Informatik
Hauptcampus

H O C H
S C H U L E
T R I E R

Künstliche Intelligenz für Tic-Tac-Toe und Vier Gewinnt

Artificial intelligence for Tic-tac-toe and Connect Four

David Popp, Philipp Schmitz
(Autoren alphabetisch nach Zunamen aufgelistet)

Hausarbeit

Betreuer: Prof. Dr. Christof Rezk-Salama

Trier, 31.08.2023

Inhaltsverzeichnis

1

Einleitung und Zielsetzung

Bei Tic-Tac-Toe und Vier Gewinnt handelt es sich um sogenannte Nullsummenspiele mit vollständiger Information. Weil den beteiligten Parteien nur eine endliche Zahl an Zügen zur Verfügung steht, terminiert jede Partie zwangsweise in einem Unentschieden oder einer der beiden Spieler gewinnt, wobei der andere verliert. Bei deterministischen Spielen existiert theoretisch immer eine Gewinnstrategie dadurch, dass von vorneherein alle Spielzustände bekannt sind. Folglich kann für jeden Zug die Menge der Folgezustände evaluiert und basierend auf dieser Berechnung eine optimale Aktion gewählt werden.

Wird bei Tic-Tac-Toe die Symmetrie des Spielbretts mit einbezogen und macht „X“ den ersten Zug, dann existieren 138 distinkte terminale Spielzustände – „X“ gewinnt bei 91 davon, „O“ ist bei 44 siegreich und drei enden in einem Unentschieden (ohne Beachtung der Symmetrie: 131.184 für X, 77.904 für O und 46.080 Unentschieden) [Bol13].

Vier Gewinnt, ist hingegen deutlich komplexer als Tic-Tac-Toe. Hier existieren bereits nach acht Zügen so viele Spielzustände wie bei Tic-Tac-Toe. Werden alle Anordnungen nach jeweils 0–42 Zügen aggregiert, so entsteht die Gesamtsumme aller Spielzustände von 4.531.985.219.092 [OEI12, A212693].

Bei Tic-Tac-Toe schafft es ein Mensch, die aktuelle Situation des Spielbrettes einzuschätzen und den optimalen Zug zu finden – bei Vier Gewinnt erfordert dies ähnlich wie bei Schach sehr viel Übung. An diesem Punkt ist es interessant zu erforschen, wie gut eine Künstliche Intelligenz trainiert werden kann, die beiden Spiele zu spielen.

Ziel dieser Arbeit ist es, Tic-Tac-Toe und Vier Gewinnt so zu implementieren, dass zwei Spieler und zwei Bots mit künstlicher Intelligenz in verschiedenen Konstellationen gegeneinander antreten können. Zum Evaluieren der Performanz sollten viele Spiele simuliert werden können. Zum Trainieren der Bots wird das sogenannte *Q-Learning* verwendet.

Bei diesem Algorithmus handelt es sich um einen modellfreien *Reinforcement-Learning-Algorithmus*. Das heißt, es wird direkt mittels Feedback aus Erfahrung oder Versuch-und-Irrtum gelernt.

1.1 Entwicklung

Die Wahl der Programmiersprache ist mit C# bereits vorgegeben. Es wurde sich an gegebenen Beispielen des Moduls „Künstliche Intelligenz für Spiele“ orientiert, wobei das Handout zu „Reinforcement Learning II “ die Grundlage dieser Software bildet (Klassen QLearning, QTable, GameState und Action). Aufgrund mangelnder Kenntnis zu grafischen Benutzeroberflächen wurde wie in den Handouts mit dem Framework WPF (*Windows Presentation Foundation*) gearbeitet.

Der aktuelle Spielzustand und zugehörige Attribute werden in der Klasse *GameState* verwaltet, welches das Interface *IGameState* implementiert. Aufgrund von Bedenken zur Performanz wurde hier nicht das *State Pattern* umgesetzt, obwohl dieses programmiertechnisch definitiv angebracht gewesen wäre.

