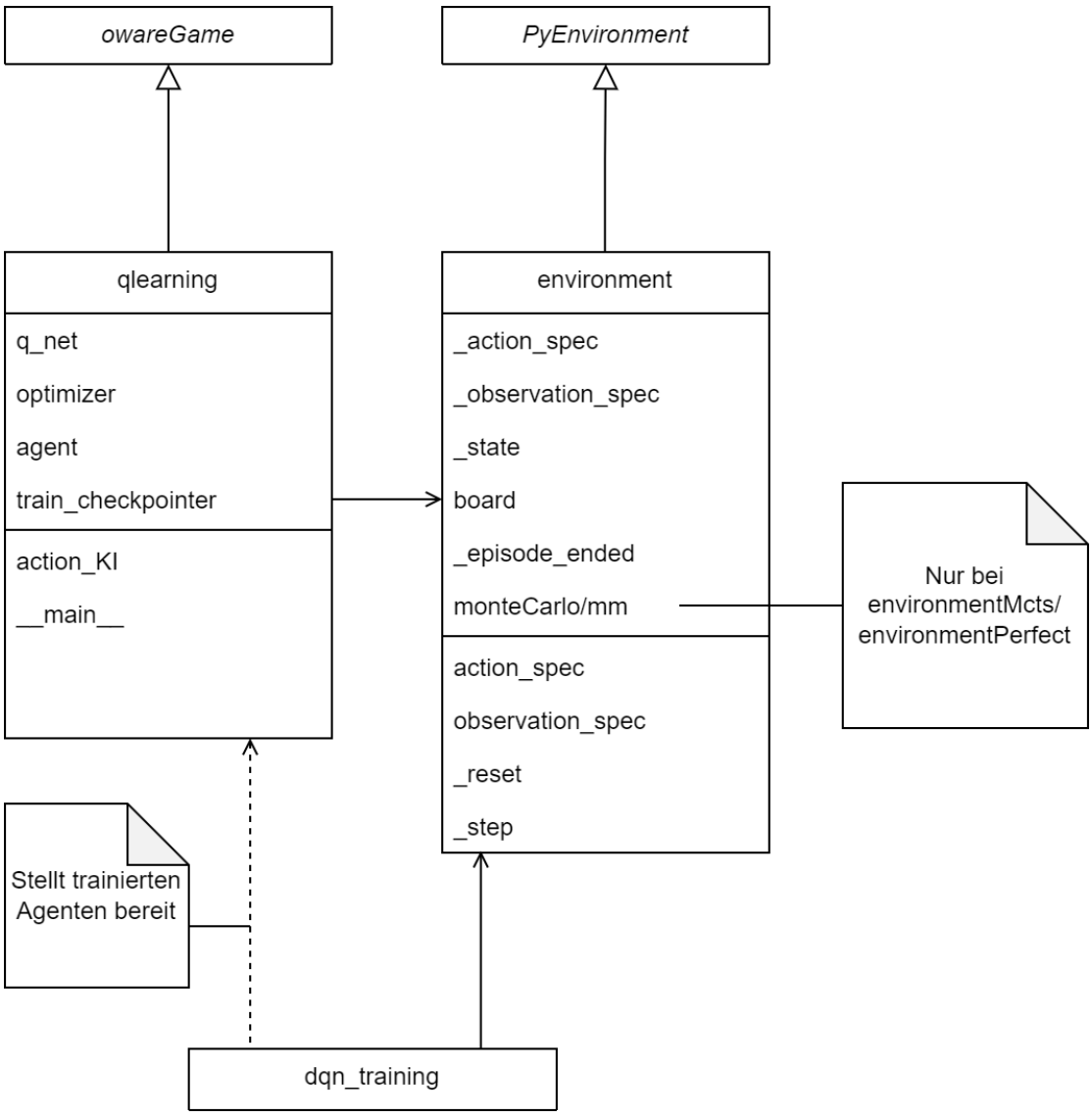


Abbildung 4.7: Klassendiagramm für Deep Q-Learning

## 4.6 Reinforcement Learning: Policy Gradient Optimierung

In diesem Kapitel erklären wir die Implementierung des Actor-Critic-Algorithmus mit neuronalen Netzen. Wir orientieren uns dabei an dem Tensorflow-Tutorial zu AC [25]. Wie im vorherigen Abschnitt zu Deep Q-Learning benötigen wir eine Umgebung und einen Agenten in Form des *actor*-Netzwerks, welchen es zu trainieren gilt. Die Klassen für die Umgebungen (*environmentRandom,...*) haben die gleiche Grundstruktur wie in Kapitel 4.5.1 und werden hier nicht näher beschrieben.

Für den *actor* und für *critic* schreiben wir eine eigene Klasse *actor_critic*, welche die beiden neuronalen Netze zusammen in einem Modell bereitstellt. Dies ist möglich, da beide Netzwerke den selben Input erhalten und die Klasse dann ein Tupel, bestehend aus der Ausgabe des *actor*-Netzwerks und der Ausgabe des *critic*-Netzwerks, zurückgibt. Die beiden neuronalen Netze haben eine gemeinsame Zwischenschicht *common* mit 250 Neuronen und der ReLU-Aktivierungsfunktion, aber verschiedene Output-Schichten. Für das *actor*-Netzwerk ist dies eine Schicht mit 6 Ausgabeneuronen für die 6 möglichen Aktionen ohne Aktivierungsfunktion. Da wir die Softmax-Strategie optimieren, wird auf den Output in späteren Berechnungen die Softmax-Funktion angewandt. Das *critic*-Netzwerk, welches den Wert von $V(s)$ berechnen soll, hat eine Ausgabeschicht mit nur einem Neuron und ebenfalls keiner Aktivierungsfunktion. Die Hyperparameter der Netze wurden empirisch bestimmt.
Für das Training der Netzwerke setzen wir den in Abschnitt 3.5.5 beschrieben Algorithmus um, ohne spezielle Python-Funktionen für das Actor-Critic-Verfahren zu verwenden. Dabei setzen wir den Dämpfungsfaktor $\gamma$ auf 0,99. Dies ist höher als beim Deep Q-Learning (0,90), da hier eine Anpassung der Netzwerke erst nach dem Durchlauf einer gesamten Episode erfolgt. Die Lernrate wird empirisch bestimmt und beträgt hier 0,01. Statt mit SGD werden die Netzwerke hier über die Adam-Optimierung angepasst.

Wir definieren erneut verschiedene Trainingsschritte:

(i) 200.000 Episoden mit *environmentRandom*

(ii) 100.000 Episoden mit *environmentRandom* + 100.000 Episoden mit *environmentPerfect* (Minimax mit Tiefe 2)

Ein Training mit *environmentMcts* findet hier nicht statt, da der MCTS-Algorithmus, wie in Kapitel 4.5.2 beschrieben, zu lange Rechenzeiten fordert. Selbst ein Training

mit nur 100 Zyklen (hier ist nicht die Episodenanzahl des Trainings gemeint) für MCTS in *environmentMcts* dauert auf einem Surface Pro 4 mit Intel-Prozessor i7-6650U länger als 24 Stunden.

Wie beim Training für die Deep Q-Netzwerke im vorigen Abschnitt, stellen wir in einer Abbildung den mittleren Reward für die Trainingsepisoden des Actor-Critic-Algorithmus dar. Hierbei berechnen wir den durchschnittlichen kumulativen Reward nach jedem 100. Durchlauf für die vergangenen 100 Episoden. Ein Mittelwertsfilter der Breite 25 wird angewandt, um einen eindeutigeren Verlauf erkennen zu können. In Abbildung 4.8 ist dieses Verfahren für Trainingsschritt (i) dargestellt.

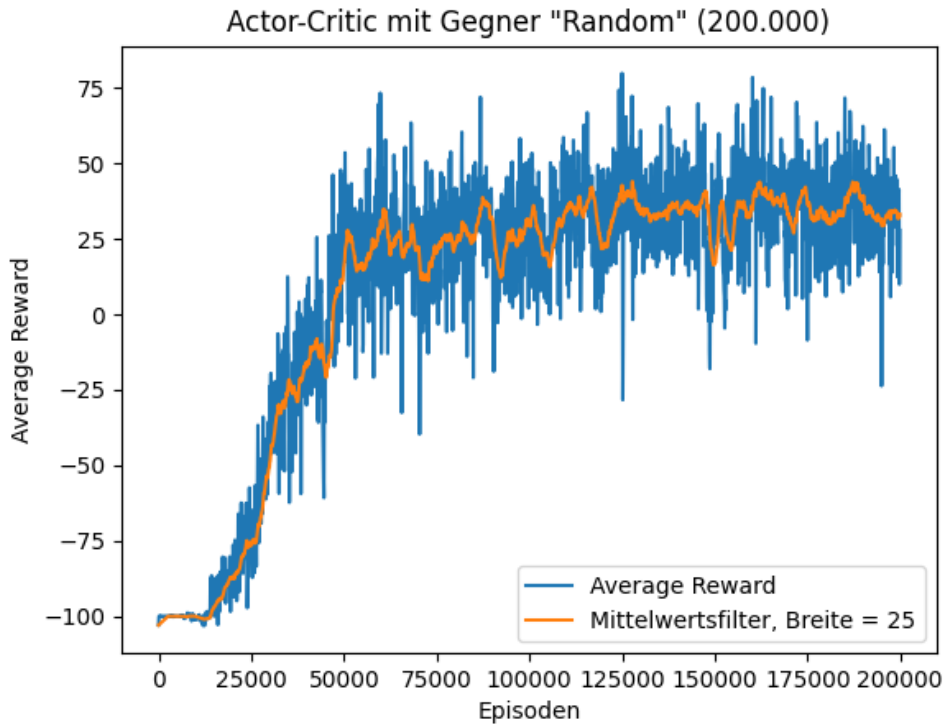


Abbildung 4.8: Policy Gradient Optimierung mit Actor-Critic, Trainingsschritt (i): 200.000 Episoden mit *environmentRandom*

Zusätzlich geben wir nach 30.000 und 150.000 Trainingsepisoden eine Statistik über die Ausgangssituationen an, um zu verstehen wie sich der mittlere Reward zusammensetzt. In Abbildung 4.9 ist für beide Zeitpunkte die Verteilung der gewonnenen, verlorenen und unentschiedenen Spiele sowie der Spiele, die durch einen nicht erlaubten Zug beendet wurden, für die darauf folgenden 100 Episoden angegeben.

