Angewandte KI

Umsetzung eines Tetris spielenden Agenten mittels Bestärkendem Lernen

Wintersemester 2018/2019

Prof. Dr. Jörg Frochte

Heiligenhaus, den 22. März 2019

Stefan Bäunker
Matrikelnr. 013-200-034

Simon Lindhorst
Matrikelnr. 014-215-586

Inhaltsverzeichnis

1. Einleitung

Im Rahmen des Wahlfachs „Angewandte Künstliche Intelligenz“ im Wintersemester 18/19 wurde mittels Bestärktem Lernen ein lernender Agent umgesetzt, der das populäre Computerspiel Tetris erlernen kann. Das entsprechende Tetris-Spiel wurde im Rahmen dieses Projektes programmiert, wobei sich an dem klassischen Tetris-Spiel orientiert wurde (siehe [1]).

1.1. Tetris

Das klassische Tetris ist ein Einzelspielerspiel, bei dem Bausteine verschiedener Form, so genannte Tetrominos, in einem Spielfeld gestapelt werden müssen. Tetrominos treten in sieben verschiedenen Varianten auf, die an die Formen der Buchstaben I, J, L, S, T, O und Z angelehnt sind. Jeder Tetromino besteht wiederum aus 4 Blöcken. Die Größe des Spielfelds ist begrenzt und beträgt klassischerweise 10x22 Blöcke. Die Aufgabe des Spielers ist es, die von oben herabschwebenden Tetrominos durch Rotation und Translation so im Spielfeld zu platzieren, dass sich möglichst vollständige horizontale Reihen ergeben. Ist eine horizontale Reihe vollständig gefüllt, so wird diese gelöscht und die darauf gestapelten Tetrominos fallen, nach der aus dem Alltag gewohnten Physik, nach unten. Wird höher gestapelt als das Spielfeld hoch ist, hat der Spieler verloren.

1.2. Motivation

Es ist bewiesen, dass die beim Tetris-Spiel zu lösenden Probleme, selbst bei bekannter Sequenz der Tetrominos NP-Vollständig sind. Das heißt, dass die Lösungen der Probleme in Polynomialzeit verifiziert werden können, allerdings keine neuen Lösungen in Polynomialzeit gefunden werden können (siehe [2]). Dennoch existieren für die nicht deterministische Version, also bei unbekannter Tetromino-Sequenz, diverse KI-Implementierungen welche die menschliche Leistung deutlich überschreiten (siehe [3]).

Implementierungen die Bestärkendes Lernen ohne die Verwendung von tiefen neuronalen Netzen benutzen, erreichen die entsprechende Leistung oft nicht oder erfordern eine Reduktion der Problemgröße um effektiv zu lernen (siehe [4],[5]). Es ist davon auszugehen, dass eine noch zu definierende, optimale Lösung mit Bestärkendem Lernen, dem originalen Spiel und einem realistischen Arbeits- und Zeitaufwand nicht umzusetzen ist. Die Optimierung der Implementierung und die Analyse des Fortschritts über den Projektzeitraum bieten allerdings das Potential der umfassenden Auseinandersetzung mit der Thematik des Bestärkenden Lernens.

1.3. Aufgabenstellung

Das Ziel dieses Projektes ist die Implementierung eines lernenden Agenten, der mittels Bestärkenden Lernens das Spiel Tetris erlernt. Im Verlaufe dieser Arbeit sollen dazu verschiedene Lösungsansätze untersucht und ein entsprechendes Lösungskonzept für das Spiel und den Agenten erstellt und implementiert werden. Außerdem soll die Güte der eigenen Implementierung untersucht und gegebenenfalls optimiert werden. Die Umsetzung und Optimierung eines Q-Learning Agenten soll mit dem didaktischen Prozess verknüpft werden. Außerdem soll die Relevanz einzelner Parameter und Methoden für die konkrete Implementierung untersucht werden.

2. Grundlagen „Bestärkendes Lernen“

In diesem Kapitel werden einige grundlegende Aspekte des Bestärkenden Lernens (Reinforcement Learning) erläutert, die im weiteren Verlauf relevant sind. Dabei wird hauptsächlich auf die Bücher [6] und [7] Bezug genommen.

Komplexe Systeme können häufig gar nicht oder nur sehr aufwendig modelliert werden. Dagegen können die Zustände und ausgeführten Aktionen ggf. „einfacher“ bewertet werden. Bei einem derartigen System ist ein Verfahren sinnvoll, dass dessen Modell durch das Ausführen von Aktionen auf dem System und den entsprechenden Rückmeldungen erlernt. Das *Reinforcement Learning* ist ein solches Verfahren.

Grundsätzlich handelt es sich um eine Vorgehensweise, bei der sich ein Agent ein bestimmtes Verhalten an trainiert, ohne dass ein „Lehrer“ benötigt wird. Somit gehört das Bestärkende Lernen zu den selbst lernenden Verfahren. In dem Buch „Neural Networks and Deep Learning“ [6] wird der Reinforcement-Ansatz als ein belohnungsgesteuertes try-and-error Verfahren beschrieben, das eine möglichst hohe Belohnung (reward) erzielen möchte:

> *„A reward-driven trial-and-error process, in which a system learns to interact with a complex environment to achieve rewarding outcomes [...]“*
> [6] S. 374

Im Gegensatz zum Überwachten Lernen, agiert der Agent beim Bestärkenden Lernen selbständig und lernt aufgrund der Rückmeldungen des Systems auf dem der Agent Aktionen ausführt. Der Agent versucht im Laufe seines Lernprozesses die Belohnung zu maximieren (siehe [6] S. 375).

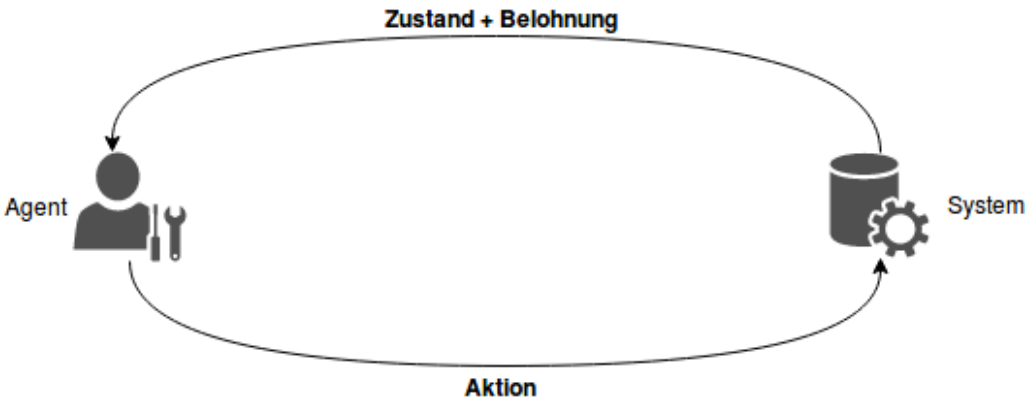

Abbildung 1: Übersicht: Reinforcement Learning (i.A.a [6] S. 378)

Die theoretische Grundlage des Bestärkenden Lernens basiert auf dem sogenannten *Markow-Entscheidungsprozess* (Markov decision process) (vgl. [8]). Es handelt sich um einen in der Informatik weit verbreiteten mathematischen Ansatz und wird in einem folgenden Kapitel näher erläutert.

2.1. Der Agent

Im Rahmen dieser Arbeit ist mit einem Agenten eine Software gemeint, die, wenn sie gestartet wurde, ein eigenständiges bzw. eigendynamisches Verhalten aufweist. Es gibt unterschiedliche Klassifizierung.

Ein Ansatz aus [9] beschreibt folgende Klassen:

- Einfache reaktive Agenten
- Beobachtende Agenten
- Zielbasierte Agenten
- Nutzenbasierte Agenten
- Lernende Agenten

Nach [7] kann durch diese Klassifizierung nicht die Fähigkeit der Zusammenarbeit zwischen Agenten beschrieben werden. Dazu werden drei weitere Attribute eingeführt:

robust - der Agent kompensiert (teilweise) äußere und innere Störungen

kognitiv - der Agent ist auf der Basis eigener Entscheidungen und Beobachtungen lernfähig

sozial - der Agent kommuniziert mit anderen Agenten

(siehe [7] S. 331)

Für das Bestärkende Lernen ist hauptsächlich die Klasse der *kognitiven lernenden Agenten* wichtig. Ein Schema dieser Agentenklasse ist in Abbildung 2 dargestellt.

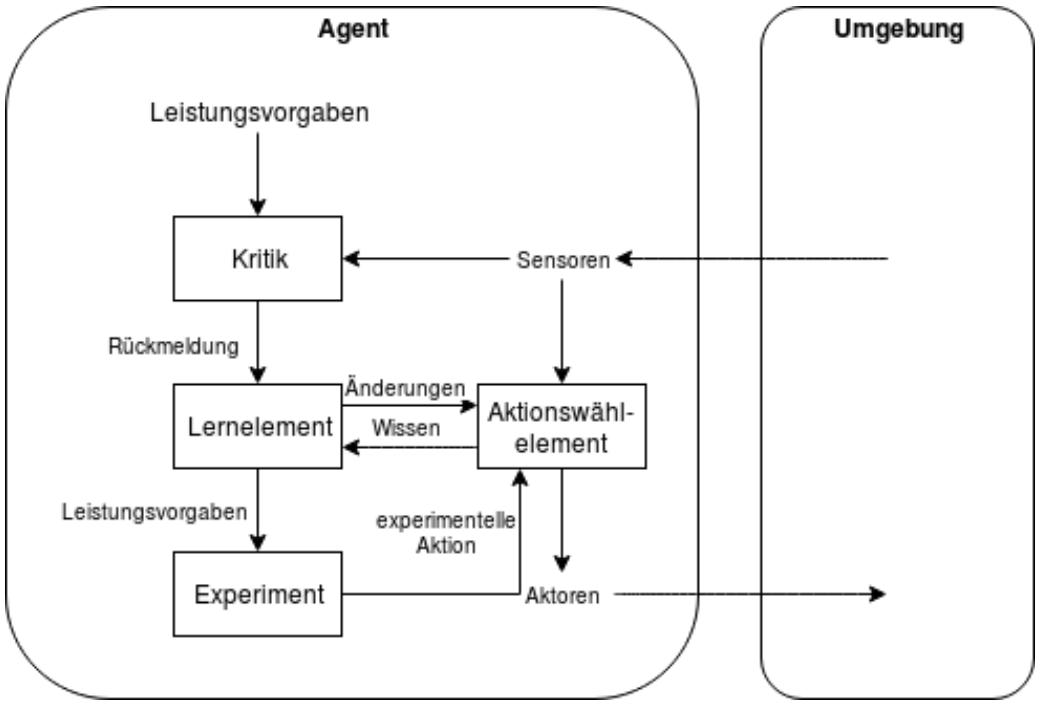

Abbildung 2: Schema eines Lernenden Agenten *(i. A. a. [7] S. 333)*

Die **Leistungsvorgaben** werden von *außen* vorgegeben und beinhalten die zu erreichenden Ziele. Das

Kritik-Modul vergleicht diese Vorgaben mit den Rückgabewerten der **Sensoren**, die bestimmte Größen der Umwelt aufnehmen. Aus dem Vergleich resultiert eine Rückmeldung die an das **Lernelement** weitergeleitet wird. Aufgrund dieser Rückmeldung verändert das Lernelement das Aktionswählelement. Wie der Name schon andeutet ist das **Aktionswählelement** für die konkrete Auswahl der vom Agenten durchzuführenden Aktion zuständig, um auf die Umgebung zu reagieren. Grundsätzlich wird immer die Aktion ausgewählt, die aufgrund des aktuellen Wissensstands am geeignetsten ist, um die Leistungsvorgaben zu erfüllen. Im optimalen Fall wird der Agent so reagieren, dass die vorgegebenen Ziele erfüllt werden. An dieser Stelle könnte man im ersten Moment davon ausgehen, dass nicht mehr Elemente für einen lernenden Agenten notwendig seien. Das folgende Beispiel soll zeigen, dass noch ein weiteres Element benötigt wird:

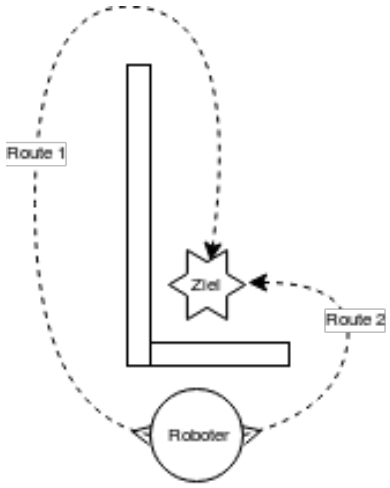

Abbildung 3: Beispiel: Lernender Agent

In Abbildung 3 ist ein Roboter (Kreis) dargestellt, der zu einem Ziel (Stern) fahren soll. Ein Hindernis versperrt den direkten Weg. Nun soll der Roboter lernen möglichst schnell an sein Ziel zu gelangen. Zu Beginn fährt er links um das Hindernis herum und findet schließlich über die *Route 1* sein Ziel. Da er von keinem schnelleren Weg weiß, wird er immer wieder die *Route 1* fahren, da ihn diese nach seinem Wissensstand am schnellsten zum Ziel bringt. Damit der Roboter aber auch die *Route 2* findet, muss er entgegen seines bisherigen Wissensstandes andere Wege ausprobieren. Nur so wird er schließlich auch die andere Route zum Ziel finden.