Die Klasse *Game* implementiert *IGame* und bietet die eigentliche Schnittstelle, um mit dem Spiel zu interagieren. Diese verwaltet ihre zugehörige Benutzeroberfläche *GameGUI* und den *GameState*. Im Nachhinein (nachdem das Projekt „fertig“ implementiert war) wurde klar, dass es am sinnvollsten gewesen wäre, die Interfaces *IGameGUI*, *IGameState* und *IGame* jeweils in eigenen abstrakten Klassen zu implementieren, da sich viel Code bei einigen Methoden wiederholt – bei der Klasse *Game* unterscheidet sich zwischen der Implementierung von Tic-Tac-Toe und Vier Gewinnt sogar nur welcher GameState im Konstruktor instanziiert wird. So hätte in den abgeleiteten Klassen die Funktionalitäten, welche sich unterscheiden, einfach überschrieben werden können.

Es wurde versucht, darauf zu achten, dass alle Abhängigkeiten von Abstraktionen abhängen (Dependency-Inversion-Prinzip). Dies wurde jedoch nicht überall konsequent umgesetzt (siehe *MainWindow* ist eine Abhängigkeit in *GameGUI*).

Ansonsten sollten kleinere Entscheidungen zur Umsetzung der Software im Code auskommentiert sein.

1.2 GUI

Die folgende Abbildung stellt den finalen Stand der Benutzeroberfläche dar. Diese ist nicht besonders ausgereift und dient lediglich funktionalen Zwecken.

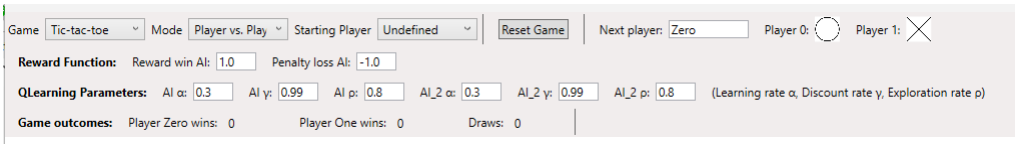


Abbildung 1.1: Finaler Stand der Benutzeroberfläche

Es wurden keine Design-Pattern bei der Entwicklung der GUI berücksichtigt.

2

Funktionsweise der Software

Die folgenden Abbildungen sind so nummeriert, dass bestimmte Teile des Bildes mit einer umkreisten Zahl referenziert werden.

2.1 Kontrollfläche für Spiele und Training

Die obere Leiste der GUI dient dazu, das gewünschte Spiel (1), den Modus (2) und welcher Spieler den ersten Zug macht (3), zu wählen. Bei dem Modus „Player vs. Player“ spielen zwei durch Benutzereingabe gesteuerte Parteien gegeneinander. Durch das Drücken der Maustaste in einem gewünschten Feld wird jeweils die Aktion ausgeführt. Dem Abschnitt (5) ist zu entnehmen, wer den nächsten Zug macht und welche Spielfigur zu diesem Spieler zugehörig ist.

Bei dem Modus „AI vs. AI“ spielen zwei computergesteuerte Bots gegeneinander. Diese sind standardmäßig untrainiert und können über die Kontrollfläche (6) trainiert werden. Nach Drücken des „Train“ Button spielen die beiden Bots so viele Partien gegeneinander, wie in *Num Iterations* definiert wurde.

Abbildung 2.1: Kontrollfläche für Spiele und Training

Bei Auswahl des Modus „Player vs. AI“ tritt ein benutzergesteuerter Spieler gegen einen Bot an. Hierbei ist zu beachten, dass der Spieler nur eine Aktion durchführt, wenn *Player Zero* den nächsten Zug macht. Ist *Player One* an der Reihe, erfolgt der Zug des Bots durch Drücken der Taste *M*. Zum Trainieren und Spielen wird der Bot verwendet, welcher mit den QLearning-Parametern unter „AI_2“ trainiert wurde.