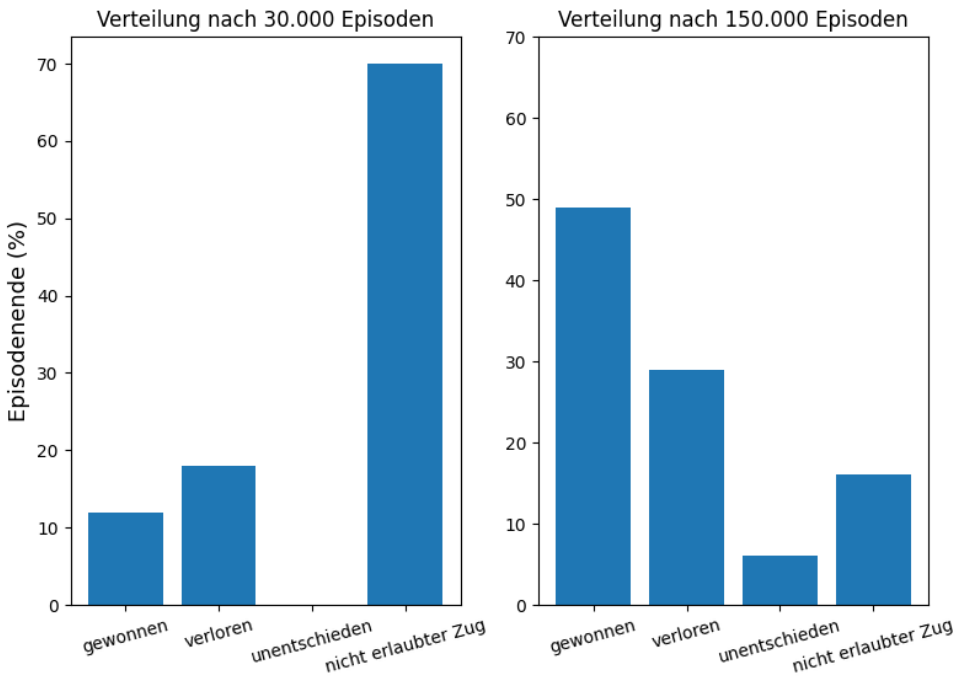


Abbildung 4.9: Verteilung der Ausgangssituation nach 30.000 und 150.000 Trainingsepisoden für AC mit *environmentRandom*

Wir erkennen in Abbildung 4.8 wieder einen ansteigenden Verlauf des Rewards für steigende Epsiodenanzahl. Die anfänglichen Werte befinden sich bei -100, da der Agent hier noch viele nicht erlaubte Zugmöglichkeiten wählt. Dieses Verhalten ist auch durch die Verteilung der Ausgangssituation in Abbildung 4.9 ersichtlich. Zu einem frühen Zeitpunkt des Trainings (30.000.Episode) ist der Prozentsatz von Episoden, die durch einen nicht erlaubten Zug beendet wurden, mit 70% wesentlich höher als zu einem späteren Trainingszeitpunkt (150.000. Episode) mit 16%.
Die Kurve des Diagramms 4.8 oszilliert stark, was durch die vielen unterschiedlichen Spielzustände, die durch das Trainieren gegen einen Zufallsgegner zustande kommen, erklärbar ist. Der mittlere Reward pendelt sich auf Werte um 25 ein. Dies ist kein perfektes Ergebnis, da Reward-Werte um 100 eine wesentlich höhere Gewinnchance versprechen, allerdings ist das Lernen einer Strategie für das Spiel erkennbar, da der Algorithmus nicht zufällig spielt (dies wäre bei einem mittleren Reward von 0 der Fall). Dies bekräftigt auch Abbildung 4.9, da nach 150.000 Episoden der Anteil der gewonnen Spiele mit ca 50% überwiegt. Eine mögliche Erklärung für das Stagnieren der Reward-Werte um 25 ist, dass das hier gewählte Netzwerk-

Modell sich nicht weiter auf die hohe Anzahl an Spielzuständen generalisieren lässt.

Beim Training mit *environmentPerfect* fällt folgende Herausforderung auf: Auf Grund der Tatsache, dass ein intelligenter Algorithmus als Gegner fungiert, verliert der Actor-Critic-Algorithmus zu Beginn des Trainings häufig. Da das Verlieren mit einem Reward von -100 bestraft wird und die gekaperten Steine des Gegners ebenfalls davon abgezogen werden, liegt am Ende der Episode ein Reward von unter -100 vor. Da ein nicht gültiger Zug mit einem Reward von -100 bestraft wird, ist es für den AC-Algorithmus von Vorteil, wenn der Gegner möglichst wenige Steine kapert. Dies kann erreicht werden, wenn der AC-Algorithmus schnell einen nicht gültigen Zug ausführt. Das Gradientenaufstiegsverfahren läuft also in ein lokales Maximum.
Um dies zu verhindern und das globale Maximum zu erreichen, verändern wir die *environmentPerfect* für Trainingsschritt (ii) wie folgt: Nur das Kapern der eigenen Steine wird zwischen den Schritten einer Episode in den Reward mit eingerechnet. Kapert der Gegner Steine, hat dies bis zum Spielende keinen Einfluss auf den Reward. Lediglich ein Reward von -100 für ein verlorenes Spiel gibt Auskunft über die Performance des Gegners. So wird das Ausführen eines ungültigen Zuges gleichgesetzt mit dem Verlieren der Spielrunde und es entsteht kein lokales Maximum.
Wir trainieren das Netzwerk für Trainingsschritt (ii) mit einer Lernrate von 0,01 und geben das Ergebnis in Abbildung 4.10 an.
Es lässt sich erkennen, dass es sich hierbei um kein optimales Lernverhalten handelt. Es wird zwar ein mittlerer Reward von 100 erreicht, aber dieser kann nicht gehalten werden. Es ist keine Konvergenz des kumulativen Rewards zu erkennen. Dies kann ebenfalls nicht herbeigeführt werden, wenn die Lernrate verändert wird oder das Training mehr Episoden durchläuft.
Wir schließen daraus, dass der Actor-Critic-Algorithmus nicht für das Lernen des Spiels Oware Abapa mit der Umgebung, in welcher der Gegner über den Minimax-Algorithmus simuliert wird, geeignet ist. Dieses Verhalten kann unterschiedliche Gründe haben. Naheliegend ist jedoch, dass der Minimax-Algorithmus durch seine deterministische Eigenschaft wenig Spielraum bietet, ihn mit unterschiedlichen Zügen zu besiegen. Lernt der AC-Algorithmus eine Strategie, mit welcher sich der Minimax-Algorithmus besiegen lässt, muss er bei dieser bleiben. Ein weiteres Training bietet zwar eine Generalisierbarkeit des Netzes, führt aber auch dazu, dass mit einem neuen abweichenden Schritt der Episode das Spiel verloren werden kann und die vorher gewinnführende Strategie nun nicht mehr als sinnvoll erachtet wird.