Damit ein Lernender Agent die vorgegebenen Ziele möglichst gut erfüllt, darf er bei der Aktionswahl nicht immer die nach seinem Wissensstand beste Aktion auswählen, sondern muss andere Aktionen ausprobieren. Für dieses Ausprobieren ist das **Experimentierelement** zuständig. Dieses beeinflusst die Aktionswahl so, dass hin und wieder nicht die vermeintlich besten Aktionen für eine Situation ausgewählt werden, sondern solche, die das Experimentierelement vorschlägt. Ein einfaches Experimentierelement könnte die Aktionswahl z. B. so beeinflussen, dass mit einer bestimmten Wahrscheinlichkeit eine zufällige Aktion, ansonsten die nach dem Kenntnisstand des Agenten beste Aktion ausgewählt wird. Das Kritikelement wird im Anschluss feststellen, ob die gewählte Aktion sinnvoll war und die Aktionswahl dementsprechend verändern. Je besser der Agent die Leistungsvorgaben erfüllt, desto geringer kann der Einfluss des Experimentierelements ausgelegt werden. Ist der erlernte Stand gut genug, so kann dieses Element auch deaktiviert werden, wenn der Anwender das für sinnvoll hält. Dieses Verfahren wird ϵ**-Greedy**-Verfahren genannt.

Es gibt auch andere Verfahren, mit denen die verschiedenen möglichen Wege erkundet werden können. In [6] S. 376f. werden der „Naïve Algorithm“ und die „Upper Bounding Methods“ vorgestellt.

Der **Naïve Algorithmus** probiert für jeden Zustand eine bestimmte Anzahl von Aktionen durch und wählt dann die beste Aktion aus. Diese Aktion wird von da an immer ausgeführt, wenn derselbe Zustand auftritt. Bei diesem Algorithmus gibt es mindestens drei Probleme:

- Wie viele und welche Aktionen sollen in jedem Zustand ausgeführt werden?
- Das Ausprobieren von vielen Aktionen dauert sehr lange
- Wird einmal eine falsche Aktion gewählt, so wird das nicht wieder korrigiert

Die **Obere Schranken Methoden** basieren auf statistischen Berechnungen und präferieren seltener aufgerufene Aktionen.

Weiterführende Informationen zu den beiden vorgestellten Verfahren können unter [6] S. 376f. nachgelesen werden.

Es gibt noch ein weiteres Verfahren, über welches die auszuführenden Aktionen ausgewählt werden können: Werden bei dem Tetris-Spiel die Blöcke platziert, kann es vorkommen, dass verschiedene Endpositionen des Tetromino in einem bestimmten Zustand ähnlich „gut“ sind. In diesem Fall könnte es von Vorteil sein, diese Aktionen auch gleich oft in diesem Zustand auszuführen. Diesen Ansatz benutzt der sogenannte **Softmax**. Es handelt sich hierbei um eine auf eins normierte Funktion, die die Wahrscheinlichkeit P_τ für jede Aktion a_i berechnet, mit welcher diese ausgeführt werden soll. Die Aktionswahl basiert demnach, im Gegensatz zu dem ϵ-Greedy Verfahren, nicht auf reinem Zufall oder der Wahl der vermeintlich besten Aktion, sondern Ausführungs-Wahrscheinlichkeiten, die aus den Erfahrungen über die Belohnungen von Aktionen in bestimmten Zuständen berechnet werden. Die Softmax-Funktion wird über einen Parameter τ parametrisiert, der beeinflusst, in wie weit nur die hohe Belohnung wichtig ist (τ sehr klein) oder eine hohe Gleichverteilung der Wahrscheinlichkeiten erreicht werden soll (τ sehr groß). Die Softmax-Funktion ist in Gleichung 1 dargestellt. Das Gleichung 2 gilt ist direkt ersichtlich, da die Summe über die Wahrscheinlichkeit von jeder Aktion immer 1≙100% ergeben muss. Weiterführende Informationen können unter [7] S. 351ff. nachgelesen werden.

$$P_\tau(a_i) = \frac{exp(x(a_i)/\tau)}{\sum_{i=1}^{n} exp(x(i)/\tau)} \tag{1}$$

$$1 = \sum_{i=1}^{n} P_\tau(a_i) \tag{2}$$

Im Rahmen dieses Projekte wurde das Softmax-Verfahren genutzt, um die durchzuführenden Aktionen zu bestimmen.

2.2. Markow-Entscheidungsprozess

Der Markow-Entscheidungsprozess bildet die Grundlage für das Lernen des im Rahmen dieses Projektes implementierten Agenten. Nach [7] S. 334 ff. handelt es sich hierbei um einen stochastischen Prozess, der nach dem russischen Mathematiker Andrei Andrejewitsch Markow (1856-1922) benannt ist. Das Ziel dieses Prozesses ist es, Wahrscheinlichkeiten für das Eintreten bestimmter Zustände, ausgehend von dem aktuellen Zustand, berechnen zu können.

Im Folgenden wird beschrieben, welche Komponenten für Markow-Prozesse benötigt werden:

Die Zustandsmenge S beinhaltet alle möglichen Zustände, die ein System einnehmen kann.

Die Aktionsmenge A beinhaltet alle Aktionen in dem System, die in einem bestimmten Zustand ausgeführt werden können.

Die Zustandsübergangsfunktion δ beschreibt den Übergang eines Zustandes $s_n \in S$ in einen Zustand $s_{n+1} \in S$ mit einer bestimmten Aktion $a_n \in A$, sodass gilt:

$$s_{n+1} = \delta(s_n, a_n) \tag{3}$$

Die Belohnungsfunktion r beschreibt die Reaktion der Umgebung auf die Zustandsänderung. Wie „gut“ ist die Aktion, im Sinne des zu erreichenden Zieles, in der entsprechenden Situation gewesen. Über die Belohnungsfunktion kann dementsprechend eine Belohnung r_n für die Aktion a_n in dem Zustand s_n ermittelt werden.

$$r_n = r(s_n, a_n) \tag{4}$$

Nach Gleichung (3) ist der Zustandsübergang bei gegebenem Zustand und gegebener Aktion immer eindeutig. Im Allgemeinen kann das nicht immer gewährleistet werden. Es ist sogar so, dass die Zustandsübergänge häufig nicht deterministisch sind. Auch in dem implementierten Tetris-Spiel sind diese nicht eindeutig. Dennoch wird im Rahmen dieses Projektes auf die Theorie der deterministischen Zustandsübergänge im Sinne der Gleichung (3) aufgebaut, in der Hoffnung, dass der resultierende Fehler das Ergebnis unwesentlich beeinflusst.

Es handelt sich hierbei um den sogenannten **deterministischen Markov-Entscheidungsprozess**.

Damit der Agent das Ziel möglichst „gut“ erreicht, muss er die **optimale Strategie** π^* kennen. Aufgrund der **Strategie** π des Agenten wählt dieser die Aktion für einen bestimmten Zustand, es gilt demnach:

$$\pi : S \to A; \pi(s_n) = a_n \tag{5}$$

Die optimale Strategie π^* bedeutet, dass die Summe aller Belohnungen bis zum Erreichen des Ziels maximal ist.

Angenommen, ein Agent kennt die optimale Strategie und erreicht nach b Schritten sein Ziel. Dann ist $\sum_{i=0}^{b} r_i$ die maximal zu erreichende Summe aller erhaltenen Belohnungen. Eine solche Aufsummierung ist in der Praxis allerdings nicht sinnvoll, da ein Agent häufig kein Ziel hat, bei dem er „fertig“ ist. Außerdem kann oftmals keine Grenze für die maximal auf zu summierenden Schritte angegeben werden, sodass man folgendes Problem erhält: $\sum_{i=0}^{\infty} r_i$

Diese Summe könnte gegen $\pm\infty$ streben, sodass keine Zahl aus den reellen Zahlen angegeben werden kann. Daher wird ein sogenannter **Diskontierungsfaktor** $0 \leq \gamma < 1$ eingebracht, sodass die kumulierte Belohnung $\sum_{i=0}^{\infty} \gamma^i r_i$ beträgt. Daraus ergibt sich die sogenannte **Value-Funktion** $V^\pi(s_n)$ mit $V^\pi(s_n) = \sum_{i=0}^{\infty} \gamma^i r_i$. Es kann gezeigt werden, dass die Value-Funktion für endliche Belohnungen gegen einen Grenzwert strebt. Der Diskontierungsfaktor gibt an, wie „stark“ spätere Belohnungen bewertet werden. Bei der optimale Strategie ist auch die Value-Funktion für alle Zustände maximal ($V^*(s)$).

Das Problem in der Praxis ist, dass weder V^*, π^*, δ noch r bekannt ist.

2.3. Q-Learning

Im Folgenden wird der Q-Learning-Ansatz erläutert. Dazu wird nur auf die für diese Arbeit relevanten Aspekte eingegangen. Eine ausführlichere Erklärung kann in [6] S. 384ff. oder [7] S.342ff. nachgelesen werden.

Der Q-Learning-Ansatz versucht aus einer gegebenen Anzahl von Aktionen die Beste für den aktuellen Zustand zu ermitteln. Dazu wird die sogenannte **Action-Value-Funktion** $Q(s, a)$ verwendet, die den zu erwartenden Nutzen einer bestimmten Aktion in einem bestimmten Zustand liefert. Es gilt:

$$Q(s, a) = r(s, a) + \gamma \cdot V^*(\delta(s, a)) \tag{6}$$

Weil δ nicht bekannt ist und es demnach unbekannt ist, wie die Umgebung auf eine Aktion reagieren wird, muss die Action-Value-Funktion iterativ gelernt werden. Dazu gilt nach [7] S. 343 der folgende Zusammenhang:

$$\hat{Q}(s, a) = r + \gamma \cdot \max\hat{Q}(s', a') \tag{7}$$

Dabei ist $\hat{Q}$ die Approximation von Q, r, die die im aktuellen Schritt beobachtete Belohnung (nicht die Belohnungsfunktion) kennt. Wie diese Approximation ermittelt werden kann wird in dem nachfolgenden Kapitel erläutert.

2.4. Growing Batch Reinforcement Learning

Eine Variante des Bestärkten Lernens (Reinforcement Learning) ist das Growing Batch Reinforcement Learning. Der Ansatz dabei ist eine iterative Funktionsapproximation der Q-Funktion aufgrund von gesammelten Daten bzw. Informationen. Der Agent interagiert mit der Umwelt und sammelt bei jeder seiner Aktionen das Tupel (s, a, r, s'), also den Ausgangszustand, die ausgeführte Aktion, die erhaltene Belohnung und den eingenommenen Zustand. Bei der Approximation der Q-Funktion werden die bislang gesammelten Daten und somit alle Informationen, die der Agent über die Umwelt hat, genutzt, um die Q-Funktion genauer anzunähern.