Bei dem Modus „AI vs. Random“ tritt der Bot gegen einen anderen Bot an, welcher nur randomisierte valide Züge macht. In diesem Modus kann auch gleichzeitig der Bot gegen diesen trainiert werden. Was jedoch nicht funktioniert ist, dass einer

der beiden Bots gegen einen randomisierten Bot trainiert und anschließend gegen einen Bot trainierten antritt. Zum Trainieren und Spielen wird der Bot verwendet, welcher mit den QLearning-Parametern unter „AI_2“ trainiert wurde.

Mit dem *Reset Game* Button (4) wird das gesamte Spiel, sowie GUI zurückgesetzt.

2.2 Steuern der Reward Function

Mit der *Reward Function* (1) werden die Werte gesteuert, welche als Feedback für das Lernen dienen. Der Wert für ein Unentschieden kann nicht geändert werden und ist mit 0,5 hinterlegt.

Abbildung 2.2: Steuern der Reward Function

2.3 Steuern der QLearning-Parameter

Über die folgenden sechs Werte werden die QLearning-Parameter (1) gesteuert. Ist die Checkbox (2) aktiviert, wird das Lernen in vier Lernphasen eingeteilt. In den Lernphasen werden jeweils die folgenden Werte für α und ρ verwendet: 1.Phase (0,5/1,0), 2.Phase (0,4/0,7), 3.Phase (0,3/0,5) und 4.Phase (0,2/0,3). Ist die Checkbox nicht aktiviert, so werden die sechs Parameter unverändert für die gesamte Trainingsphase verwendet.

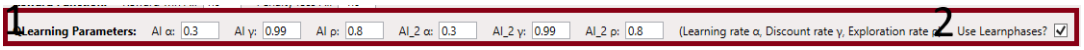


Abbildung 2.3: Steuern der QLearning-Parameter

2.4 Statistik der Ergebnisse

Der folgende Abschnitt (1) zeigt die Statistik für simulierte Spiele. Wird eine Partie durch zwei Spieler oder einem Bot und Spieler gespielt, so wird das Spielergebnis über ein Pop-up-Fenster mitgeteilt. Über den Input in Abschnitt (2) wird die Anzahl der zu simulierenden Spiele gesteuert. Mit der Checkbox (3) wird kontrolliert ob, die einzelnen Züge grafisch dargestellt werden sollen. Mit dem Button (4) wird die Simulation gestartet. Die nebenstehende Leiste zeigt den aktuellen Fortschritt in Echtzeit an.

Abbildung 2.4: Statistik der Ergebnisse

3

Reinforcement-Learning

Reinforcement-Learning zeichnet sich dadurch aus, dass über ein Versuch-und-Irrtum-Verfahren gelernt wird. Die Wahl einer optimalen Aktion in einem bestimmten Zustand zu approximieren. Dabei wird jede Aktion eines lernenden Agenten numerisch bewertet. Durch Ausprobieren versucht ein Agent herauszufinden, welche Aktion das beste Feedback erhält [Sut98].

3.1 Q-Learning

Q-Learning ist ein Algorithmus im Bereich des Reinforcement-Learning. Dabei wird eine Lookup-Tabelle (QTable) verwaltet, welche zu allen States (Spielzustände) und gültigen Aktionen eine zugehörige numerische Bewertung speichert. In jedem Trainingsschritt n, also einer ausgeführten Aktion, bekommt ein lernender Agent Feedback für den aktuell ausgeführten Zug. Anhand dieser Bewertung wird die QTable nach der folgenden Formel aktualisiert:

- aktueller Zustand x_n,
- ausgewählte Aktion a_n,
- nachfolgender Zustand y_n,
- Bewertung der Aktion r_n,
- Lernrate α und
- Discount-Rate γ:

$$Q_n(x,a) = \begin{cases} (1-\alpha_n)Q_{n-1}(x,a) + \alpha_n[r_n + \gamma V_{n-1}(y_n)], & \text{wenn } x = x_n \text{ und } a = a_n \\ Q_{n-1}(x,a), & \text{ansonsten,} \end{cases}, \tag{3.1}$$

mit

$$V_{n-1}(y) \equiv \max_b \{Q_{n-1}(y,b)\}. \tag{3.2}$$

Mit der Formel in 3.1 wird die Qualität einer gewählten Aktion α in einem Zustand x und dessen Folgezustandqualität berechnet. Anhand von Formel 3.2 lässt sich eines der Basis-Prinzipien ablesen, dass die Qualität eines Zustands so gut ist, wie die beste Aktion, die in diesem Zustand gewählt werden kann. Weiterhin existiert der Einflussfaktor ρ, welcher während des Trainings einen Zufallsschwellwert angibt, ob die beste bekannte oder eine zufällige Aktion gewählt wird [Sut98].

4

Evaluieren der Implementierung

Die Performanz eines trainierten Agenten gegen einen Agenten, welcher zufällige Züge macht, dient als Bewertungskriterium der Trainingsgüte. Dabei wird jeweils die Spielstatistik abhängig davon, wie viele Spiele zum Trainieren gespielt wurden, grafisch dargestellt. Um stochastische Abweichungen bei den Ausgängen einer Partie möglichst gering zu halten, spielen die Agenten jeweils immer 10000 Spiele gegeneinander. Ein Bot, der gegen zufällige Aktionen trainiert, kann in dieser Implementierung nicht gegen einen Bot antreten, welcher mit einem anderen Bot trainiert hat. Unterschieden wird ebenfalls, ob zwei Agenten gleichzeitig beim Spielen gegeneinander trainiert wurden oder gegen einen Agenten, welcher zufällige Züge macht. Zusätzlich wird unterschieden, ob die Q-Learning Parameter während des Trainings konstant blieben oder in Lernphasen eingeteilt wurden.

Mit vier gleich großen Lernphasen:

- Phase 1: $\alpha = 0.5, \gamma = 0.99, \rho = 1.0$,
- Phase 2: $\alpha = 0.4, \gamma = 0.99, \rho = 0.7$,
- Phase 3: $\alpha = 0.3, \gamma = 0.99, \rho = 0.5$,
- Phase 4: $\alpha = 0.2, \gamma = 0.99, \rho = 0.3$.

Mit konstanten Parametern:

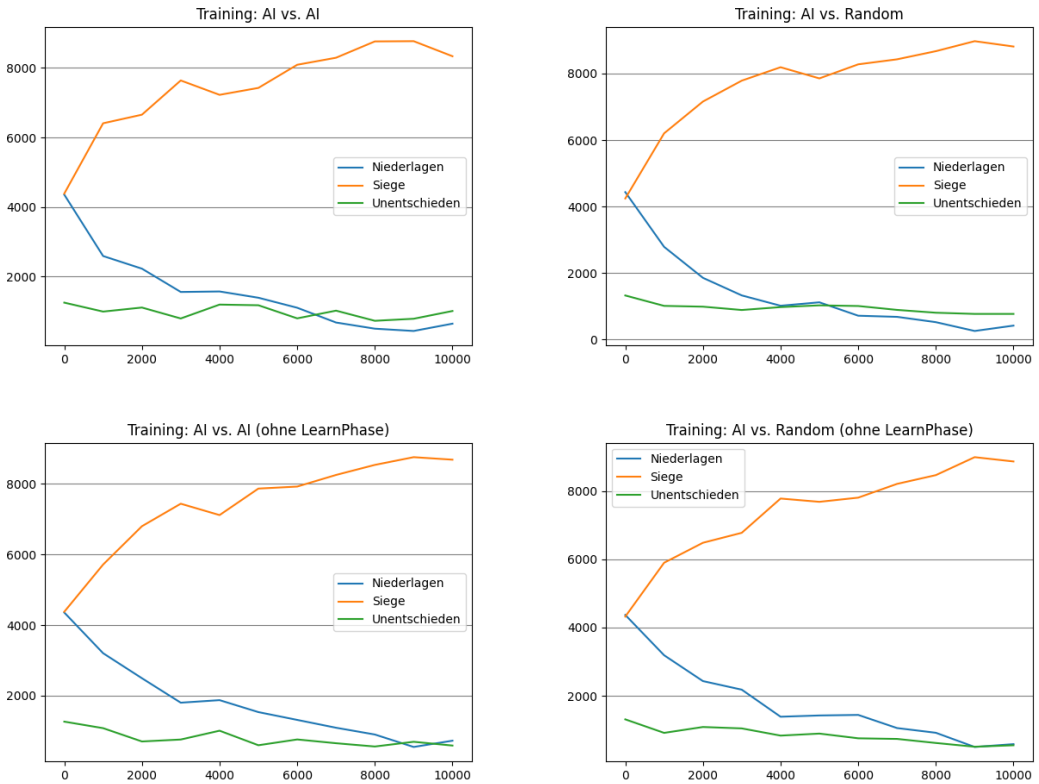
- $\alpha = 0.3, \gamma = 0.99, \rho = 0.8$.