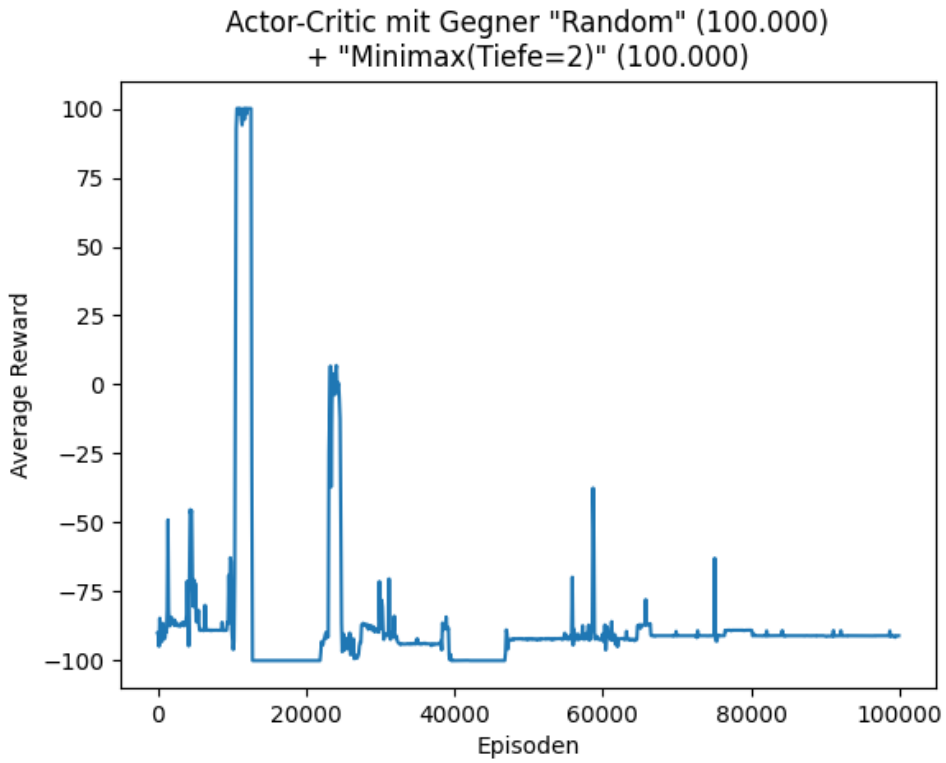


Abbildung 4.10: Policy Gradient Optimierung mit Actor-Critic, Trainingsschritt (ii): Zusätzliche 100.000 Episoden mit *environmentPerfect* (Tiefe = 2)

Nach dem Trainieren speichern wir die Gewichte der Netzwerke von *actor* und *critic*. Hierfür verwenden wir die Tensorflow-Funktion *save_weights*. Um die erlernte Strategie nun für zukünftige Spiele nutzen zu können, erstellen wir ein neues Objekt der Klasse *actor_critic* und initialisieren das Modell, indem wir die Gewichte über *load_weights* laden.
Wir nutzen für die Berechnung der besten Zugaktion nur das *actor*-Netzwerk und bestimmen diese über die höchste Ausgabe der 6 Output-Neuronen. Die Ausgabe des Netzwerks kann mit der Methode *call* abgerufen werden. Um zu vermeiden, dass ungültige Aktionen gewählt werden, setzen wir alle Ausgabewerte auf $-\infty$, die leere Felder repräsentieren.
Eine Übersicht der Klassenstruktur findet sich in Abbildung 4.11.

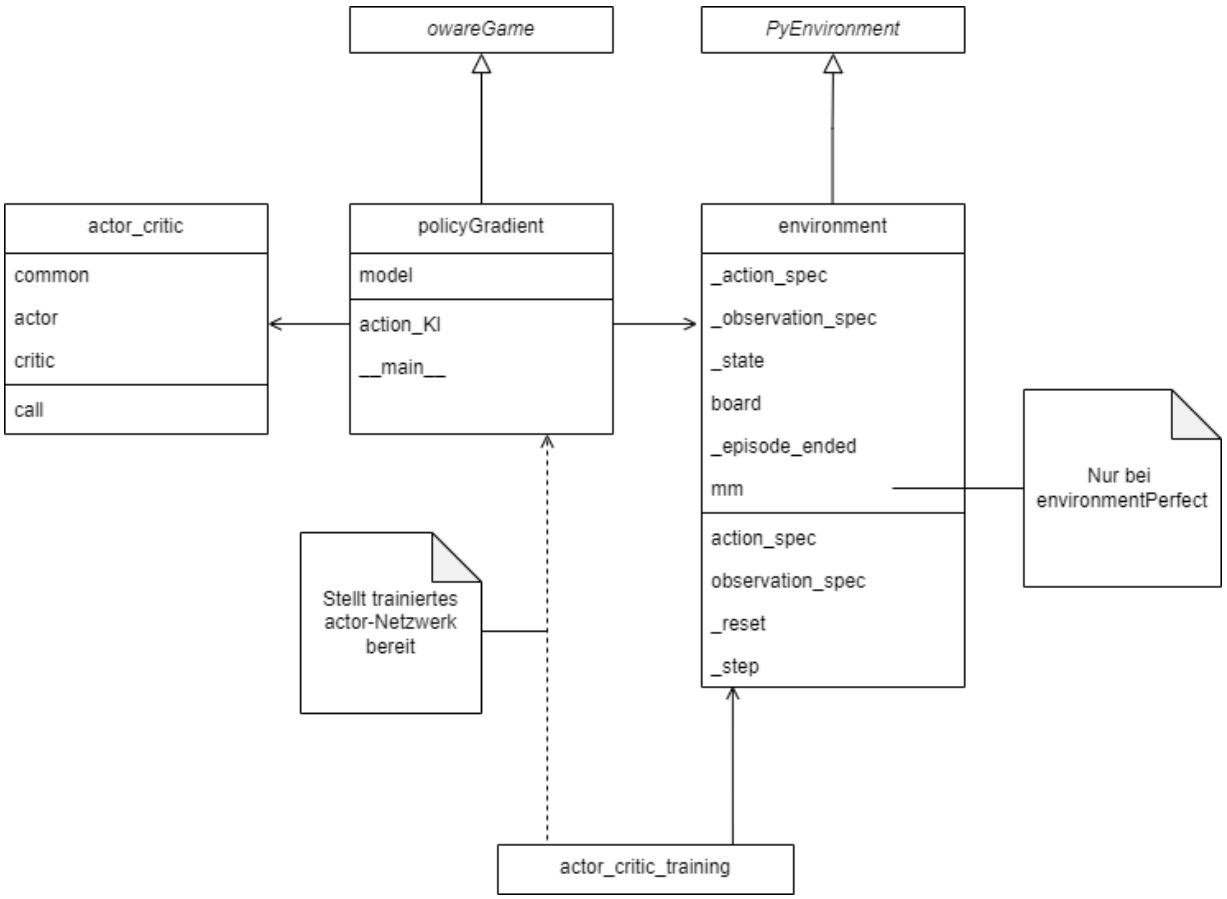


Abbildung 4.11: Klassendiagramm für Policy Gradient Optimierung mit Actor-Critic

# 5 Vergleich und Evaluierung

In diesem Teil der Arbeit möchten wir die Anwendung der KI-Algorithmen auf das Spiel Oware Abapa evaluieren, indem wir die Algorithmen hinsichtlich der aufzuwendenden Rechenzeit, den Gewinnchancen und der durchschnittlichen Anzahl an Zügen miteinander vergleichen.
Wir lassen unterschiedliche Algorithmen gegeneinander spielen und variieren dabei die verschiedenen Parameter (Suchtiefe, Zyklenanzahl,...), um die Verfahren untereinander zu vergleichen, aber auch um die besten Parameter für den entsprechenden Algorithmus herauszufinden.

## 5.1 Vergleich der Adversarial Search Algorithmen

Wie in Kapitel 3.2.3 beschrieben, ist der Suchbaum des Alpha-Beta-Pruning-Algorithmus weniger komplex als der Suchbaum des Minimax-Algorithmus. Dies wirkt sich auch auf die Rechenzeit der rekursiven Tiefensuche aus. Um diese Eigenschaft zu überprüfen, lassen wir zwei Bots mit den beiden Algorithmen gegeneinander spielen und messen die durchschnittliche Rechenzeit für die Ausführung eines Zuges in Sekunden. Dabei variieren wir die Suchtiefe der Algorithmen.
Auf Grund der Tatsache, dass beide Algorithmen deterministisch sind, würden mehrere Spieldurchläufe zum exakt selben Ergebnis führen. Somit wird pro Suchtiefe nur ein Spieldurchlauf benötigt, um vergleichbare Resultate zu erzeugen. Nachdem ein Spiel beendet ist, wird ein neues gestartet und die Tiefe der Suchbäume um Eins erhöht. In Abbildung 5.1 ist ein Diagramm dargestellt, welches die Rechenzeiten der Adversarial Search Algorithmen für Tiefen zwischen 1 und 9 angibt.