Als Regressionstechnik wurde für dieses Projekt ein neuronales Netzwerk gewählt. Die dafür nötigen Trainingsdaten werden über das sogenannte **Experience Replay**-Verfahren gewonnen. Bei diesem Verfahren werden, aufbauend auf Gleichung (7), für die gespeicherten Tupel die folgenden Berechnungen ausgeführt:

$$q_{s,a}^{i+1} = r + \gamma \cdot \max_{a'}\hat{Q}_i(s', a') \tag{8}$$

$$y_n = (1 - \alpha) \cdot \hat{Q}_i(s, a) + \alpha \cdot q_{s,a}^{i+1} \tag{9}$$

$$x_n = (s, a) \tag{10}$$

Wieso die vorgestellten Gleichungen gelten kann in bei [7] auf S. 371 oder [10] auf S. 74 nachgelesen werden.

Dem trainierten neuronalen Netzwerk wird x_n, also der aktuelle Zustand des Spiels und eine Aktion

übergeben. Der Output ist die nach der aktuellen Approximation der Q-Funktion erwartete Belohnung für diese Aktion, $\hat{Q}(s,a)$. Um mittels einer Policy $\pi(s_n) = a_n$ eine Aktion auszuwählen, werden die approximierten ermittelten Q-Values $\hat{Q}(s,a)$ für alle Aktionen a an das Softmax-Verfahren übergeben. Das bestimmt dann die Wahrscheinlichkeit für jede Aktion in Abhängigkeit des approximierten Q-Wertes und wählt die auszuführende Aktion aus. Die tatsächliche Belohnung des Systems für diese Aktion wird von dem Agenten für den nächsten Trainingszyklus gespeichert, um die Q-Funktion noch besser zu approximieren.

2.5. Weitere Betrachtungen

2.5.1. TD-Learning

Temporal Difference Learning oder TD-Learning ist eine Methode des modellfreien bestärkenden Lernens wie in [10] S. 72ff. beschrieben. TD-Learning verlässt sich dabei nicht auf abgeschlossene Episoden um eine Strategie zu ändern und Informationen über zukünftige Rewards rückwärts über die Zustände zu propagieren.

Die einfachste TD-Methode ist dabei TD(0), welche mit der State-Value Funktion V^π in Abhängigkeit der Policy π arbeitet. Dabei wird nach jedem Übergang von Status s zur s' mit Reward r die neue Valuefunction V_{neu} bestimmt:

$$V_{neu}(s) = V_{alt}(s) + \alpha \cdot [r + \gamma V(s') - V_{alt}(s)] \tag{11}$$

$V(s')$ ist hierbei natürlich ein unbekannter über Approximation zu bestimmender Wert. Vergleicht man die Funktion des TD(0) mit der in dieser Arbeit verwendeten Zuweisung der Aktion-Value-Function für das Q-Learning auf partially observable Markov-Decision processes, wird deutlich dass es sich beim Q-Learning um einen Ansatz zur Umsetzung des TD-Learnings handelt:

$$Q_{neu}(s,a) = Q_{alt}(s,a) + \alpha \cdot [r + \gamma \max_{a'} Q(s',a') - Q_{alt}(s,a)] \tag{12}$$

Die State-Value-Function des TD-Algorithmus bewertet im Gegensatz zum Q-Learning ausschließlich in Abhängigkeit vom Zustand s und unabhängig von den möglichen Aktionen a. Entsprechend ist die Bestimmung des besten im nächsten Zustand möglichen Wertes über das Bilden des Maximums aller Aktionen nicht erforderlich. Der vorgestellte TD(0)-Algorithmus wird auch als one-step Algorithmus bezeichnet. Es werden auch n-step TD-Algorithmen verwendet welche Zusammenhänge zwischen einem Zustand und dem n-ten darauf folgenden Zustand herstellen (siehe [10] S. 72ff.).

2.5.2. Deep-Q-Network

Unter [11] S. 262ff. werden weitere Ansätze zum Reinforcement Learning diskutiert. Deep Minds Deep-Q-Network ist aus der Motivation entstanden Q-Funktionen auch auf extrem großen Zustandsräumen Approximieren zu können. DQN basiert auf dem bereits vorgestellten Q-Learning Ansatz, erweitert diesen aber durch weitere Techniken.

Zunächst wird die Stabilität des Lernvorgangs als kritisch identifiziert. Die vom tiefen neuronalen Netzwerk zu approximierenden Zielwerte von Q ändern sich beim Lernfortschritt. Außerdem hängen die zu lernenden Q-Werte bereits von vorhergesagten Q-Werten ab. Um die Stabilität zu vergrößern wird ein zweites neuronales Netz, das Target-Q-Network, eingeführt. Das erste Netzwerk wird nun verwendet um $\hat{Q}(s, a)$ zu bestimmen und das Target-Q-Network um $\hat{Q}(s', a')$ vorherzusagen. Das Target-Q-Network ist eine Kopie des ersten Netzes, das verzögerte Parameterupdates erfährt. Es wird immer nach mehreren Batches mit dem ersten Netzwerk gleichgesetzt.

Auch die Verwendung von Deep Recurrent Q-Networks DRQN wird unter [11] S. 273ff. diskutiert. Diese werden für Probleme verwendet, welche sich nicht mehr direkt über den MDP mit $s_{n+1} = \delta(s_n, a_n)$ beschreiben lassen, sondern auch von historischen Zuständen und Aktionen abhängen. Mit dem DRQN gelingt es dem Modell selbst zu lernen wie viele Zustände rückwirkend relevant sind.

3. Existierende Implementierungen

Die Implementierung einer KI zur Lösung von Tetris ist mittels diverser Ansätze erprobt worden. Lösungsansätze in den frühen 2000er Jahren verwendeten dabei eine statische, vom Programmierer definierte Strategie (Policy). Auch lernende Ansätze mit entsprechend dynamischer Policy wurden zu dieser Zeit implementiert. Diese basierten teilweise auf einer starken Reduzierung des Tetris-Spiels durch Beschränkung der Spielfeldgröße, Tetrominoarten und -formen. In den letzten Jahren scheint vor allem die Implementierung genetischer Algorithmen, wie unter [12] und [13] zu finden, populär zu sein. Im Folgenden sollen verschiedene Ansätze diskutiert werden und erfolgreiche Ansätze identifiziert werden, um sie ggf. für die eigene Implementierung nutzen zu können.

3.1. Statische Strategie

Eine gute per Hand programmierten Strategie wurden von Colin P. Fahey und Pierre Dellacherie implementiert (siehe [14]). Dabei ist zwischen Algorithmen, welchen nicht nur der aktuell zu platzierende, sondern auch der darauf folgende Tetromino bekannt ist, zu unterscheiden. Bei einem bekannten Stein gelang es dem Algorithmus im Durchschnitt bis zu 2 Millionen Reihen pro Spiel zu löschen. Der Rekord des Algorithmus bei zwei bekannten Steinen liegt bei 7,2 Millionen gelöschten Reihen.

Die Implementierung der statischen Strategie basiert dabei auf den Beobachtungen und Lösungsansätzen des Programmierers. Da sie keinen lernenden Anteil hat, können keine weiteren Strategien umgesetzt werden, welche über das Spielverständnis des Programmierers hinausgehen. Die lernende Eigenschaft eines Algorithmus ist als fundamentaler Anteil der KI anzusehen, weshalb für diese Arbeit die Implementierung einer „handkodierten“ Policy nicht vorgesehen wird.

3.2. Reinforcement Learning

Eine Implementierung basierend auf Reinforcement Learning erfolgte 1998 von Melax (siehe [4]). Diese Implementierung basiert auf der starken Reduzierung des Problems auf weniger und einfachere Tetrominos, sowie die Limitierung der Spielfeldgröße, wie in Abbildung 4 und 5 zu sehen ist. Der gesamte Zustandsraum wurde hier auf insgesamt 4096 Zustände reduziert. Der primäre Fokus bei dem von [4] vorgestelltem Agenten liegt dabei, die Höhe zu minimieren, die mit einer festen Anzahl von Steinen erreicht wird. Dem Agenten gelingt eine minimale Höhe von 289 nach 10.000 platzierten Steinen. Der verwendete Q-Learning Ansatz gleicht hierbei der in dieser Arbeit implementierten, in Abschnitt 7 beschriebenen „Utility Function” mit den Konstanten $0 \leq \alpha, \gamma \leq 1$. Durch den Diskontierungsfaktor γ wird eine Konvergenz für die Utility Function garantiert. Durch das über α implementierte, inkrementelle Lernen wird der nicht-deterministische Aspekt des Tetris-Spiels berücksichtigt. Weiterhin werden hier die Aspekte eines „migrating alpha”, also ein α welches mit wachsender Laufzeit kleiner wird, als positiv identifiziert. Durch die drastische Reduktion der Komplexität war eine Approximation der Q-Funktion nicht nötig, und es konnte eine Implementierung über Tabellen verwendet werden.

Abbildung 4: vereinfachte Tetrominos aus [4]

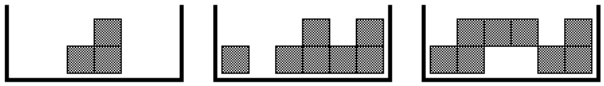

Abbildung 5: Spielfeld aus [4]

Weitere Verbesserungen am Melax Ansatz durch Bdolah und Livnat konnten die Lernergebnisse drastisch verbessern. Die konkreten Verbesserungen waren die Beschreibungen der Zustände durch Konturen und die Identifizierung von symmetrischen Zuständen und resultierende Reduktionen des Zustandsraums (siehe [5]). Beide Ansätze verwenden keine Regression der Q-Funktion sondern implementieren das Lernen über Tabellen wie es zum Beispiel unter [7] S. 331ff. und [4] zu finden ist.

Ein mittels BrainSimulator von GoodAI, C# und einem „Tetris World" genanntem Tetris-Framework programmierter Agent, welcher Reinforcement Learning mittels Approximation einer Value-Function über neuronale Netze verwendet wurde von [3] implementiert und arbeitet mit dem originalen Tetris. Diese Arbeit versucht unter anderem die herkömmlichen Spieler-Inputs als Aktionen mit der Angabe der finalen Rotation und Translation des Tetrominos als Aktion zu vergleichen. Weiterhin sollen die Verwendung des gesamten Spielfelds und einer Höhenangabe pro Spalte als Zustand verglichen werden. Die vorgestellte Lösung verwendet außerdem das im Abschnitt 2.5.1 beschriebene Temporal Difference Learning und Experience Replay. Keinem der von [3] implementierten Agenten ist es gelungen in einem Spiel eine einzige Reihe zu löschen.

3.3. Lungaard und McKee

Weitere Ansätze, welche unter anderem Reinforcement Learning verwenden, werden in [15] diskutiert. Es werden diverse Agenten mit als „high- und low-level" definierten Aktionen, sowie verschiedenen Darstellungen der Zustandsräume verglichen.

Aus den unverarbeiteten Informationen zu Spielfeld und Tetrominoart, dem „low-level Statespace", wird auf Grund der Größe dieses Zustandsraums ein weiterer „high-level Statespace" konstruiert. Dieser enthält folgende 7 Features und kann so auf eine Größe von 40.000 Zuständen reduziert werden.:

- aktueller Tetromino $[1, 7]$
- nächster Tetromino $[1, 7]$
- Durchschnittshöhe $h, 0 \leq h \leq 20$
- Bool: Ist das Spielfeld in einer definierten Grenze eben?
- Bool: Ist im Spielfeld ein Platz für das I-Stück?
- Bool: Sind mehrere Plätze für das I-Stück vorhanden?

- Anzahl der verdeckten Löcher in Gruppen keine, wenige, viele ... [1, 5]

Weiterhin werden hierzu passende high-level Aktionen definiert. Diese Aktionen definieren jeweils einen Greedysearch, welcher über alle möglichen Translationen und Rotationen iteriert und die optimale Aktionen für die entsprechende Aktion und den Zustand ermittelt. Die beschriebenen Aktionen sind:

- minimiere die Anzahl der Löcher
- minimiere die Höhe
- maximiere die Anzahl gelöschter Reihen
- erstelle einen Platz für ein I
- erstelle einen Platz für ein L
- leere das Spielfeld

Je nach Spielfeld sind verschiedene Aktionen unterschiedlich effektiv. Kann zum Beispiel keine Reihe gelöscht werden und die Aktion „maximiere die Anzahl gelöschter Reihen" wird gewählt, so wird vom Greedysearch ein Zufallsergebnis geliefert. Für jeweils low- und high-level Ansätze wurden diverse Agenten implementiert.