4.1 Tic-Tac-Toe

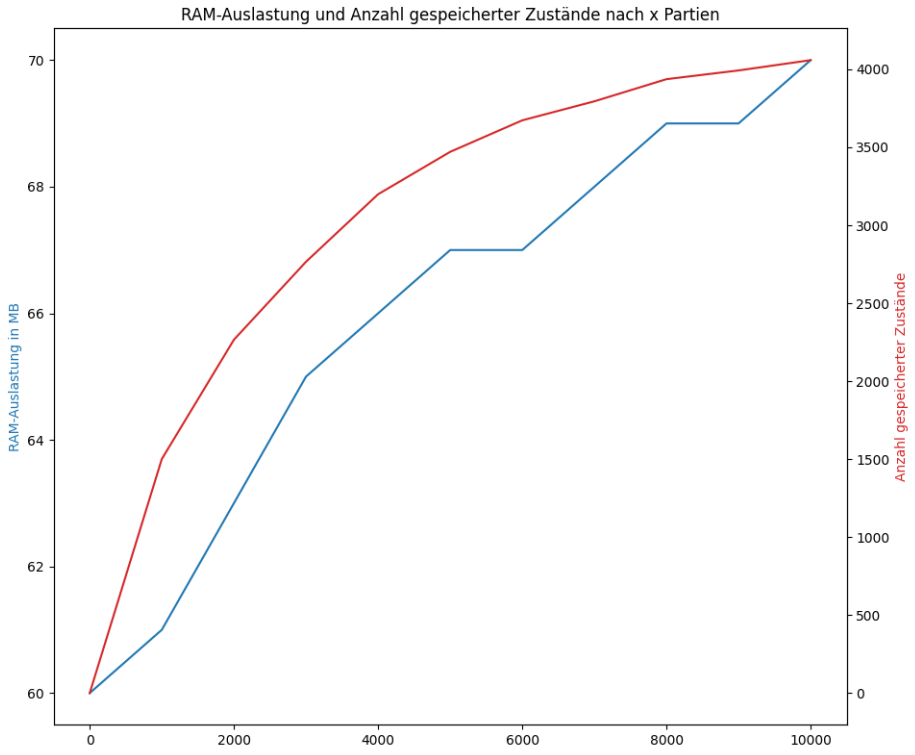
Anhand der folgenden Graphen ist zu erkennen, dass das Unterteilen des Trainings in vier Trainingsphasen keinen ausschlaggebenden Einfluss auf die Güte des trainierten Agenten hat. Wird mit konstanten Parametern trainiert, konvergiert der Graph anfangs langsamer. Ab 5.000 gespielten Partien ist jedoch ein ähnlicher Verlauf zu erkennen. Trainiert der Agent gegen einen Spieler, welcher zufällige Züge macht, so weist dieser auch in der Evaluation eine bessere Performanz auf. Hier wird vermutet, dass dadurch insgesamt mehr Zustände erreicht und somit bessere Entscheidungen bei der Wahl einer Aktion getroffen werden. Wird der Agent mit unterschiedlichen Lernphasen für mehr als 50.000 (wurde auch für 100.000 und 1.000.000 getestet) Spiele trainiert, so wird erreicht, dass dieser nahezu perfekt spielt ($\leq$ 10 Niederlagen).