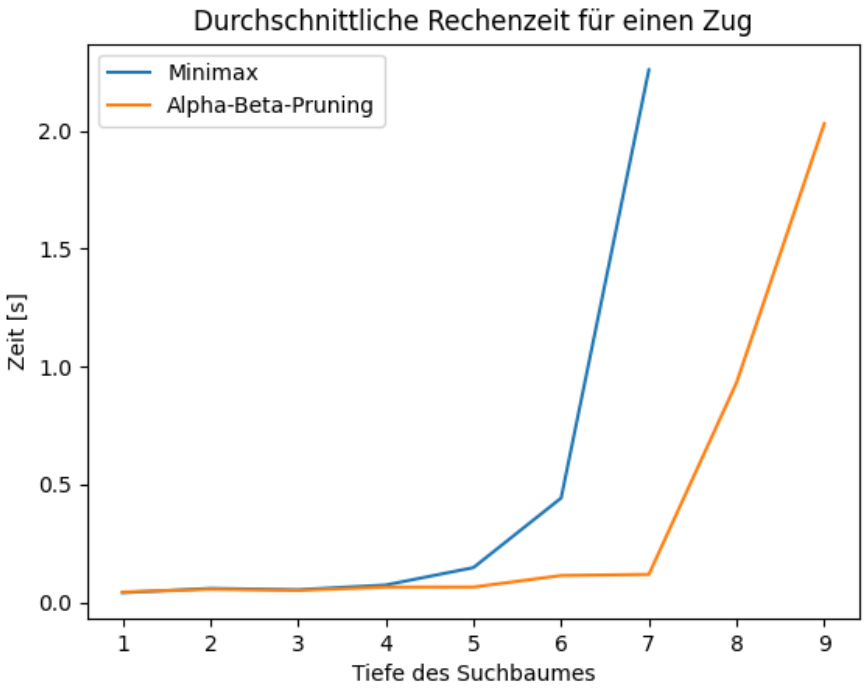


Abbildung 5.1: Vergleich der durchschnittlichen Rechenzeit eines Zuges des Minimax- und Alpha-Beta-Pruning-Algorithmus für unterschiedliche Suchbaumtiefen

Es lässt sich gut erkennen, dass die Rechenzeiten fakultativ ansteigen und der Minimax-Algorithmus mit steigender Suchbaumtiefe um einiges langsamer wird als der Alpha-Beta-Pruning-Algorithmus. Da beide Algorithmen, wie in Kapitel 3.2.3 beschrieben, das selbe Ergebnis berechnen, werden wir für die weiteren Untersuchungen nur den Alpha-Beta-Pruning-Algorithmus heranziehen. Dies ermöglicht auch eine intensivere Suche mit höheren Suchbaumtiefen. Wir verwenden für den weiteren Vergleich in dieser Arbeit Tiefen zwischen 1 und 8.

Um festzustellen, welche Suchtiefe sich am besten eignet, lassen wir den Alpha-Beta-Pruning-Algorithmus gegen andere Algorithmen antreten und untersuchen, wie oft dieser gewinnt, verliert oder ein Unentschieden auftritt.
In diesem Abschnitt vergleichen wir die Gewinnchance gegen einen Zufallsalgorithmus. Wir führen 100 Spiele mit dem Alpha-Beta-Pruning-Algorithmus für jede betrachtete Suchtiefe gegen den Zufallsalgorithmus durch und geben die Gewinnbilanz prozentual an.
Da wir die Größe des Suchbaumes nicht kennen und dessen Pfade unterschiedlich

tief sein können, ist es schwierig eine Aussage darüber zu treffen, ob es für den Alpha-Beta-Pruning-Algorithmus von Vorteil ist, beginnender oder nachziehender Spieler zu sein. Diese Problematik haben wir in Kapitel 3.2.2 erörtert. Um gleiche Bedingungen zu schaffen, ist bei diesem Vergleich der Alpha-Beta-Pruning-Algorithmus immer der beginnende Spieler. Die Spielergebnisse sind in Tabelle 5.1 angegeben.

| Suchtiefe | Gewonnen (%) | Verloren (%) | Unentschieden (%) |
|---|---|---|---|
| 1 | 91 | 8 | 1 |
| 2 | 100 | 0 | 0 |
| 3 | 98 | 2 | 0 |
| 4 | 98 | 2 | 0 |
| 5 | 96 | 3 | 1 |
| 6 | 97 | 2 | 1 |
| 7 | 98 | 2 | 0 |
| 8 | 98 | 2 | 0 |

Tabelle 5.1: Prozentualer Anteil der Gewinne, verlorenen Runden sowie Unentschiedenspiele des Alpha-Beta-Pruning-Algorithmus gegen einen Zufallsalgorithmus für unterschiedliche Suchtiefen

Wir erkennen, dass der Alpha-Beta-Pruning-Algorithmus in der Regel sehr gut spielt. Spiele mit einer Suchtiefe von 1 haben mit 91% Gewinnwahrscheinlichkeit die niedrigste Erfolgsquote. Dies lässt sich dadurch erklären, dass der Algorithmus mit Suchtiefe 1 das Handeln des Gegners nicht mit in die Berechnung einbezieht und nur seine Aktion für diese Runde bewertet. Wir schließen Suchtiefe 1 als besten Parameter für die Adversarial Search Algorithmen aus.

Im nächsten Schritt untersuchen wir, wie viele Runden der Alpha-Beta-Pruning-Algorithmus durchschnittlich benötigt um zu gewinnen. Dafür lassen wir den Algorithmus für jede Suchtiefe 100 mal gegen den Zufallsalgorithmus spielen und berechnen das arithmetische Mittel für die Rundenanzahlen bei gewonnen Spielen. Zusätzlich geben wir die Standardabweichung (SD) und das 95%-Konfidenzintervall (KI) an. Die Ergebnisse sind in Tabelle 5.2 abgebildet.
Wir erkennen, dass die durchschnittliche Anzahl an Zügen mit zunehmender Suchtiefe leicht sinkt. Für Suchtiefe 5 liegt mit 16,81 Zügen das Minimum vor.

| Suchtiefe | Mittlere Anzahl Züge bei Gewinn (SD) | 95%-KI |
|---|---|---|
| 1 | 20,96 (7,57) | [19,41, 22,50] |
| 2 | 19,48 (6,70) | [18,10, 20,87] |
| 3 | 19,69 (6,71) | [18,34, 21,04] |
| 4 | 18,63 (7,56) | [17,10, 20,16] |
| 5 | 16,81 (6,32) | [15,54, 18,08] |
| 6 | 18,89 (6,05) | [17,70, 20,08] |
| 7 | 18,18 (5,76) | [17,04, 19,33] |
| 8 | 17,80 (6,01) | [16,60, 18,99] |

Tabelle 5.2: Statistische Auswertung der Rundenanzahl für gewonnene Spiele des Alpha-Beta-Pruning-Algorithmus gegen einen Zufallsalgorithmus für unterschiedliche Suchtiefen

## 5.2 Auswertungen mit Monte Carlo Tree Search

Um den MCTS-Algorithmus mit den Adversarial Search-Algorithmen und dem Zufallsalgorithmus zu vergleichen, setzen wir den Explorations-Parameter konstant auf 1. Dies bedeutet, dass ein ausgewogenes Verhältnis zwischen Exploration und Exploitation herrscht. Der für uns relevante Parameter ist die Anzahl der Zyklen, die der Algorithmus in jeder Runde durchläuft. Wir betrachten die durchschnittliche Rechenzeit eines Zuges, die Gewinnchancen gegen Random und den Alpha-Beta-Pruning-Algorithmus, sowie die Anzahl an Zügen in gewonnen Spielen für verschiedene Zyklenanzahlen.

In Abbildung 5.2 ist die durchschnittliche Rechenzeit eines Zuges des MCTS-Algorithmus in Sekunden für verschiedene Zyklengrößen angegeben. Wir betrachten dabei Zyklenanzahlen zwischen 200 und 1400 in Abständen von 100. Für jede Anzahl wird ein Spiel zwischen dem MCTS-Algorithmus mit diesem Parameter gegen den Zufallsalgorithmus gestartet. Dabei wird für jeden Zug von MCTS die Rechenzeit erfasst und am Schluss das arithmetische Mittel gebildet und in das Diagramm eingetragen. Wie zu erwarten, steigt die Rechenzeit mit Anzahl der Zyklen. Wir erkennen einen nahezu linearen Verlauf mit kleinen Schwankungen.

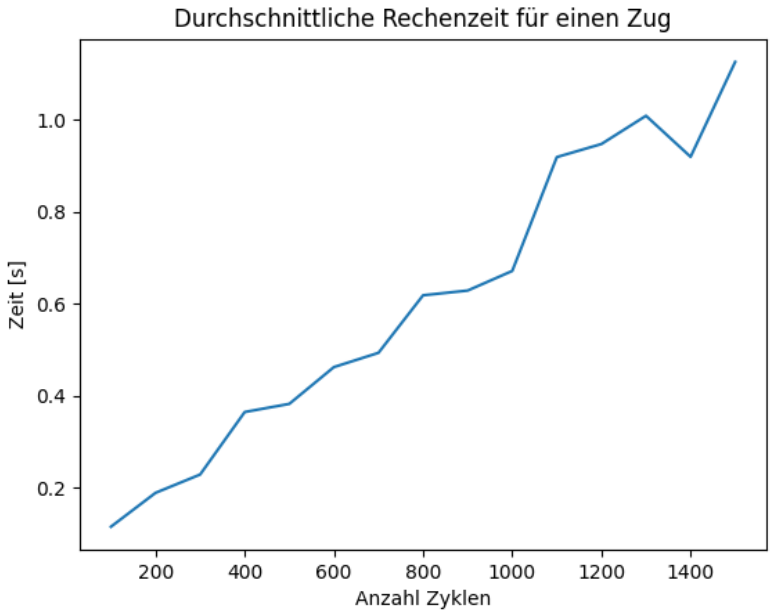


Abbildung 5.2: Vergleich der durchschnittlichen Rechenzeit eines Zuges des MCTS-Algorithmus für unterschiedlich hohe Zyklenanzahlen