Zunächst wird die Performance heuristischer Agenten, welche ausschließlich eine definierte high-level Aktion ausführen evaluiert. Diese arbeiten wie oben beschrieben ausschließlich auf einer Suche über alle möglichen Tetrominoplatzierungen. Dabei erzielen die heuristischen Agenten, welche die Aktion „minimiere die Anzahl der Löcher" oder „minimiere die Höhe" ausführen, durchschnittlich 9,5 gelöschte Reihen pro Spiel. Der heuristische Agent mit der Aktion „leere das Spielfeld", welche nach gewichteten Kriterien, wie minimale Anzahl der Löcher, minimale Höhe, möglichst glatte Spielfeldkontur und maximale Anzahl gelöschter Reihen, bewertet, erreicht etwa 467 gelöschte Reihen pro Spiel. Diese Strategie ähnelt stark der von Colin P. Fahey und Pierre Dellacherie unter [14] implementierten statischen Policies.

Weiterhin wurden Agenten, welche mit Q-Learning arbeiten, evaluiert. Die Agenten nutzen dabei keine approximierte Q-Funktion und arbeiten auf den oben beschriebenen high-level Aktionen und Zustandsräumen, sowie mit einer ϵ-greedy Exploration. Es wurden hierbei weiterhin Hyperparameter wie α und ϵ variiert. Dem Q-Learning Agenten gelingt es nicht sich gegenüber dem ausschließlich mit der Aktion „leere das Spielfeld" spielenden, heuristischen Agenten zu verbessern. Stattdessen ist der lernende Agent sogar schlechter, obwohl ihm letztere Aktion ebenfalls zur Verfügung steht. Dieses Verhalten wird auf die hohe Inkonsistenz bei der Performance der ausgeführten high-level Aktionen und den zu erwartenden Rewards zurückgeführt.

Weitere Ansätze aus [15] arbeiten dabei mit einer approximierten State-Value-Function. Hierbei wird das vorgestellte one-step Temporal Differential Learning verwendet, da sich dieses anbietet wenn der nächste Tetromino ebenfalls bekannt ist. Die Explorationsstrategie ist ϵ-greedy. Es werden zwei verschiedene Agenten basierend auf diesem Ansatz implementiert, welche entweder auf high- oder low-level Aktionen und Zuständen lernen.

Der low-level Agent konnte dabei keinen Erfolg gegenüber zufällig platzierten Tetrominos erzielen. Dies ist darauf zurückzuführen, dass das implementierte low-level Netzwerk 200 Input-Neuronen für

das Spielfeld und 2 für die Tetrominoart hat. Es werden keine Convolutional-Layer und wenige Hidden-Layers verwendet. Ein auf einem so großen Zustandsraum lernendes Netzwerk ist entsprechend schwierig in einem realistischem Zeitraum zu trainieren.

Das high-level Netzwerk, welchem, wie auch dem über Tabellen lernenden Q-Learning Agenten, die heuristisch gewählten Aktionen zur Verfügung stehen, verbessert sich ebenfalls nicht gegenüber dem Agenten, welcher nur mit der Aktion „leere das Spielfeld" spielt. Es werden etwa 260 Reihen pro Spiel gelöscht. Theoretisch wäre es dem Agenten möglich zu lernen, ausschließlich letztere Aktion auszuführen. Dies geschieht allerdings nicht, weil die Belohnungsfunktion an den Tetris-Score gekoppelt wurde. Hier gibt es für das gleichzeitige Löschen mehrerer Reihen größere Belohnungen. In Folge dessen lernt der Agent eine risikoreichere Strategie, um pro Aktion mehr Belohung zu erzielen, verliert aber dafür auch schneller das Spiel.

3.4. Q-Learning mit Approximation über Convolutional Neural Networks

Weitere Ansätze, wie z. B. [16], verwenden Convolutional Neural Networks zur Approximation der in Gleichung 7 beschriebenen Q-Funktion und lernen auf dem gesamten Spielfeld als Input. Es werden verschiedene tiefe neuronale Netze mit mehreren Convolutional und Pooling Layers getestet. Außerdem werden bis zu 8 Hidden-Layers getestet. Hier werden weiterhin die Explorationsverfahren ϵ-Greedy und Softmax verglichen. Dabei wird die ϵ-Greedy Policy als inferior identifiziert. Ferner wird das Experience Replay Verfahren zum Training der Netze verwendet. Mittels Prioritized Sweeping werden bestimmten Datensätzen Prioritäten zugeordnet und entsprechend für die Learning Batches in Betracht gezogen. Die Priorität p wird hierbei wie folgt bestimmt:

$$p = \left| Q(s,a) - r - \max \hat{Q}_i(s',a') \right| \tag{13}$$

Hiermit soll gewährleistet werden, dass das Netzwerk auf den Datensätzen mit dem größten Fehler lernt, weil es dort das größte Verbesserungspotenzial hat. Es wird erneut zwischen Aktionen, welche die Endposition des Tetrominos im Spielfeld vorgeben und solchen die Spielerinputs ähnlich sind, unterschieden. Bei Letzterem wird die Totzeit zwischen Input und Belohnung als besonders kritisch identifiziert. Es wird eine Belohnungsfunktion, welche in Abhängigkeit von Höhe, gelöschten Reihen, Löchern und Glätte der Kontur bewertet, verwendet. Die Lernrate Alpha wurde für den Ansatz von [16] zwischen $2 \cdot 10^{-6}$ und $3 \cdot 10^{-7}$ variiert. Der Diskontierungsfaktor wurde zwischen 0,75 und 0,9 variiert. Das beste erzielte Ergebnis waren 18 gelöschte Reihen pro Spiel bei einem nicht mit Prioritzed Sweeping Samples und Endpositions-Aktionen verwendenden Agenten. Dass das Prioritized Sweeping nicht erfolgreich war, wird der fehlenden Trainingszeit des entsprechenden Agenten zugesprochen und abschließend als zielführend identifiziert.

3.5. Genetische Algorithmen

Schon 1996 wurde der erste genetische Algorithmus für Tetris von Roger Espel Llima für das „xtris"-Tetris implementiert (siehe [14]). Diesem gelingt das Löschen von durchschnittlich 42.000 Reihen pro Spiel (siehe [5]). Eine detaillierte Dokumentation der Implementierung ist nicht mehr verfügbar. Neuere Implementierungen genetischer Algorithmen erreichen deutlich bessere Ergebnisse. Der von [13] implementierte Algorithmus ist noch nicht bis zum Spielabbruch getestet worden, erreicht aber nach mehreren Wochen über 2 Millionen gelöschte Reihen. Es ist eine webbasierte Demonstration der spie-

lenden KI unter [13] verfügbar. Die Bezeichnung als „genetischer Algorithmus" ist bei der Betrachtung der Implementierung nur unter Einschränkungen zu gebrauchen. Tatsächlich wird auch hier, wie bei den unter [15] verwendeten high-level Aktionen eine Suche über alle möglichen Aktionen durchgeführt um einen heuristisch definierten Reward zu maximieren. Der Reward r ist mit den Gewichten w_i Abhängig von Löchern, der Höhe, gelöschten Reihen und der Glätte der Kontur:

$$r = w_1 * \text{holes} + w_2 * \text{lines} + w_3 * \text{height} + w_4 * \text{bumpiness} \tag{14}$$

Über den Genetischen Algorithmus wird nun lediglich das Tupel w der Gewichte w_i optimiert. Es ist davon auszugehen das ein ähnlicher Ansatz auch von Roger Espel Llima verwendet wurde.

4. Konzept

Zur Entwicklung eines Konzeptes für die Implementierung eines Agenten, welcher über Neuro Fitted Q Iteration lernt, werden einzelne erfolgreiche Ansätze aus der Literatur aufgegriffen. Es wurde deutlich, dass Agenten welche über eine Heuristik und ein Suchverfahren in jedem Zustand lernen für das Problem deutlich besser geeignet sind, als Agenten, welche die beste Aktion direkt über eine gelernte Aktion-Value-Function oder ein ähnliches Verfahren bestimmen. Ersterer Ansatz wird im Folgenden nicht verwendet, da dieser dem Agenten einen kleineren Lernfortschritt ermöglicht und näher an einer statisch implementierten Policy ist. Die verwendete Implementierung über Q-Learning auf der Spielfeldinformationen und Tetromino-Endpositionen erlaubt dem Agenten theoretisch Strategien zu erlernen, welche über das Lernen von Gewichten für Suchparameter nicht möglich ist. Letzteres ist zwar unwahrscheinlich, da auf diese Weise lernende Agenten deutlich schwächer sind, allerdings wäre solch ein Ergebnis besonders interessant zu beobachten.

4.1. Güte der Strategie

Zum Bestimmen eines Lernerfolgs, ist es notwendig, die vom Agenten zu lernende Aufgabe, sowie eine Metrik für die Güte der Strategie und die optimale Strategie zu definieren. Bei der Bestimmung der algorithmischen Komplexität mit vollständig bekannter Tetrominosequenz sind die folgende Aufgaben untersucht worden:

1. Maximierung der Anzahl der gelöschten Reihen
2. Maximierung der Anzahl der platzierten Tetrominos vor dem Verlieren des Spiels
3. Maximierung der Anzahl des gleichzeitigen Löschens von vier Reihen
4. Minimierung der Höhe des höchsten im Spielverlauf platzierten Tetrominos

(siehe [1, 2])

Bei der Bewertung der Strategie wird in Abschnitt 3 oft die durchschnittliche Anzahl der pro Spiel gelöschten Reihen angegeben. Diese Angabe ist sinnvoll, wenn der Agent auf einem Spiel lernt, welches beim Verlieren des Spiels das gesamte Spielfeld löscht und neu startet. Alternativ wird die Güte der Strategie durch die nach einer konstanten Anzahl platzierter Tetrominos nicht gelöschter Reihen beurteilt (vgl. [4]). Letzteres Bewertungskriterium ist sinnvoll, wenn das Spielfeld beim Verlieren nicht zurückgesetzt wird.

Es wird ein Ansatz ohne Rücksetzen des Spielfeldes implementiert, um möglichst konstante Zustandsübergänge zu ermöglichen. Beim Verlieren werden stattdessen eine konstante Anzahl der untersten Reihen gelöscht und das Spielfeld entsprechend verschoben. Diese Änderung im Zustand ist vom Agenten nicht zu erkennen, wie in Abschnitt 4.3 deutlich wird. Die Güte der aktuellen Strategie des Agenten wird im Folgenden an der Anzahl der pro platziertem Tetromino gelöschten Reihen beschrieben. Je nach Spielfeldbreite ergibt sich eine maximal mögliche Anzahl gelöschter Reihen pro Tetromino, worauf im Folgenden nur noch als Güte Bezug genommen wird. Bei einer Spielfeldbreite von b Feldern und der durchschnittlichen Anzahl der Blöcke pro Tetromino n_t ergibt sich dieses Maximum mit:

$$G_{max} = \frac{n_t}{b} \tag{15}$$

Die von Melax verwendete Beschreibung der Güte des Algorithmus von 289 **nicht** gelöschten Reihen pro 10.000 Steinen bei einer Spielfeldbreite von 6 und 2,4 Blöcken pro Tetromino lässt sich in etwa zu 0,371 gelöschte Reihen pro Tetromino umrechnen. Das Maximum wäre 0,4.

Bei Erreichen der maximalen Güte wäre der Agent dennoch nicht in der Lage das Spiel unendlich lange ohne zu verlieren zu spielen. Nach [17] ist es mathematisch bewiesen, dass es möglich ist eine bestimmte Tetrominosequenz zu generieren welche bei einer Spielfeldbreite von $2(2n + 1)$ auch bei perfektem Spielen zum Verlieren führt. Bei einem unendlich langen Spiel würde diese wahrscheinlich eintreten.