Dies gilt für das Trainingsregiment „AI vs. AI“ und „AI vs. Random“ . Gleiches gilt jedoch nicht, wenn ohne Lernphasen trainiert wird ($\sim$ 30 Niederlagen bei „AI vs. AI“ und $\sim$ 50 „AI vs. Random“).

Wird das Trainingsregiment „AI vs. AI“ in $\geq$ 5.000 Trainingsspielen verwendet und spielen beide Bots anschließend gegeneinander, so endet jede Partie in einem Unentschieden.

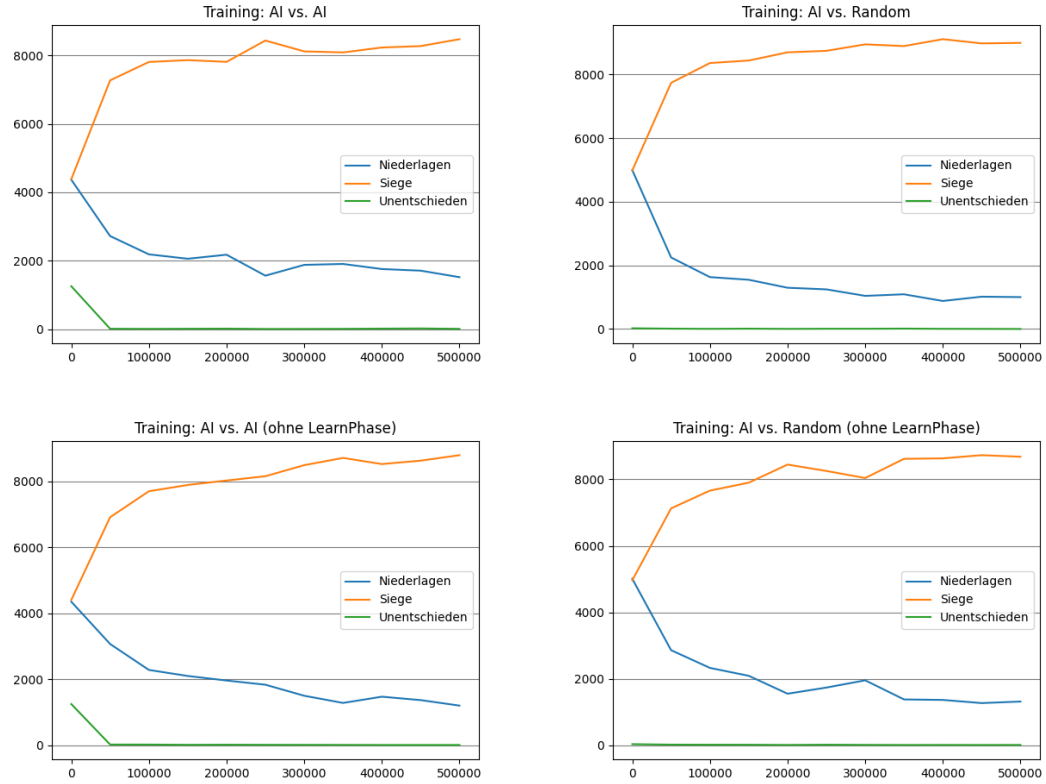


Die RAM-Auslastung übersteigt bei Tic-Tac-Toe nicht 70MB. Dies liegt an der sehr begrenzten Anzahl von Spielzuständen. Diese ist begrenzt auf den Wert 4.520.