Um die Gewinnchancen von MCTS für verschiedene Zyklenanzahlen zu evaluieren, lassen wir diesen gegen Random und Alpha-Beta mit Suchtiefe 2,4,6 und 8 jeweils 100 mal spielen und betrachten die prozentuale Gewinnverteilung. Dabei variieren wir die Zyklenanzahl zwischen 200 und 1400 in 200er-Schritten. Das Ergebnis findet sich in Tabelle 5.3.

| | Random | | | Alpha-Beta(2) | | | Alpha-Beta(4) | | |
|---|---|---|---|---|---|---|---|---|---|
| Zyklen | G (%) | V (%) | U (%) | G (%) | V (%) | U (%) | G (%) | V (%) | U (%) |
| 200 | 100 | 0 | 0 | 82 | 17 | 1 | 75 | 19 | 6 |
| 400 | 100 | 0 | 0 | 87 | 12 | 1 | 88 | 9 | 3 |
| 600 | 100 | 0 | 0 | 90 | 10 | 0 | 92 | 8 | 0 |
| 800 | 100 | 0 | 0 | 94 | 6 | 0 | 92 | 5 | 2 |
| 1000 | 100 | 0 | 0 | 98 | 2 | 0 | 95 | 4 | 1 |
| 1200 | 100 | 0 | 0 | 95 | 5 | 0 | 92 | 7 | 1 |
| 1400 | 100 | 0 | 0 | 95 | 3 | 2 | 94 | 4 | 0 |

| | Alpha-Beta(6) | | | Alpha-Beta(8) | | |
|---|---|---|---|---|---|---|
| Zyklen | G (%) | V (%) | U (%) | G (%) | V (%) | U (%) |
| 200 | 73 | 24 | 3 | 74 | 21 | 5 |
| 400 | 83 | 12 | 5 | 75 | 22 | 3 |
| 600 | 94 | 6 | 0 | 85 | 13 | 2 |
| 800 | 95 | 4 | 1 | 89 | 10 | 1 |
| 1000 | 98 | 2 | 0 | 90 | 10 | 0 |
| 1200 | 94 | 4 | 2 | 93 | 5 | 2 |
| 1400 | 93 | 6 | 1 | 95 | 4 | 1 |

Tabelle 5.3: Prozentualer Anteil der Gewinne, verlorenen Runden sowie Unentschiedenspiele des MCTS-Algorithmus gegen einen Zufallsalgorithmus und Minimax für unterschiedliche Zyklenanzahl

Wir erkennen, dass der MCTS-Algorithmus im Gesamten sehr gut spielt. Gegen Random gewinnt der Algorithmus für alle untersuchten Zyklenanzahlen ohne Ausnahme. Auch dem Alpha-Beta-Pruning-Algorithmus scheint er überlegen zu sein. Zwar gibt es hier auch ein paar verlorene Runden (maximal 24%), im Gesamten ist ein Gewinn aber deutlich wahrscheinlicher.
Wir sehen auch, dass der Alpha-Beta-Pruning-Algorithmus mit zunehmender Suchtiefe mehr Spiele für sich entscheidet. Dies spricht dafür, eine höhere Suchtiefe für diesen Algorithmus zu wählen.
Die Zyklenanzahl des MCTS-Algorithmus scheint im Bereich von niedrigen Werten

(200-800) einen Einfluss auf die Gewinnchance zu haben. Je höher die Zyklenanzahl ist, desto höher ist auch die Gewinnchance gegen Alpha-Beta-Pruning.
Ab einem Wert von 1000 Zyklen steigen die Gewinnchancen nicht mehr und nehmen sogar leicht ab. Hierzu muss allerdings gesagt werden, dass 100 Spiele nicht ausreichen, um einen signifikanten Unterschied festzustellen, da sich die Gewinnchancen nur um ein paar Prozent unterscheiden.

Wir betrachten für den MCTS-Algorithmus ebenfalls, wieviele Züge dieser durchschnittlich benötigt, um zu gewinnen. Dafür berechnen wir für die gewonnenen Spiele gegen Random aus Tabelle 5.3 wieder die mittlere Anzahl der Züge, die Standardabweichung, sowie das 95%-Konfidenzintervall. Wir betrachten hier nur die Spiele gegen Random, um einen Vergleich zu der mittleren Anzahl Züge für den Alpha-Beta-Pruning-Algorithmus (Tabelle 5.2) zu erhalten. Die Ergebnisse sind in Tabelle 5.4 dargestellt.

| Zyklen | Mittlere Anzahl Züge bei Gewinn (SD) | 95%-KI |
|---|---|---|
| 200 | 15,94 (4,47) | [15,06, 16,82] |
| 400 | 15,11 (4,80) | [14,16, 16,06] |
| 600 | 13,77 (4,08) | [12,97, 14,57] |
| 800 | 13,78 (3,84) | [13,02, 14,54] |
| 1000 | 13,79 (3,92) | [13,02, 14,56] |
| 1200 | 14,55 (3,56) | [13,85, 15,25] |
| 1400 | 13,94 (3,63) | [13,22, 14,66] |

Tabelle 5.4: Statistische Auswertung der Rundenanzahl für gewonnene Spiele des MCTS-Algorithmus gegen einen Zufallsalgorithmus für unterschiedliche Zyklenanzahl

Wir erkennen einen (mit leichten Schwankungen) absteigenden Verlauf für zunehmende Zyklenzahlen. Dies bedeutet, je mehr Zyklen der MCTS-Algorithmus durchläuft, desto schneller kann er ein Spiel gewinnen.
Vergleichen wir diese Ergebnisse mit der mittleren Anzahl an Zügen der gewonnenen Runden des Alpha-Beta-Pruning-Algorithmus gegen Random, erkennen wir, dass der MCTS-Algorithmus im Gesamten wesentlich weniger Züge benötigt. Das Minimum von 16,81 für Alpha-Beta aus Tabelle 5.2 ist höher als jeder Eintrag für MCTS aus Tabelle 5.4. Auch die Standardabweichung ist für MCTS um einiges niedriger, was für weniger Ausreißer (extrem hohe oder niedrige Werte) spricht.

## 5.3 Evaluierung der Deep Q-Learning Netzwerke

In diesem Kapitel möchten wir die Rechenzeit, Gewinnchancen und Rundenanzahl der trainierten Deep Q-Netzwerke untersuchen.
Da die Netzwerke aus gleich vielen Schichten und Neuronen bestehen und somit immer die selbe Anzahl an Rechenoperationen ausgeführt wird, ist hinsichtlich der Rechenzeit kein Vergleich der Netzwerke untereinander notwendig. Wir ermitteln einen einheitlichen Wert, indem wir die Rechenzeit für einen Zug in einem Testspiel messen und vergleichen diesen mit den Rechenzeiten der anderen KI-Algorithmen. Der gemessene Wert beträgt 0.0517 Sekunden, damit ist der Deep Q-Learning Algorithmus wesentlich schneller als MCTS und Alpha-Beta-Pruning.

Um die Gewinnchancen zu ermitteln, lassen wir die trainierten Netzwerke gegen den Zufallsalgorithmus Random, den Alpha-Beta-Pruning-Algorithmus mit Tiefe 2,4,6 und 8, sowie gegen den MCTS-Algorithmus mit 200,600,1000 und 1400 Zyklen spielen. Für jede Konstellation werden 100 Spiele gespielt und die Anzahl gewonnener, verlorener und unentschiedener Spiele prozentual angegeben.
Hierbei beschreibt *DQN-Random* das in Kapitel 4.5.2 definierte Netzwerk aus Trainingsschritt (i) und *DQN-Perfekt2* bis *DQN-Perfekt6* die Netzwerke aus Trainingsschritt (ii). Das in Trainingsschritt (iii) definierte Netzwerk weist keinen guten Lernfortschritt auf und wird aus diesem Grund nicht betrachtet. Die Ergebnisse sind in den Tabellen 5.5 - 5.7 aufgelistet. Die Auswertung für die Anzahl an Zügen in Gewinnrunden findet sich in Tabelle 5.8.

| | Random | | |
|---|---|---|---|
| Netzwerk | G (%) | V (%) | U (%) |
| DQN-Random | 81 | 12 | 7 |
| DQN-Perfekt2 | 70 | 27 | 3 |
| DQN-Perfekt4 | 79 | 20 | 1 |
| DQN-Perfekt6 | 88 | 12 | 0 |