Die Definition der Güte über die pro Spiel gelöschten Reihen ist also bei perfekten oder nahezu perfekten Agenten nicht sinnvoll, da die durchschnittliche Anzahl pro Spiel platzierter Tetrominos, bei ausreichend großer Testmenge, nur die Wahrscheinlichkeit des Eintretens einer solchen, zum Verlust führenden Sequenz, angeben würde. Ab einer bestimmten Anzahl platzierten Tetrominos ohne, dass das Spiel verloren wurde, sind die Strategien also nicht mehr sinnvoll gegeneinander zu bewerten.

4.2. Belohnungsfunktion

Aus der Definition einer guten Strategie ergibt sich ein offensichtliches Konzept für die Belohnungsfunktion. Zunächst erscheint es sinnvoll lediglich die Anzahl der gelöschten Reihen zu belohnen. Die Erweiterung um eine Belohnung für den in Abschnitt 4.1 aufgeführten Optimierungspunkt 3 würde eine Bewertung nach dem Punktestand des original Tetris entsprechen. Wie die Ergebnisse zeigen werden ist die Implementierung einer solchen Belohnung nicht zielführend.

Es wird die Belohnung für eine dem menschlichen Spieler offensichtliche Strategie mitgegeben. Generell gilt: Je dichter gepackt die Tetrominos platziert werden, desto mehr gelöschte Reihen resultieren. Es werden also Belohnungen oder Bestrafungen für folgende Attribute bestimmt:

1. Anzahl der gelöschten Reihen
2. Änderung der maximalen Höhe
3. Änderung in der Anzahl der verdeckten Löcher

Es wird absichtlich darauf verzichtet noch spezifischere Belohnungen für erweiterte Strategien, wie zum Beispiel für das Offenlassen eines Platzes für den O-Stein, zu vergeben. Die Beobachtung, in wie fern der Agent eigenständig ähnliche Strategien entwickelt, wird in der Auswertung untersucht.

4.3. Zustandsraum

Der Zustandsraum beinhaltet alle möglichen Kombinationen des Spielzustandes. Der Zustandsraum des Tetris-Spiels enthält daher mindestens alle Spielfeldkombinationen, sowie den aktuell zu platzierenden Tetromino. Bei einem klassischem Spielfeld der Größe 10×22 ergeben sich damit $2^{10*22} = 2^{220}$

Zustände für das Spielfeld, da jede Spielfeldposition nur zwei Zustände einnehmen kann, nämlich belegt oder frei. Insgesamt gibt es im klassischen Tetris-Spiel sieben verschiedene Tetrominoarten bzw. -formen. Der Zustandsraum, bestehend aus dem aktuellen Tetromino und dem Spielfeld, enthält somit $7 \cdot 2^{220} \approx 1{,}1795^{67}$ Zustände. Wird der auf den aktuellen Tetromino folgende Tetromino noch mit betrachtet, sind es sogar um den Faktor sieben mehr mögliche Zustände. Diese Anzahl ist für reine Bestärkende Lernen ohne Deep-Learning Ansätze zu groß. Daher muss eine Möglichkeit gefunden werden den Zustandsraum wesentlich zu verkleinern.

Dafür wird ein Ansatz gewählt, der nur die Spielfeldkontur beachtet und nicht das gesamte Spielfeld. Diese beschreibt, wie in Abbildung 6 dargestellt, die Höhendifferenz benachbarter Spalten. Dabei ist die Beschreibung der Kontur auf das Intervall von -3 bis 3 begrenzt. Alle Differenzen zwischen zwei Spalten die größer als drei sind, werden auf plus drei bzw. minus drei gesetzt. Für Abbildung 6 ist die Kontur des Spielfeldes: (+1,+3,0,-3,0). Bei einer Spielfeldbreite von b sind es demnach $b-1$ Konturen mit jeweils 7 Zuständen {-3,-2,-1,0,1,2,3}. Bei einer Spielfeldbreite von $b = 10$, dem aktuellen Tetromino und der Spielfeldkontur besteht der Zustandsraum aus $7 \cdot 7^{10-1} = 7 * 7^9 = 282.475.249$ Zuständen. Ist der nachfolgende Tetromino ebenfalls bekannt versiebenfacht sich die Größe des Zustandsraumes. Über den Kontur-Ansatz kann der Zustandsraum somit wesentlich verkleinert werden. Den größten Einfluss auf die Zustandsanzahl hat die Spielfeldbreite, da diese im Exponenten steht. In Tabelle 1 wird dies verdeutlicht.

Tabelle 1: Zustandsraumgröße in Abhängigkeit von der Spielfeldbreite

Spielfeldbreite	**Zustandsraumgröße (nächster Tetromino unbekannt)**	**Zustandsraumgröße (nächster Tetromino bekannt)**
10	282.475.249	1.977.326.743
8	5.764.801	40.353.607
6	117.649	823.543

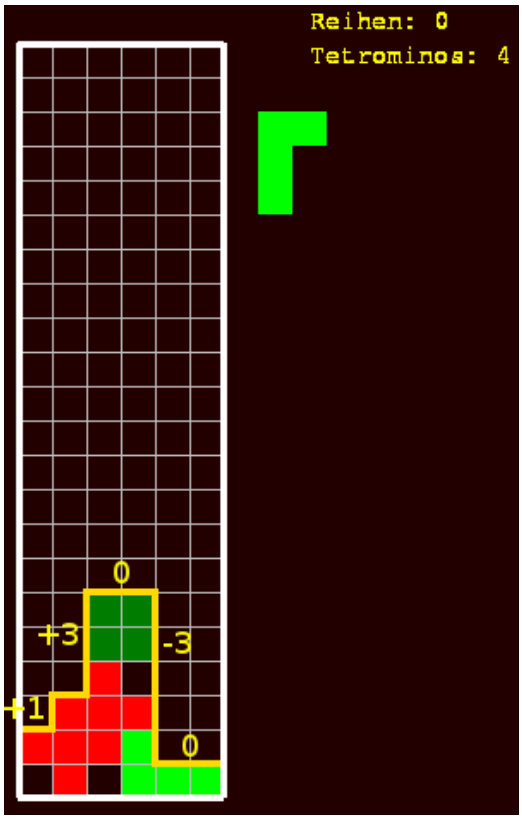

Abbildung 6: Spielfeldbeschreibung als Kontur

4.4. Aktionsraum

Wie bereits zuvor diskutiert bieten sich zwei Optionen für die Implementierung der Aktionen an. Erstere wäre die Umsetzung der Spieler-Inputs als Aktionen. Die verwendete Alternative ist die Implementierung von Aktionen, welche direkt die Endposition des Tetrominos vorgeben. Beide Optionen bieten jeweils Vorteile und Nachteile. Der Nachteil bei der Vorgabe der Endposition ist, dass diese Methode eine größere Anzahl an Aktionen je nach Spielfeldbreite ermöglicht. Ferner wäre es komplex das „Hineindrehen“ in Lücken zu realisieren.

Bei der Simulation der Spieler-Inputs ist vor allem die Zeitverzögerung zwischen Input und Belohnung kritisch. Solange der Tetromino noch nicht abgelegt ist, ist jeder Zustand gleich viel Wert. Um der Bewegung in der „Luft“ unterschiedliche Wert zuordnen zu können, müsste der Agent mit einem sehr großen γ (γ gegen 1) lernen. Auch ist es hier unwahrscheinlich, dass der Agent das „Hineindrehen“ lernt, da Situationen bei denen dieses von Vorteil ist äußerst selten auftreten und auch dann eine spezifische Abfolge von Aktionen erfordern.

Bei dem gewählten Konzept wird darauf verzichtet das „Hineindrehen“ zu berücksichtigen. Es werden jede Endrotation und Endtranslation in der Breite des Spielfelds einer Aktion zugeordnet. Nach dem Erreichen der Position wird der Tetromino senkrecht nach unten verschoben bis er seine Endposition erreicht. Nun soll die Anzahl der notwendigen Aktionen bestimmt werden. Es ist offensichtlich, dass der Aktionsraum eng mit der Spielfeldbreite verbunden ist. Jeder Tetromino muss maximal viermal rotiert werden, bis er seine ursprüngliche Orientierung wieder erlangt. Je nach Symmetrie auch weniger. Es ist leicht zu erkennen, dass Tetrominos mit einer geringen Symmetrie mehr Aktionen benötigen. Um die maximal notwendige Aktionsanzahl zu bestimmen wird daher zunächst der L-Tetromino

betrachtet.

In seiner ursprünglichen Orientierung (wie der Buchstabe L) hat der Tetromino eine Breite $B_1 = 2$ und eine Höhe von drei. Bei einer Spielfeldbreite von b könnte er auf $C_1 = b + 1 - B_1$ Endpositionen abgestellt werden ohne rotiert zu werden. Wird er einmal entgegen dem Uhrzeigersinn rotiert, so hat er eine Breite $B_2 = 3$ und kann auf $C_2 = b+1-B_2$ Endpositionen abgestellt werden. Wird er noch einmal entgegen dem Uhrzeigersinn rotiert, so ergibt sich wieder eine Breite $B_3 = 2 = B_1$. Und bei erneuter Rotation gilt $B_4 = B_2$. Die maximal notwendige Anzahl an Aktionen C kann dementsprechend nach Gleichung 18 berechnet werden.

$$C = C_1 * 2 + C_2 * 1 \tag{16}$$
$$C = (b + 1 - B_1) * 2 + (b + 1 - B_2) * 2 \tag{17}$$
$$C = 2 * (2(b + 1) - (B_1 + B_2)) \tag{18}$$

Ist ein Tetromino π rotationssymmetrisch, wie z. B. der Z-Tetromino, so halbiert sich die benötigte Aktionsanzahl. Bei einer π und $\frac{\pi}{2}$ Rotationsymmetrie ist nur ein Viertel der nach Gleichung 18 berechneten maximalen Aktionsanzahl notwendig (siehe Gleichung 19 20).

$$C_{\pi-rot} = 2(b + 1) - (B_1 + B_2) \tag{19}$$
$$C_{\frac{\pi}{2}-rot} = b + 1 - B_1 \tag{20}$$

In Tabelle 2 sind die resultieren Aktionsanzahlen für die jeweiligen Tetrominos aufgelistet.

Tetromino	**Symmetrien**	**Aktionsanzahl**
L	0	18
J	0	18
T	0	18
S	2	9
Z	2	9
I	2	9
O	4	5

Tabelle 2: Erforderliche Anzahl an Aktionen je Tetrominoart

4.5. Explorationsstrategie

Wie zuvor identifiziert handelt es sich bei dem zu erkundenden Zustandsraum um einen zu großen Raum für einen ohne Approximation der Q-Funktion lernenden Agenten. Auch wenn der Ansatz der Spielfeldkontur gewählt wird. Entsprechend sind auch ϵ-greedy Explorationsstrategien, welche zunächst den gesamten Zustandsraum erkunden wollen ungeeignet. Auch unter [5] wird identifiziert, dass ein ϵ-greedy Verfahren den Agenten effektiv „blind“ machen würden, weshalb dort ebenfalls eine Softmax-Strategie umgesetzt wurde.

4.6. Approximationsfunktion

Die Approximation der Q-Funktion kann über diverse Ansätze implementiert werden. So bieten sich zum Beispiel Decision Trees für besonders sprunghafte Belohnungsfunktionen an, wie sie im konkreten Fall vorliegen (siehe [7] S. 372f.). Weiterhin wird die von Gewichten abhängige Funktion der durchschnittlichen Tetris Performance aus [14] als sprunghaft beschrieben. Ferner besäße diese viele lokale Maxima und Minima, wodurch z. B. ein multidimensionales Gradientenverfahren, welches von einem neuronalen Netz verwendet wird, erschwert werden würde.

Für die Implementierung des Agenten wird dennoch zunächst auf ein neuronales Netz zurückgegriffen, da dieses sicher und schnell, in Kombination mit dem Experience Replay, Ergebnisse verspricht. Außerdem kann das neuronale Netz leichter durch andere Techniken, wie zum Beispiel dem Lernen auf dem gesamten Spielfeld als Input mittels Convolutional Layers oder Deep Learning, erweitert werden.

5. Umsetzung

Im Rahmen dieses Projektes wurde ein simples Tetris-Spiel in der Programmiersprache Python entwickelt. Dazu wurde das Paket *pygame* (siehe [18]) genutzt. Im weiteren Verlauf des Projektes wurde für dieses Tetris-Spiel ein Agent, ebenfalls in Python, geschrieben, der zum Ziel hat, das Spiel zu spielen. Für die Umsetzung des für den Agenten notwendigen neuronalen Netzwerks wurde das Paket *Keras* (siehe [19]) als Erweiterung eingebunden.