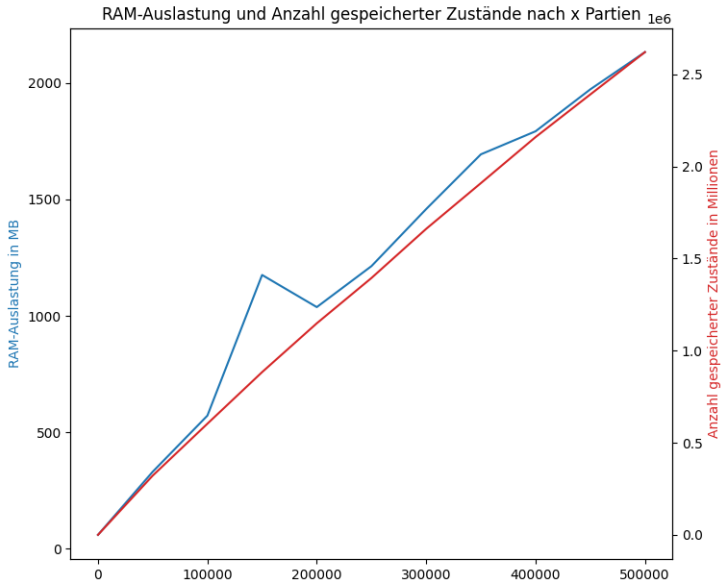


4.2 Vier Gewinnt

Bei Vier Gewinnt ist ein ähnliches Muster zu erkennen wie bei Tic-Tac-Toe. Das Trainieren mit mehreren Lernphasen hat zur Folge, dass der Graph anfangs schneller konvergiert – später jedoch eine vergleichbare Güte aufweist, wie das Trainieren ohne Lernphasen. Aufgrund der größeren Komplexität muss jedoch auch mit deutlich mehr Spielen trainiert werden. Bei dem Trainingsregiment „AI vs. Random“ ist zu erkennen, dass dieses ein wenig besser abschneidet als „AI vs. AI“. Vermutet wird, dass der Bot die Schwächen des zufälligen Spielen aufgrund der erlernten Erfahrung besser ausnutzen kann. Würde dieser jedoch gegen einen Bot antreten, welcher mit „AI vs. AI“trainiert wurde, ist es gut möglich, dass dieser („AI vs. Random“) deutlich mehr Partien verliert.



Die RAM-Auslastung ist im Falle von Vier Gewinnt ebenfalls deutlich höher. Wird bei „AI vs. AI“ mit fünf Millionen Spielen trainiert, so erreicht das Programm eine Speicherauslastung von mehr als 16 Gigabyte. Grund dafür ist, dass die *QTable* mehr als 20 Millionen Spielzustände verwalten muss. Weitere Statistiken sind in der beigelegten Excel-Tabelle „Statistiken.xlsx“ zu finden.


RAM-Auslastung und Anzahl gespeicherter Zustände nach x Partien
1e6
RAM-Auslastung in MB
Anzahl gespeicherter Zustände in Millionen
0
500
1000
1500
2000
0.0
0.5
1.0
1.5
2.0
2.5
0
100000
200000
300000
400000
500000

5

Probleme bei der Implementierung

Die meisten Probleme hat das Implementieren des Reinforcement-Learning-Algorithmus bereitet. Wenn auch die Implementierung des Handouts größtenteils übernommen wurde, gab es dennoch genügend Hürden. Eines davon war die Methode *MakeMove* der Klasse *QLearning*. Diese entspricht der *Step* Methode aus dem QLearning-Handout. Darin wurde nach jedem gemachten Zug die *QTable* entsprechend der Lernregel angepasst. Dies hat bei unserer Implementierung jedoch nicht wirklich funktioniert, da zwei *QTables* nach einem gemachten Zug angepasst werden müssen. Außerdem wurde eine anfällige Gewinnstrategie erlernt (Trainierter Agent hat immer versucht zu gewinnen, ungeachtet ob dieser im nächsten Zug verliert). Behoben wurde dies, indem die beiden *QTables* erst nach Beenden einer Partie aktualisiert und die vergangenen Züge temporär in einer Liste abgelegt wurden (siehe Methode *QLearning.UpdateTable*).