Tabelle 5.5: Prozentuale Anzahl der Gewinne, verlorenen Runden sowie Unentschiedenspiele der Deep Q-Learning-Netze gegen einen Zufallsalgorithmus

| Netzwerk | Alpha-Beta(2) G (%) | V (%) | U (%) | Alpha-Beta(4) G (%) | V (%) | U (%) |
|---|---|---|---|---|---|---|
| DQN-Random | 0 | 100 | 0 | 0 | 100 | 0 |
| DQN-Perfekt2 | 100 | 0 | 0 | 0 | 100 | 0 |
| DQN-Perfekt4 | 0 | 100 | 0 | 100 | 0 | 0 |
| DQN-Perfekt6 | 0 | 100 | 0 | 0 | 100 | 0 |

| Netzwerk | Alpha-Beta(6) G (%) | V (%) | U (%) | Alpha-Beta(8) G (%) | V (%) | U (%) |
|---|---|---|---|---|---|---|
| DQN-Random | 0 | 100 | 0 | 0 | 100 | 0 |
| DQN-Perfekt2 | 0 | 100 | 0 | 0 | 100 | 0 |
| DQN-Perfekt4 | 100 | 0 | 0 | 100 | 0 | 0 |
| DQN-Perfekt6 | 100 | 0 | 0 | 0 | 100 | 0 |

Tabelle 5.6: Prozentuale Anzahl der Gewinne, verlorenen Runden sowie Unentschiedenspiele der Deep Q-Learning-Netze gegen Alpha-Beta-Pruning

| Netzwerk | MCTS(200) G (%) | V (%) | U (%) | MCTS(600) G (%) | V (%) | U (%) |
|---|---|---|---|---|---|---|
| DQN-Random | 0 | 99 | 1 | 0 | 99 | 1 |
| DQN-Perfekt2 | 6 | 94 | 0 | 0 | 100 | 0 |
| DQN-Perfekt4 | 13 | 87 | 0 | 3 | 97 | 0 |
| DQN-Perfekt6 | 7 | 93 | 0 | 5 | 93 | 2 |

| Netzwerk | MCTS(1000) G (%) | V (%) | U (%) | MCTS(1400) G (%) | V (%) | U (%) |
|---|---|---|---|---|---|---|
| DQN-Random | 0 | 100 | 0 | 0 | 100 | 0 |
| DQN-Perfekt2 | 6 | 94 | 0 | 4 | 96 | 0 |
| DQN-Perfekt4 | 3 | 97 | 0 | 1 | 99 | 0 |
| DQN-Perfekt6 | 1 | 99 | 0 | 1 | 99 | 0 |

Tabelle 5.7: Prozentuale Anzahl der Gewinne, verlorenen Runden sowie Unentschiedenspiele der Deep Q-Learning-Netze gegen MCTS

Wir sehen in Tabelle 5.5, dass alle Netzwerke gegen den Zufallsalgorithmus eine gute Gewinnchance haben (circa 80%). Allerdings sind diese Werte geringer als die Gewinnchancen des Alpha-Beta-Pruning- und MCTS-Algorithmus. Dieses Verhalten lässt sich auf zwei mögliche Gründe zurückführen: Durch die vielen möglichen Zustände des Spiels bedarf es eines intensiveren Trainings, die 150.000 Iterationen sind nicht ausreichend genug. Ein weiterer möglicher Grund ist auch, dass die

gewählte Modellstruktur nicht ausreicht, um alle Spielsituationen abzudecken, das Modell ist also nicht weiter generalisierbar.
Da sowohl die Deep Q-Learning-Netze, als auch der Alpha-Beta-Pruning-Algorithmus deterministisch sind, führt ein Spiel zwischen diesen beiden Strategien immer zum gleichen Ergebnis. Aus diesem Grund sind die Gewinnchancen in Tabelle 5.6 immer 0% oder 100%. Auffällig ist, dass die Netzwerke, welche mit der Umgebung trainiert wurden, in welcher der entsprechende Minimax- bzw- Alpha-Beta-Pruning-Algoritmus als Gegner fungierte, immer gegen diesen gewinnen. Ansonsten sind die Gewinnchancen des Deep Q-Learnings gegen Alpha-Beta schlecht.
Dieses Verhalten lässt sich dadurch erklären, dass die Agenten so trainiert wurden, dass sie gegen genau eine Gegnerstrategie gewinnen. Da Alpha-Beta aber deterministisch ist und für gleiche Spielzustände exakt die selben Züge ausführt, muss das Netzwerk einen Umgang mit nur wenigen Spielzuständen lernen. Trifft das Netzwerk dann auf einen anderen Gegner, führt das zu neuen, unbekannten Spielzuständen, die es noch nicht beherrscht.
Ein Zusammenspiel der genannten Gründe führt auch dazu, dass die Gewinnchancen gegen MCTS sehr schlecht sind. Dies ist in Tabelle 5.7 zu erkennen. Da MCTS der bisher stärkste Algorithmus ist, ist dieses Ergebnis nicht verwunderlich.
Die in Tabelle 5.8 dargestellte Auswertung der mittleren Anzahl an Zügen bei Gewinnrunden gegen einen Zufallsalgorithmus zeigt, dass der Deep Q-Learning Algorithmus sehr viele Züge benötigt, um zu gewinnen. Da die Agenten beim Training nicht bestraft werden, wenn sie zu viele Züge ausführen, und nur dann belohnt werden, wenn sie viele Steine kapern und gewinnen, liegt der Fokus des Agenten darauf, eine möglichst große Anzahl an Steinen zu kapern. Dies benötigt eine erhöhte Anzahl an Zügen.

| Netzwerk | Mittlere Anzahl Züge bei Gewinn (SD) | 95%-KI |
|---|---|---|
| DQN-Random | 22,59 (9,69) | [20,47, 24,72] |
| DQN-Perfekt2 | 24,43 (10,04) | [22,06, 26,80] |
| DQN-Perfekt4 | 29,52 (12,49) | [26,75, 32,29] |
| DQN-Perfekt6 | 31,84 (11,85) | [29,36, 34,34] |

Tabelle 5.8: Statistische Auswertung der Rundenanzahl für gewonnene Spiele des DQN-Algorithmus gegen einen Zufallsalgorithmus für unterschiedliche Netze

## 5.4 Vergleich mit Policy Gradient Optimierung

Wir führen die selben Auswertungen wie in den vorigen Abschnitten durch. Da die Ausgabe des AC-Algorithmus mit Hilfe des actor-Netzwerks berechnet wird, ist die Rechenzeit, wie bereits im Kapitel zu den Deep Q Learning-Netzen beschrieben, auf Grund der selben Anzahl an Rechenoperationen immer gleich. Auch hier berechnen wir die Rechenzeit nur für einen Zug. Diese beträgt 0,0391 Sekunden und ist damit, ähnlich wie die DQN-Netze, um einiges schneller als MCTS, Alpha-Beta-Pruning und Minimax. Die Ergebnisse für die Gewinnchancen finden sich in Tabelle 5.9.

| AC-Algorithmus vs. | G (%) | V (%) | U (%) |
|---|---|---|---|
| Random | 80 | 15 | 5 |
| Alpha-Beta(2) | 0 | 100 | 0 |
| Alpha-Beta(4) | 0 | 100 | 0 |
| Alpha-Beta(6) | 0 | 100 | 0 |
| Alpha-Beta(8) | 0 | 100 | 0 |
| MCTS(200) | 1 | 97 | 2 |
| MCTS(600) | 0 | 100 | 0 |
| MCTS(1000) | 0 | 100 | 0 |
| MCTS(1400) | 0 | 100 | 0 |
| DQN-Random | 0 | 100 | 0 |
| DQN-Perfekt2 | 100 | 0 | 0 |
| DQN-Perfekt4 | 0 | 100 | 0 |
| DQN-Perfekt6 | 0 | 100 | 0 |

Tabelle 5.9: Prozentuale Anzahl der Gewinne, verlorenen Runden sowie Unentschiedenspiele des AC-Algorithmus gegen Random, Alpha-Beta, MCTS und DQN

Der AC-Algorithmus scheint eine nicht zufällige Strategie gelernt zu haben, da die Gewinnchance gegen Random 80% beträgt. Er ist mit dieser Strategie den anderen KI-Algorithmen jedoch nicht gewachsen, da im Training nur die *environmentRandom*-Umgebung verwendet wurde. Ein Training gegen andere Algorithmen blieb aus oder war nicht erfolgreich (siehe Kapitel 4.6).
Da die Ausgaben von Alpha-Beta, den DQN-Netzen und dem actor-Netz von AC deterministisch sind, führen Spiele dieser Algorithmen gegeneinander immer zum selben Ergebnis. Damit lassen sich die Werte von 0% und 100% erklären. AC ge-

winnt in diesen Konstellationen nur gegen DQN-Perfekt2.

| Algorithmus | Mittlere Anzahl Züge bei Gewinn (SD) | 95%-KI |
|---|---|---|
| AC | 28,84 (11,10) | [26,39, 31,29] |

Tabelle 5.10: Statistische Auswertung der Rundenanzahl für gewonnene Spiele des AC-Algorithmus gegen einen Zufallsalgorithmus

Die in Tabelle 5.10 aufgelistete Auswertung für die mittlere Rundenanzahl gewonnener Spiele des AC-Algorithmus gegen Random zeigt ähnliche Werte wie bei DQN. Im Vergleich zu den Suchbaumverfahren sind bei den Reinforcement Learning-Algorithmen mehr Runden vonnöten, um ein Spiel zu gewinnen.