Um sich noch einmal darüber klar zu werden, welche Komponenten für das Lernen des Tetris-Spiels mittels Bestärkendem Lernen notwendig sind und wie deren Abhängigkeiten sind, ist in Abbildung 7 eine entsprechende Übersicht dargestellt.

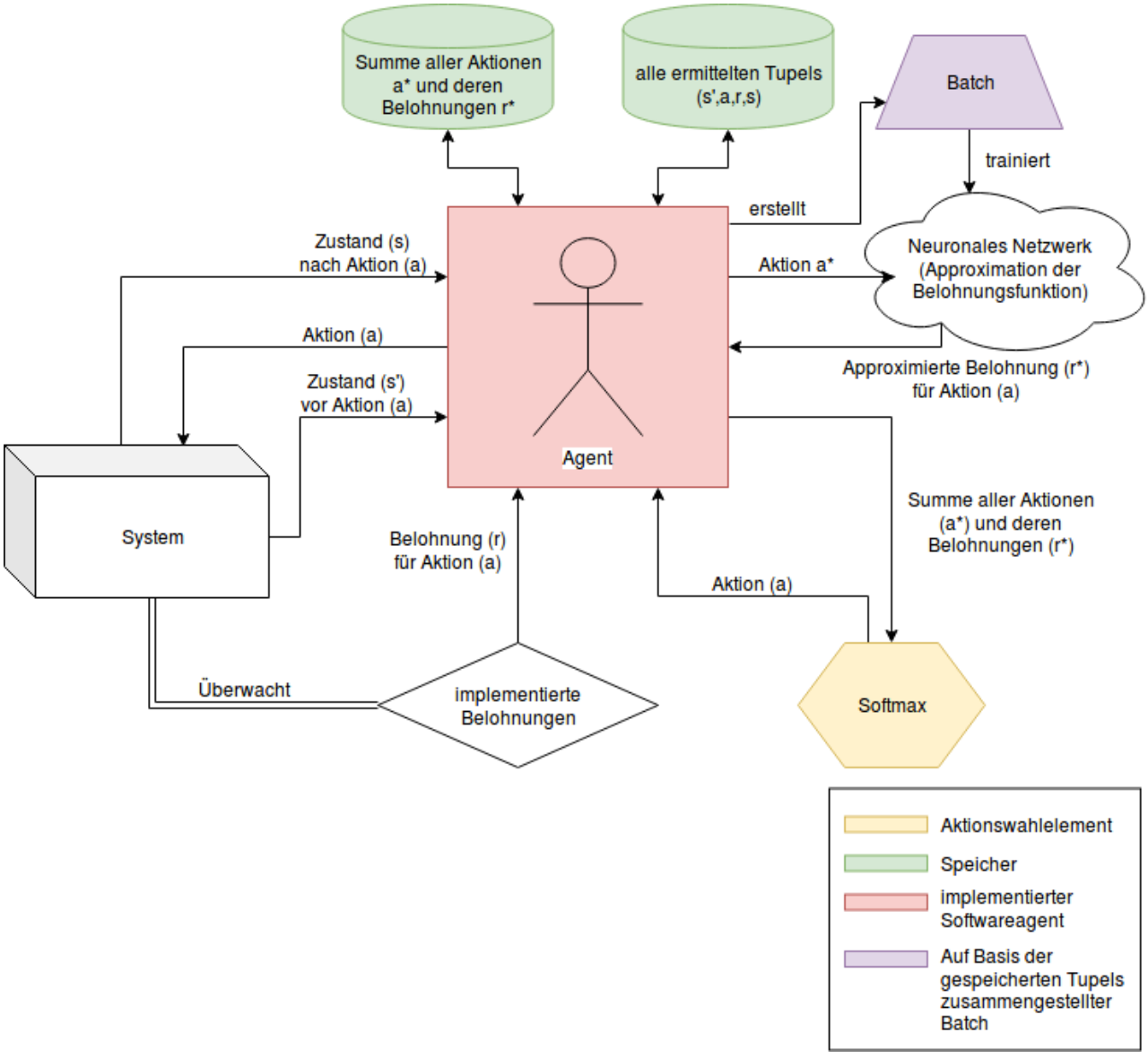

Abbildung 7: Komponentenübersicht des zu implementierenden Bestärkenden Lernens

5.1. Agent

In Kapitel 2.1 wurde die theoretische Funktion und der theoretische Aufbau eines lernenden Agenten beschrieben. Im Folgenden wird die praktische Umsetzung erläutert.

Grundsätzlich arbeitet der Agent mit Zuständen, Aktionen und Belohnungen. Alle diese erfassten Daten werden gespeichert. Allerdings wurde eine maximale Anzahl von 1.000.000 Einträgen definiert, um die Speicher-Ressourcen zu begrenzen.

Zur Initialisierung werden unter anderem die Parameter *Alpha, Gamma, Tau, Spielfeldgröße* und *Batchgröße* benötigt. Die notwendigen Speicher werden reserviert und die erste Phase begonnen.

In der ersten Phase sammelt der Agent Daten, um eine Grundlage für das initiale Training zu haben. Dazu werden eine bestimmte Anzahl zufälliger Aktionen ausgeführt. Diese Phase kann auch übersprungen werden, indem vorher gesammelte Daten geladen werden.

Nachdem der Agent nun einige initiale Datensätze gespeichert hat, beginnt der eigentliche zyklische Ablauf von Daten sammeln, Aktion wählen und lernen.

Die gewählte Regressionsmethode zur Approximation der Q-Funktion ist ein ausschließlich sequentielles neuronales Netz. Dieses Netz besteht aus den folgenden Layers:

1. 256 Dense ReLu
2. 128 Dense ReLu
3. 64 Dense ReLu
4. 1 Dense Linear

Um Overfitting zu verhindern wird in jedem Layer eine L_2-Regularisierung verwendet. Das neuronale Netz wird in jedem Lernzyklus für 3 Epochen und mit einer Gradientenaktualisierung alle 5 Samples trainiert:

```
self.Q.fit(x, qNeu, epochs=3, batch_size=5, verbose=False)
```

Da es sich offensichtlich um ein relativ kompaktes Netz handelt, ist das Training über die GPU nicht zu empfehlen und ist deutlich zeitaufwendiger. Dies wird wie folgt verhindert:

```
os.environ["CUDA_DEVICE_ORDER"] = "PCI_BUS_ID"
os.environ["CUDA_VISIBLE_DEVICES"] = ""
```

5.2. Batches

Um sinnvoll zu Lernen wird ein Minibatch mit einer festen Größe aus den gesammelten Daten erstellt. Dazu werden die besten und schlechtesten Erfahrungen heraus gefiltert. Ist das Batch damit noch nicht gefüllt, wird der Batch mit zufälligen Datensätzen aufgestockt. Sind dagegen schon zu viele extreme Erfahrungen gesammelt worden, werden zufällig einige daraus ausgewählt. So wird in jedem Fall die feste Batchgröße mit einer Teilmenge aller gesammelten Erfahrungen erreicht. Aufgrund dieser Zusammenstellung lernt der Agent. Das neuronale Netz, welches eine Approximation der Q-Funktion

darstellt, wird auf Basis des Batches als Trainingsdaten trainiert, in der Hoffnung eine weniger fehlerhafte Approximation zu erhalten. Bei der Analyse der Ergebnisse werden weitere Möglichkeiten zur Aggregation der Batches abgewogen. Die verschiedenen Ansätze zur Erstellung der Batches sind in Tabelle 3 aufgeführt und werden bei der Darstellung der Ergebnisse mit den hier aufgeführten Bezeichnungen referenziert.

Bez.	ges. Größe	Zufällige	Zusätzliche Auswahl
B1	18.000	15.000	3.000 zufällige mit $r < 0$
B2	18.000	15.000	3.000 zufällige mit $r < 0.9 \min(r)$, $r > 0.9 \max(r)$
B3	18.000	3.000	7.500 Größte r und 7.500 Kleinste r
B4	22.500	15.000	7.500 zufällige $r < 0.7 \min(r)$, $r > 0.7 \max(r)$
B5	20.000	15.000	5.000 zufällige $r < 0.5 \min(r)$, $r > 0.5 \max(Q)$
B6	20.000	15.000	5.000 zufällige aus 500 neusten, $r < 0.5 \min(r)$, $r > 0.5 \max(r)$
B6	20.000	15.000	5.000 zufällige aus Neusten, $r < 0.5 \min(r)$, $r > 0.5 \max(r)$
B7	20.500	15.000	Neuste + 5.000 zufällige aus $r < 0.5 \min(r)$, $r > 0.5 \max(r)$

Tabelle 3: Batches

Das Bilden der Batches wie in B3 über die 7.500 Größten und Kleinsten erfordert die Implementierung einer zusätzlichen Bibliothek. Durch die Verwendung der *nlargest* Funktion von *heapq* können die Indizes der Größten und kleinsten Elemente des Arrays wie in folgt bestimmt werden.

```
index1 = np.array(nlargest(int(self.badMemory/2),range(len(self.rewards[learnSet])),self.
    rewards.take))
index3 = np.array(nlargest(int(self.badMemory/2),range(len(self.rewards[learnSet])),(-1*
    self.rewards).take))
index1 = np.hstack( (index1, index3) )
```

Die Bestimmung der größten und kleinsten Elemente auf diese Weise fordert eine Sortierung des Learn-Sets von der *nlargest* Funktion und birgt deshalb inhärente Performance-Einbuße.

5.3. Belohnungen

Wie bereits zuvor kurz erläutert, führen unterschiedliche Belohnungsfunktionen zu unterschiedlich guten Ergebnissen. Bei der folgenden Diskussion der Ergebnisse wird auf die in Tabelle 4 beschriebenen Belohnungsfunktionen Bezug genommen.

Name	gel. Reihen	Δ H	Δ L	$\Delta H \leq 0$	$\Delta L = 0$
R1	1.000	0	0	0	0
R2	1.000	r_1	-50	50	100
R3	250	r_1	-50	50	100
R4	0	r_1	-50	50	100
R5	250	r_2	-50	50	100
R6	0	r_2	-50	50	100
R7	30	$0.2 \cdot r_2$	-50	50	100
R8	250	$0.2 \cdot r_2$	-50	50	100
R9	150	r_2	-50	50	100
R10	30	r_2	-50	50	100

Tabelle 4: Belohnungsfunktionen

Die in der obigen Tabelle 4 beschriebenen Werte sind Faktoren für die Anzahl gelöschter Reihen, Höhendifferenz ΔH und Differenz in der Anzahl der Löcher ΔL. R1 würde zum Beispiel das Löschen von zwei Reihen mit einer Belohnung von 2.000 bewerten. Die mit r_1 und r_2 Bezeichneten Funktionen sind von der Höhendifferenz nach einer Aktion abhängig. Die Intention bei der Beschreibung der Belohnung über diese Funktionen war die Bestrafung für die Bildung von Stapeln und die Belohnung für eine möglichst Glatte Kontur. Außerdem sollte es bei einer bereits großen Höhe der platzierten Steine eine größere Strafe geben, damit das Spiel nicht vorzeitig beendet wird. Hierbei wird r_1 wie folgt implementiert:

```
max(heightVorher[:])- min(heightVorher[:])
```

Und r_2:

```
5 * max(heightNachher[:])
```

Dabei beschreiben die Variablen *heightVorher* und *heightNachher* die Höhe der einzelnen Spalten des Spielfelds. Die Höhendifferenz ergibt sich entsprechend wie folgt:

```
heightDiff= max(heightNachher[:])-max(heightVorher[:])
```

Bei R2 würde die Belohnung in Bezug auf den Höhenunterschied wie folgt vergeben:

```
if(heightDiff > 0):
    self.rewards[self.memoryCounter] -=5*(max(heightNachher[:])[0])*heightDiff
else:
    self.rewards[self.memoryCounter] += 50
```

Die Relevanz der Belohnungsfunktion für den Lernerfolg des Agenten wird bei der Analyse der Ergebnisse diskutiert.