Ein weiterer Fehler war, dass die innere Schleife in der Methode zur Berechnung der StateID (*Connect_Four.GameState.Id*) nur sechs Iterationen lief anstelle von sieben, da die Methode aus der anderen *GameState*-Implementierung kopiert wurde. Dies hatte zur Folge, dass unterschiedliche Spielzustände auf die gleiche ID abgebildet haben und auch unter Umständen ungültige Züge in der *QTable* abgelegt wurden.

Zuletzt hat das Speichern des Schlüssel-Wert-Paares <IAction, double> in der *QTable* Probleme bereitet, da das Hinzufügen gleicher Aktionen einen neuen Eintrag in dem Dictionary nach sich zog. Erst nachdem eine zuvor initialisierte Liste von vorgegebenen Aktionen definiert wurde und diese in das Dictionary eingetragen wurden, gab es nur noch distinkte Aktionen innerhalb eines Objekts.

6

Zusammenfassung und Ausblick

6.1 Ausblick und Verbesserungen

Verbesserungen gibt es zu unserer Implementierung einige. Dazu zählt einmal das Implementieren des *IGame*-Interfaces in einer abstrakten Klasse. Des Weiteren sollte die *GameGUI* unabhängig von einer *MainWindow*-Implementierung sein und ebenfalls mit einem Interface kommunizieren. Zuletzt könnte vorausgesetzt werden, dass eine *IAction*-Implementierung die Methode *Equals* überschreibt, sodass Objekte nicht doppelt in ein Dictionary eingetragen werden, ohne, dass eine vorinitialisierte Liste verwendet werden muss.

Um die Arbeitsspeichernutzung zu optimieren, sollte die Verwendung des Datentyps *float*, statt *double* vorgezogen werden. Ebenfalls kann die Größe der QTable durch die Umsetzung von sogenannten „Afterstates“ verringert werden. Zuletzt sollte es noch ermöglicht werden, dass ein Agent, welcher während Trainings gegen zufällige Züge gespielt hat, gegen einen Agenten antritt, welcher gegen einen anderen trainierenden Agenten gespielt hat.

Weitere Verbesserungen sind im Quellcode auskommentiert beziehungsweise mit einem „TODO:“ gekennzeichnet.

6.2 Fazit und Zusammenfassung

Zusammenfassend war das Erlernen Tic-Tac-Toe und Vier Gewinnt zu spielen, erfolgreich. Die trainierten Bots haben nach ausreichendem Training Spielverständnis gezeigt. Es war vorher bereits klar, dass die Bots bei Tic-Tac-Toe deutlich besser abschneiden werden. Bei Vier Gewinnt konnte nicht mit mehr als 5.000.000 Spielen trainiert werden, da unsere Hardware ein begrenzender Faktor war. Es wird aber vermutet, dass die Bots mit ausreichend Training bei Vier Gewinnt nahezu perfekt spielen können. Eine weitere Erkenntnis ist, dass die Bots für beide Spiele dieselbe Lern-Architektur wiederverwenden können.

Literaturverzeichnis

Bol13. Bolon, Thomas: *How to never lose at Tic-Tac-Toe* -. BookCountry, 2013.

OEI12. OEIS Foundation Inc.: *Entry A212693 in The On-Line Encyclopedia of Integer Sequences*, 2012. Published electronically at `http://oeis.org`.

Sut98. Sutton, Richard S.: *Reinforcement learning*. MIT Press, 1998.

A

Glossar

DisASTer	Distributed Algorithms Simulation Terrain, eine Plattform zur Implementierung verteilter Algorithmen [**?**]
DSM	Distributed Shared Memory
AC	Atomic Consistency (dt.: Linearisierbarkeit)
RC	Release Consistency (dt.: Freigabekonsistenz)
SC	Sequential Consistency (dt.: Sequentielle Konsistenz)
WC	Weak Consistency (dt.: Schwache Konsistenz)