# 6 Zusammenfassung und Ausblick

In dieser Arbeit wurden die Suchbaumverfahren Minimax, Alpha-Beta-Pruning und Monte Carlo Tree Search, sowie die Reinforcement Learning-Algorithmen Q-Learning und Actor-Critic für die Anwendung auf das Spiel Oware Abapa untersucht. Die RL-Algorithmen wurden mit neuronalen Netzen ausgestattet. Alle Algorithmen wurden in der Theorie mit Beispielen aufgestellt, mathematisch untersucht und für diese eine passende Implementierungsmöglichkeit angegeben. Es wurde ein Vergleich der KI-Algorithmen, in welchem sie gegeneinander und gegen einen Zufallsalgorithmus spielten, durchgeführt. Dabei wurden die Rechenzeit, die Gewinnchance und die mittlere Anzahl Züge für gewonnene Spiele festgehalten. Die Ergebnisse wurden tabellarisch aufgeführt und interpretiert.

Bei Betrachtung der Ergebnisse stellt sich heraus, dass die Suchbaumverfahren hinsichtlich der Gewinnchance ein besseres Vorgehen darstellen als die Reinforcement Learning-Algorithmen. Dabei erzielt MCTS das beste Ergebnis. Ebenfalls benötigt dieser Algorithmus am wenigsten Züge um ein Spiel zu gewinnen.
Sowohl für MCTS als auch für die Adversarial Search Algorithmen steigt die Gewinnchance, wenn ihre Parameter (Zyklenanzahl und Suchbaumtiefe) einen höheren Wert annehmen.
Hinsichtlich der Rechenzeit lässt sich MCTS mit den Adversarial Search Algorithmen nicht vergleichen, da sie von verschiedenen Parametern abhängen. Man kann aber feststellen, dass die Rechenzeit von MCTS mit einem linearen Verlauf geringere Steigerungen beinhaltet als die Rechenzeit der Adversarial Search Algorithmen mit fakultativem Verlauf. Da Alpha-Beta-Pruning eine weniger rechenintensive und damit schnellere Version von Minimax ist, ordnen wir Minimax den letzten Platz der Suchbaumverfahren zu.

Die Ergebnisse der Reinforcement Learning-Algorithmen sind ernüchternd. Der einzige Punkt, in welchem sie den Suchbaumverfahren überlegen sind, ist die Tatsache, dass die Rechenzeiten für einen Zug um ein Vielfaches kleiner sind. Diese Eigenschaft reicht allerdings nicht aus, um den Anforderungen eines guten KI-Algorithmus für das Spiel Oware Abapa gerecht zu werden. Die mittleren Anzahlen

an Runden, die benötigt werden, um einen Gewinn des Spiels herbeizuführen, sind wesentlich höher als die hierzu gemessenen Anzahlen der Suchbaumverfahren. Zwar spielen beide Algorithmen gegen einen Zufallsalgorithmus gut, was bedeutet, dass ein Lerneffekt vorhanden ist, sobald die Algorithmen aber gegen eine Strategie spielen, gegen die sie nicht gelernt haben, erzielen sie sehr schlechte Ergebnisse. Dass die DQN-Perfekt-Algorithmen gegen Random zum größten Teil gewinnen, hängt damit zusammen, dass sie in zwei Schritten trainiert wurden, wovon der erste Schritt ein Training gegen den Zufallsalgorithmus war und erst im zweiten Schritt das Training gegen Minimax erfolgte.
Dieser Sachverhalt lässt sich auf das Problem zurückführen, dass im Standard-Reinforcement Learning mit nur einem Agenten der Gegner als Teil der Umgebung gesehen wird, und nicht als eigenständiger Agent, welcher ebenfalls lernfähig ist und unterschiedliche Strategien anwendet. Ein möglicher Ansatz für eine Verbesserung der Ergebnisse bietet das in [13] vorgestellte Prinzip von Multi-Agent Reinforcement Learning. Hierbei gibt es mehrere Agenten, welche eigene Rewards, Zustände und Aktionen berechnen, aber in einer gemeinsamen Umgebung miteinander agieren. Die Literatur hierzu stellt dieses Verfahren aber für ein Lernproblem, in dem die Agenten gleichzeitig agieren, wie beim bekannten Zwei-Personen-Spiel „Schere, Stein, Papier“, oder für ein Lernproblem, in welchem die Agenten auf ein gemeinsames Ziel hinarbeiten und nicht gegnerisch agieren, vor. Für Zwei-Personen-Spiele mit alternierenden Zügen ist dieses Verfahren dagegen noch nicht ausgeprägt. Reinforcement Learning mit nur einem Agenten ist ein gutes Verfahren für Spiele, die keinen aktiven Gegner beinhalten. Einige Beispiele hierfür wären „Snake“ oder „CartPole“. Für Zwei-Personen-Spiele wie Oware Abapa ist hiervon allerdings abzusehen.

Als KI-Algorithmus für das Spiel Oware Abapa eignet sich also von den betrachteten Fällen MCTS mit einer sehr hohen Zyklenanzahl (hier ca. 1000) am besten, darauf folgt der Alpha-Beta-Pruning-Algorithmus mit einer möglichst großen Suchbaumtiefe. Wir legen uns für die Anwendung auf das Spiel Oware Abapa, unter Berücksichtigung der zu benötigenden Rechenzeit, auf die Tiefe 8 fest.

# Literaturverzeichnis


[1] ANTHONY SO, W. S. ; NAGY, Z. : *The Applied Artificial Intelligence Workshop - Start Working with AI Today, to Build Games, Design Decision Trees, and Train Your Own Machine Learning Models*. Packt Publishing, Limited, 2020. – ISBN 9781800203730

[2] BROWNE, C. B. ; POWLEY, E. ; WHITEHOUSE, D. ; LUCAS, S. M. ; COWLING, P. I. ; ROHLFSHAGEN, P. ; TAVENER, S. ; PEREZ, D. ; SAMOTHRAKIS, S. ; COLTON, S. : A Survey of Monte Carlo Tree Search Methods. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), Nr. 1

[3] CASSIRER, A. ; BARTH-MARON, G. ; BREVDO, E. ; RAMOS, S. ; BOYD, T. ; SOTTIAUX, T. ; KROISS, M. : *Reverb: A Framework For Experience Replay*. `https://github.com/deepmind/reverb#readme`. Version: 2021. – abgerufen am 26.02.2023

[4] CAZENAVE TRISTAN, W. M. H. M. Teytaud Olivier O. Teytaud Olivier: *Monte Carlo Search : First Workshop, MCS 2020, Held in Conjunction with IJCAI 2020, Virtual Event, January 7, 2021, Proceedings / edited by Tristan Cazenave, Olivier Teytaud, Mark H. M. Winands*. 1st ed. 2021. Springer International Publishing, 2021. – ISBN 9783030894535

[5] CHARLES A. KAMHOUA, F. F. Christopher D. Kiekintveld K. Christopher D. Kiekintveld ; ZHU, Q. : *Game Theory and Machine Learning for Cyber Security*. John Wiley & Sons, Incorporated, 2021. – ISBN 9781119723912

[6] CHASLOT, G. ; BAKKES, S. ; SZITA, I. ; SPRONCK, P. : Monte-carlo tree search: A new framework for game ai. In: *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment* Bd. 4, 2008

[7] DA SILVA, I. N. ; SPATTI, D. H. ; FLAUZINO, R. A. ; LIBONI, L. H. B. ; REIS ALVES, S. F.: *Artificial neural networks*. Bd. 39. Springer, 2017

[8] ESPONDA, I. ; POUZO, D. : Equilibrium in misspecified Markov decision processes. In: *Theoretical Economics* 16 (2021), Nr. 2

[9] EVEN-DAR, E. ; MANSOUR, Y. ; BARTLETT, P. : Learning Rates for Q-learning. In: *Journal of machine learning Research* 5 (2003), Nr. 1

[10] HAO DONG, S. Z. Zihan Ding D. Zihan Ding: *Deep Reinforcement Learning*. 1. Springer Singapore, 2020. – ISBN 9789811540950

[11] HENZE, N. u. a.: *Stochastik für Einsteiger*. Bd. 8. Springer, 1997

[12] IOSIFIDIS, A. ; TEFAS, A. : *Deep Learning for Robot Perception and Cognition*. Elsevier Science & Technology, 2022. – ISBN 9780323885720

[13] LITTMAN, M. L.: Markov games as a framework for multi-agent reinforcement learning. In: *Machine learning proceedings 1994*. Elsevier, 1994

[14] MAURER, C. : *Ein strukturorientierter Aufbau der klassischen Zahlenbereiche*. Springer. – ISBN 9783662648865

[15] OWEN, G. : *Game Theory*. Emerald Group Publishing, 2013. – ISBN 9781781905081

[16] PEARL, J. : The solution for the branching factor of the alpha-beta pruning algorithm and its optimality. In: *Communications of the ACM* 25 (1982), Nr. 8

[17] POWELL, A. ; TEMPLE, O. : Seeding ethnomathematics with oware: Sankofa. In: *Teaching children mathematics* 7 (2001), Nr. 6

[18] PRECUP, D. ; SUTTON, R. S. ; SINGH, S. : Theoretical results on reinforcement learning with temporally abstract options. In: NÉDELLEC, C. (Hrsg.) ; ROUVEIROL, C. (Hrsg.): *Machine Learning: ECML-98*. Berlin, Heidelberg : Springer Berlin Heidelberg, 1998

[19] REINEFELD, A. : *Spielbaum-Suchverfahren*. Springer Berlin Heidelberg (Informatik-Fachberichte). `https://books.google.de/books?id=i9uiBgAAQBAJ`. – ISBN 9783642744136. – abgerufen am 26.02.2023

[20] SCHWENKER, F. : *Theorie neuronaler Netze*. – Vorlesung im Sommersemester 2022, Universität Ulm

[21] SILVER, D. ; TESAURO, G. : Monte-Carlo simulation balancing. In: *Proceedings of the 26th Annual International Conference on Machine Learning*, 2009

[22] SUTTON, R. S. ; BARTO, A. G.: *Reinforcement learning: An introduction*. 2. MIT press, 2018

[23] THE TF-AGENTS AUTHORS: *TensorFlow: Environments*. `https://www.tensorflow.org/agents/tutorials/2_environments_tutorial`.