6. Ergebnisse und Optimierung

Im Folgenden sollen die durch die Variation der Hyperparameter auf der unter Kapitel 5 beschriebenen Ausgangsimplementierung erzielten Ergebnisse diskutiert werden. Außerdem werden auf diesen Ergebnissen begründete Optimierungen erläutert und ausgewertet.

Die Implementierung des Agenten liefert eine Anzahl von Freiheitsgraden, welche verändert werden können um das Ergebnis bzw. die Lern-Performance zu verbessern. Unter anderem sind folgende Parameter relevant:

- **Hyperparameter**
  - Alpha
  - Gamma
  - Tau
- **Belohnungsfunktion**
  - Belohnung für Löschen von Reihen
  - Belohnung für die Anzahl neuer „Löcher“
  - Belohnung für die Veränderung der Höhe
- **Training**
  - Aggregation der Batches
  - Größe der Batches
  - Anzahl und Frequenz der Lernzyklen
  - Epochen für das Fitten des Netzwerks
  - Anzahl der Samples für ein Gradienten Upgrade des Netzwerks
  - Exploration-Verfahren (ϵ-Greedy oder Softmax)
  - Initialisieren des Agenten über vom Menschen eingespielte Daten
- **Approximationsart**
  - Veränderung der Architektur des neuronalen Netzes
  - Approximation über andere Regressionsmethoden, z.B. Decision Trees
  - Trainingsepochen und Frequenz von Gradientenupdates, Loss-Function
  - Regularisierungsmethode

- **Spielparameter**
  - Spielfeldgröße
  - Kenntnis über den nachfolgenden Tetromino
  - Art des Zufallsgenerators

Zum Testen jeder möglichen Kombination bei nur einer Änderung zu jedem oben genannten Parameter wären 2^{20} Testläufe erforderlich. Es wird sich im Folgenden zeigen, dass die zu erwartende Güte des Agenten etwa bei $3 \cdot 10^5$ platzierten Tetrominos absehbar ist. Im Durchschnitt dauert ein solcher Lernzyklus etwa 8 Stunden. Es ist ersichtlich, dass nur ausgewählte Test durchgeführt werden können.

Es wird im Folgenden immer von der im Abschnitt 5 beschriebenen Implementierung ausgegangen. Alle Änderungen werden jeweils kenntlich gemacht. Falls nicht anders angegeben, sind die Hyperparameter $\alpha = 0{,}3$, $\gamma = 0{,}2$ und $\tau = 1$. Die Diskussion der Ergebnisse folgt im Wesentlichen dem chronologischen Verlauf der Tests.

6.1. Belohnungsfunktion

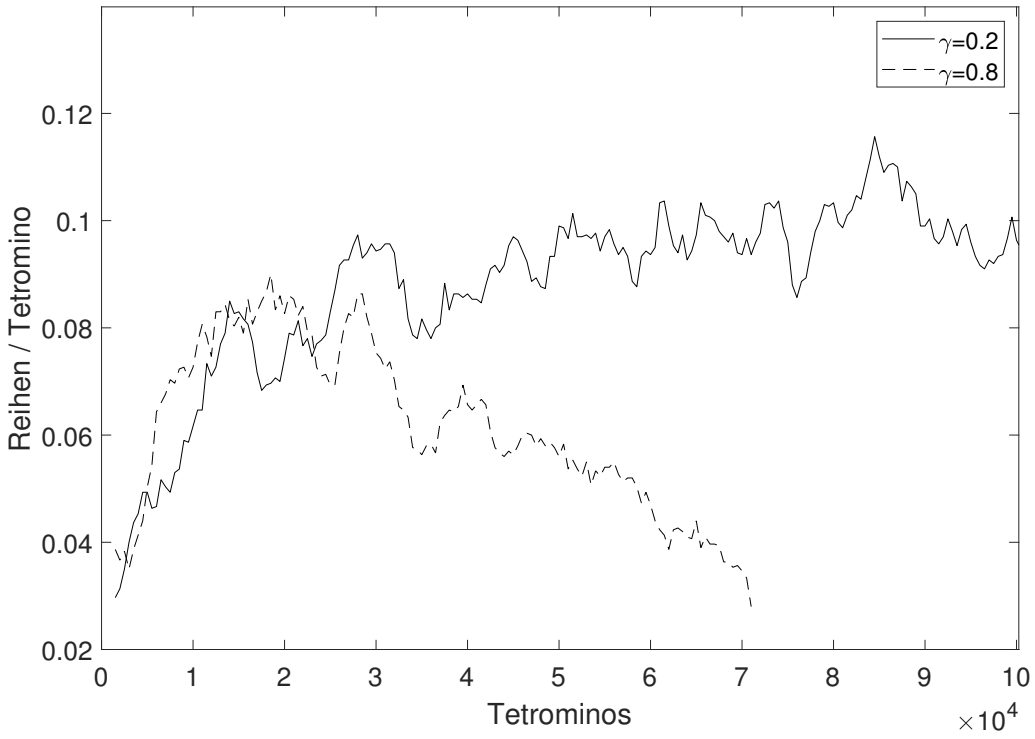

Abbildung 8: Belohnung für gelöschte Reihen

Wie zuvor erläutert, ist die einfachste Belohnungsfunktion die Implementierung einer Belohnung für das Löschen von Reihen. Abbildung 8 stellt die pro Tetromino gelöschten Reihen im über 2.500 platzierten Tetrominos gebildeten Durchschnitt für zwei unterschiedliche Werte von Gamma dar. Es wird deutlich, dass ein großes Gamma ungeeignet zum Lernen ist, da eine kurzzeitige Strategie erfolgreicher

zu sein scheint. Auch die Güte des mit $\gamma = 0{,}2$ lernenden Agenten ist mangelhaft unter Berücksichtigung, dass ein Maximum von 0,66 möglich ist und etwa 0,1 erreicht wurden. Die bei zufällig platzierten Tetrominos erreichte Güte ist selten besser als 0,03. Dennoch ist bereits ein Lernerfolg beim Agenten zu erkennen. In dem folgenden Abschnitt 6.3 wird bei der Evaluierung verschiedener Methoden zur Aggregation der Batches darauf eingegangen, wieso der mit $\gamma = 0{,}2$ lernende Agent eine vergleichsweise gute Strategie wieder zu verlernen scheint. In Abbildung 9 ist der Graph mit $\gamma = 0{,}8$ aus Abbildung 8 zum

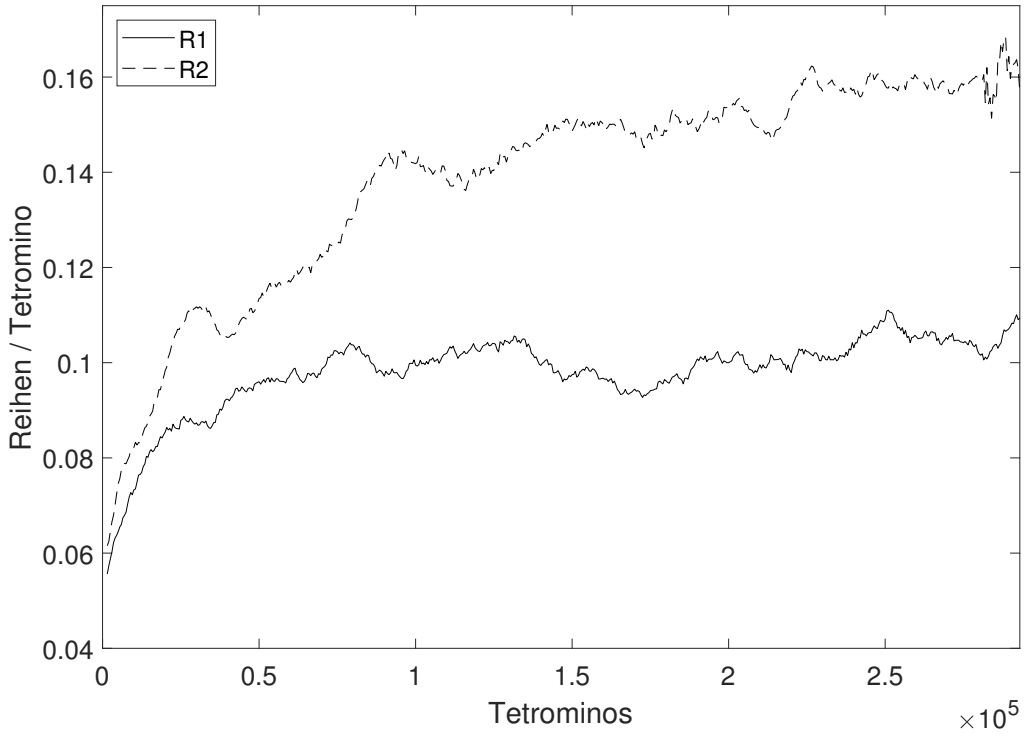

Abbildung 9: Belohnungsfunktion R1 und R2

Vergleich erneut als R1 aufgeführt. Das Hinzufügen einer Belohnung für Löcher und Höhendifferenz in R2 wirkt sich positiv auf die Güte der Strategie aus. Die implementierten Belohnungsfunktionen sind zuvor in Abschnitt 5.3 genauer erläutert. Abbildung 9 stellt die Güte im über 15.000 platzierte Tetrominos gebildeten Durchschnitt dar. Die Verbesserung im Lernverhalten ist offensichtlich. Nach $3 \cdot 10^5$ platzierten Tetrominos erreicht der Agent mit Belohnungsfunktion R2 0,16 gelöschte Reihen pro Tetromino im über 500 Tetrominos gebildeten Durchschnitt.

Bei der Betrachtung der vom Agenten verfolgten Strategie wird deutlich, dass die größte Belohnung einer Aktion nicht immer auf das Löschen einer Reihe folgen muss. Mitunter kann ein Tetromino so platziert werden, dass eine Reihe gelöscht wird, damit aber das Löschen weiterer Reihen blockiert wird. Die Belohnungsfunktionen R3 und R4 sollen durch das Vermindern der Belohnung für das Löschen von Reihen verbessert werden. Abbildung 10 stellt das Lernergebnis für Belohnungsfunktionen R2, R3 und R4 dar. Es wird deutlich, dass ein kleinerer Belohnung für das Löschen der Reihen sinnvoll ist. Keine Belohnung für das Löschen von Reihen, wie bei R4, führt zu einem schlechteren Ergebnis. Der Agent, welcher mit Belohnungsfunktion R3 lernt, erreicht ein Maximum von 0,43 nach über $6 \cdot 10^5$ platzierten Tetrominos. Die im folgenden diskutierten Verbesserungen werden einen kleineren Einfluss auf die Güte der Strategie haben, da das erzielte Ergebnis schon nahe an dem theoretisch Maximum von 0,66 Reihen pro Tetromino ist. Es wird die in Abschnitt 5.3 vorgestellte Belohnungsfunktion für die Höhendifferenz r_2 getestet. Abbildung 11 stellt die bislang beste Belohnungsfunktion R3 im Vergleich zu R5 und R6 dar.