Version: 2021. – Software available from tensorflow.org; abgerufen am 23.01.2023

[24] THE TF-AGENTS AUTHORS: *TensorFlow: Train a Deep Q Network with TF-Agents*. `https://www.tensorflow.org/agents/tutorials/1_dqn_tutorial`. Version: 2021. – Software available from tensorflow.org; abgerufen am 23.01.2023

[25] THE TF AUTHORS: *TensorFlow: Playing CartPole with the Actor-Critic method*. `https://www.tensorflow.org/tutorials/reinforcement_learning/actor_critic`. Version: 2022. – Software available from tensorflow.org; abgerufen am 02.02.2023

[26] WATKINS, C. J. ; DAYAN, P. : Q-learning. In: *Machine learning* 8 (1992), Nr. 3

# A Appendix

In diesem Abschnitt sind einige zusätzliche Diagramme abgebildet. Außerdem wird das Prinzip der Varianzreduktion erklärt.

## Q-Learning mit Gegner Minimax (Tiefe = 4 und 6)

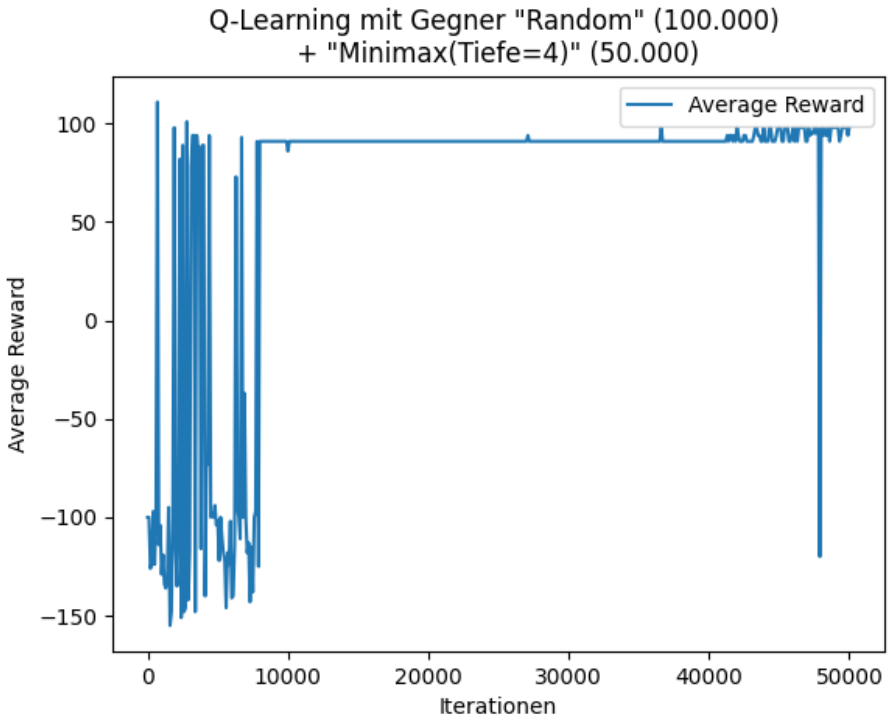


Abbildung A.1: Deep Q-Learning, Trainingsschritt (ii): Zusätzliche 50.000 Iterationen mit *environmentPerfect* (Tiefe = 4)

Der kurzzeitige Abfall des mittleren Rewards in Abbildung A.1 am Ende des Trainings ist durch das $\epsilon$-greedy Verfahren oder durch einen nicht erlaubten Zug auf Grund eines bisher unbekannten Zustands zu erklären.

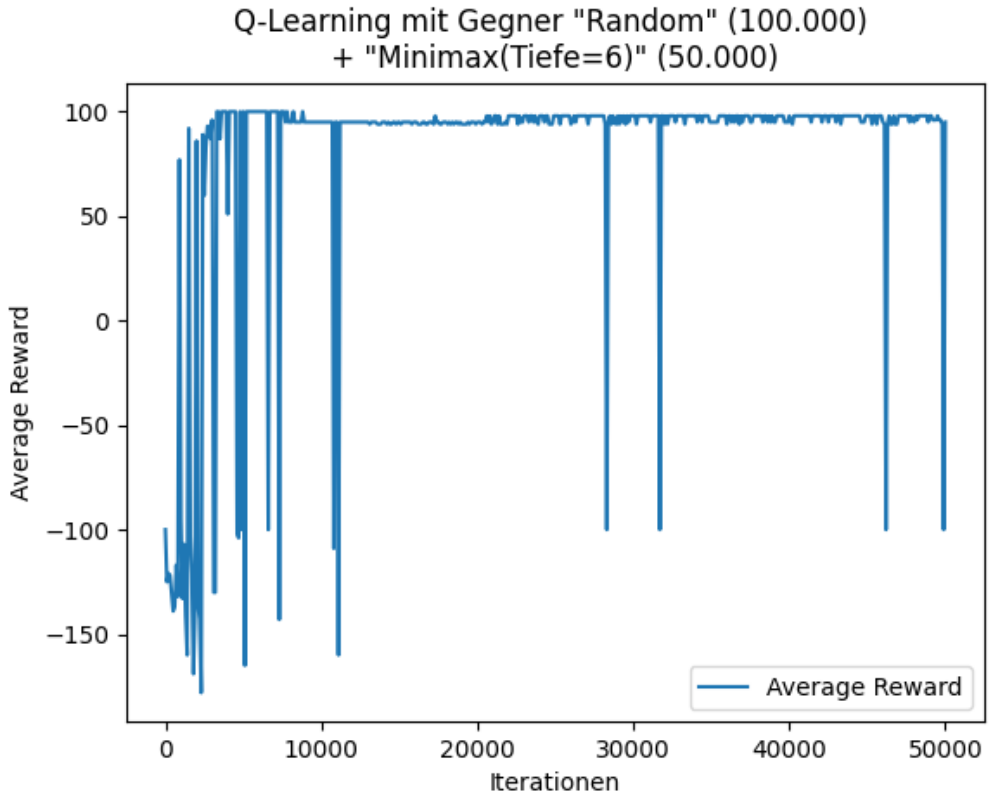


Abbildung A.2: Deep Q-Learning, Trainingsschritt (ii): Zusätzliche 50.000 Iterationen mit *environmentPerfect* (Tiefe = 6)

## Varianzreduktion

In Bemerkung 3.4.3.5 soll die Varianz des Gradientenschätzers mit Hilfe einer Baseline minimiert werden. Wir erklären anhand von [10], wie das Prinzip der Varianzreduktion funktioniert.

Sei $X$ eine Zufallsvariable, deren Erwartungswert $\mathbb{E}(X)$ geschätzt werden soll. Dafür eignet sich zum Beispiel der empirische Erwartungswert (das arithmetische Mittel) von $X$.
Sei $Y$ eine Zufallsvariable mit der Eigenschaft, dass $\mathbb{E}(Y) = 0$. Damit ist der empirische Erwartungswert von $X - Y$ ebenfalls ein guter Schätzer für $\mathbb{E}(X)$, denn

$$\mathbb{E}(X - Y) = \mathbb{E}(X) - \mathbb{E}(Y) = \mathbb{E}(X).$$

Für die Varianz von $X - Y$ gilt

$$\mathbb{V}(X - Y) = \mathbb{V}(X) + \mathbb{V}(Y) - 2cov(X, Y),$$

wobei $cov(X, Y)$ die Kovarianz zwischen $X$ und $Y$ ist.
Falls man $Y$ mit der Eigenschaft $\mathbb{E}(Y) = 0$ so findet, dass die Zufallsvariable eine geringe Varianz hat und mit $X$ stark korreliert (geringe Kovarianz), ist die Varianz von $X - Y$ geringer als die von $X$. Also kann der empirische Erwartungswert von $X-Y$ als ein guter Schätzer für $\mathbb{E}(X)$ mit einer geringerer Varianz gesehen werden. $Y$ ist für unseren Fall die Baseline.

Name: Daniela Hirsch Matrikelnummer: 942484

**Erklärung**

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Leinfelden, den 28.03.2023 Daniela Hirsch

Daniela Hirsch