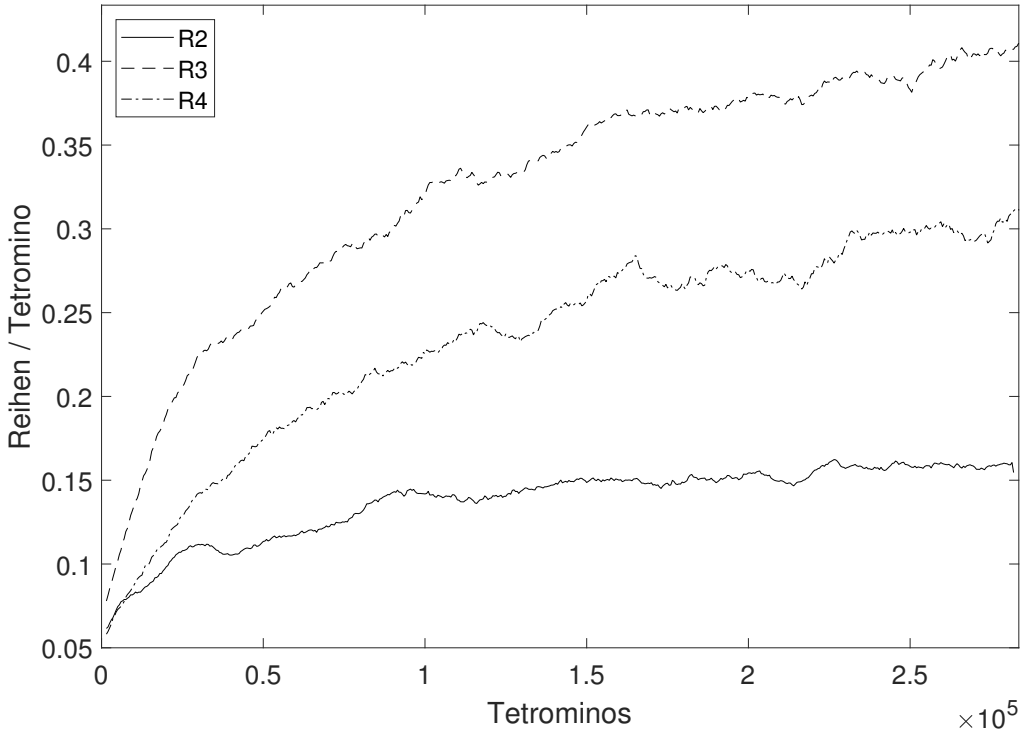

Abbildung 10: Belohnungsfunktionen R2 R3 R4

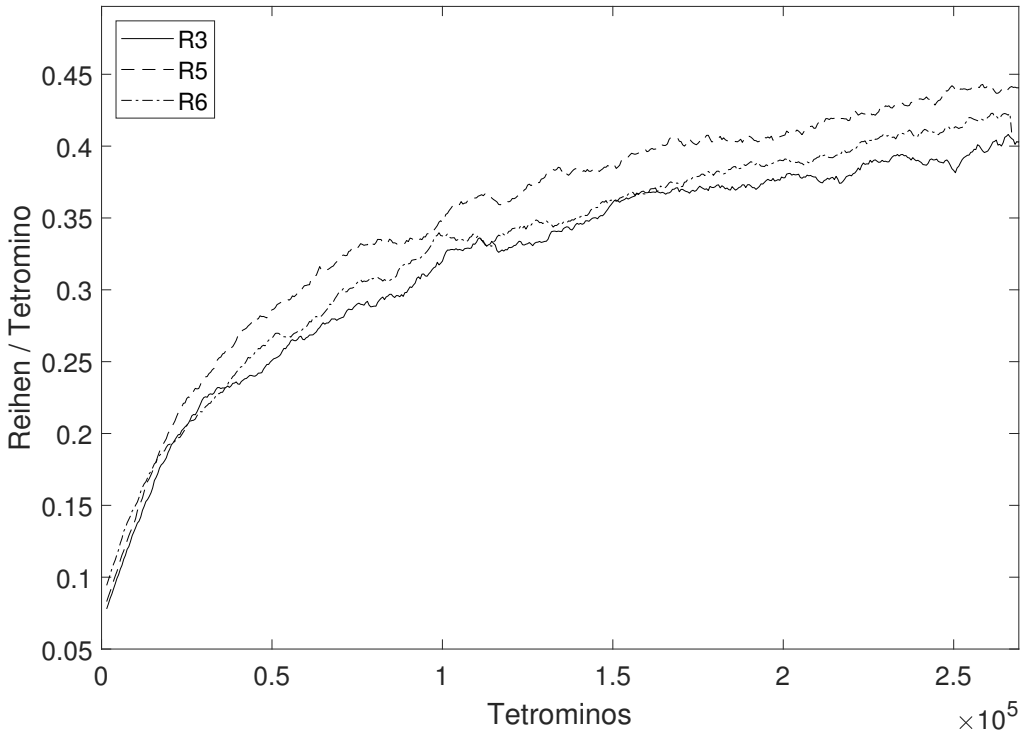

Abbildung 11: Belohnungsfunktionen R3 R5 R6

Mit der Funktion r_2 zur Berechnung der Belohnung für die Höhendifferenz lernt der Agent besser. Hier ist die Belohnungsfunktion R6 besser, welche keine Belohnung für gelöschte Reihen gibt. Dies könnte dadurch begründet werden, dass r_2 indirekt eine Belohnung für gelöschte Reihen über die resultierende negative Höhendifferenz verursacht. Die vollständigen Messreihen der Belohnungsfunktionen R3 und R6 sind zum Vergleich im Anhang unter I aufgeführt.

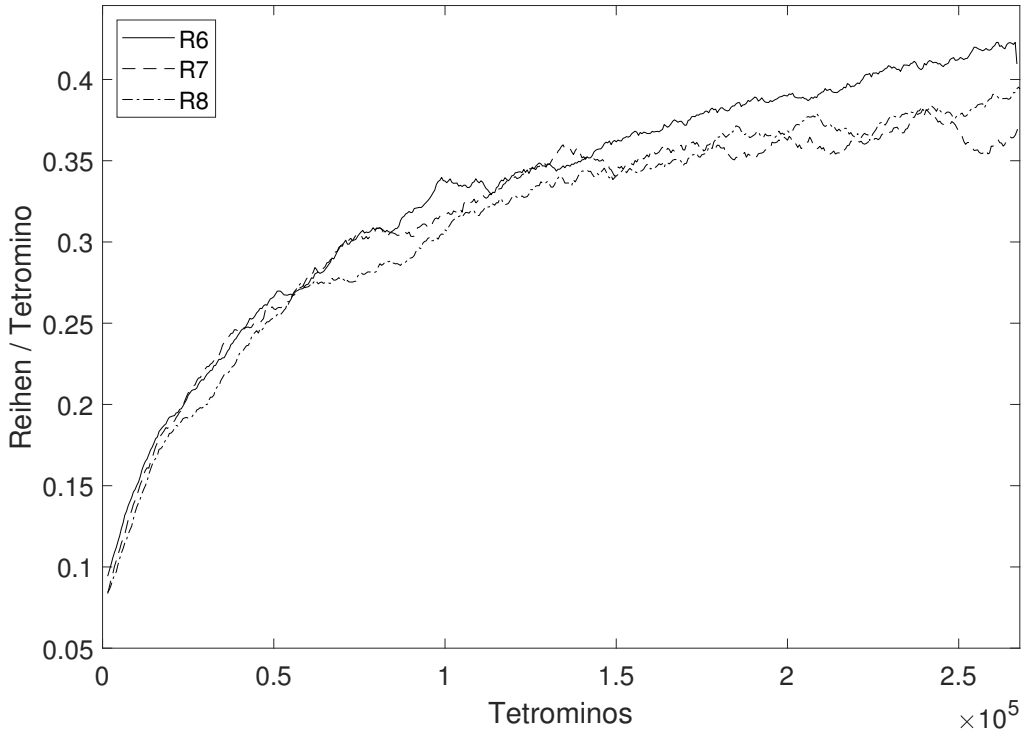

Abbildung 12: Belohnungsfunktionen R6 R7 R8

Mit R7 und R8 wird versucht die Stärken von R3 und R6 zu kombinieren, indem eine Gewichtung von r_2 eingeführt wird und es eine explizite Belohnung für gelöschte Reihen gibt. Dies ist nicht erfolgreich, wie in Abbildung 12 zu erkennen ist. Es wird angenommen, dass aus der Steigung des Graphen am Ende der jeweiligen Testreihen die noch zu erwartenden Verbesserungen bei einem Versuch über einen längeren Testzeitraum abzuschätzen sind.

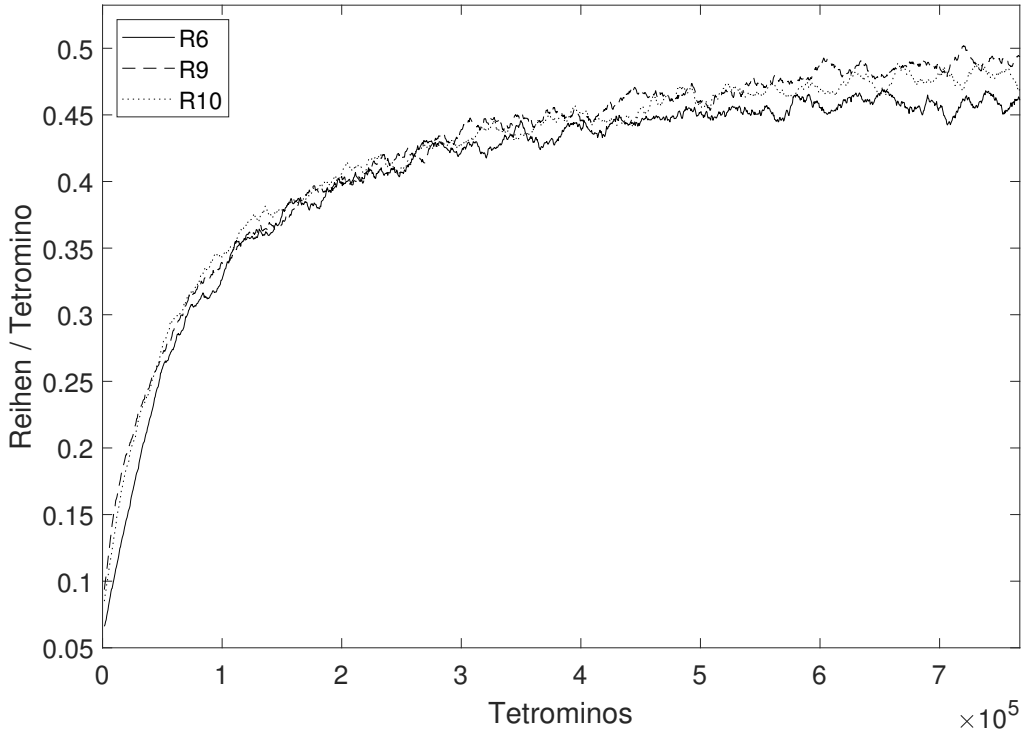

Abbildung 13: Belohnungsfunktionen R6 R9 R10

Es werden R6, R9 und R10 über einen längeren Zeitraum getestet. Nach einer Lerndauer von jeweils etwa 48 Stunden und über $8 \cdot 10^6$ platzierten Tetrominos stellt sich R9 als beste Belohnungsfunktion dar, wie in Abbildung 13 zu erkennen ist. Diese erreicht ein Maximum von 0,566 gelöschten Reihen pro platzierten Tetromino bei einem über 500 Tetrominos gebildeten Durchschnitt und etwa 0,5 gelöschten Reihen pro Tetromino bei dem in der Abbildung über 15.000 platzierten Tetrominos dargestellte Durchschnitt.

6.2. Hyperparameter

Der Hyperparameter τ wird in der Softmax-Funktion verwendet. Über ihn lässt sich die Aggressivität der Exploration-Strategie variieren. Es soll der Einfluss auf das Lernverhalten des Agenten untersucht werden. Abbildung 14 stellt den Lernfortschritt eines Agenten mit R6, $\alpha = 0{,}3$ und $\gamma = 0{,}2$ bei verschiedenen τ dar. Es zeigt sich, dass die Wahl von $\tau = 1$ als Ausgangseinstellung für andere Testreihen geeignet ist. Wie zu erwarten, verhindert das sehr kleine τ das erfolgreiche Finden neuer Strategien. Der Agent mit $\tau = 0{,}05$ kann keinen signifikanten Lernerfolg erzielen.

Die Lernrate α definiert wie stark sich die Q-Funktion mit jedem Approximationsschritt ändern kann. Abbildung 15 bildet den Lernfortschritt mit 2 exemplarischen $\alpha \leq 0{,}3$ ab. Hier wird deutlich, dass kleine Alphas beim Lernprozess eine kleinere Änderung der Approximationsfunktion verursachen und der Agent langsamer lernt. Kleinere Alpha wären eventuell bei einem Versuch über einen längeren Zeitraum vom Vorteil. Dies wurde jedoch nicht getestet. Wie bei dem Versuch mit größeren Alpha-Werten zu erkennen ist (siehe Abbildung 14), hat eine starke Änderung der Q-Funktion keinen Einfluss auf das Lernverhalten. Dies könnte durch die teilweise zufällige Bildung der Learning-Batches zurückzuführen sein. Bei großen Alpha-Werten sollten schnellere Änderungen in der Strategie zu beobachten

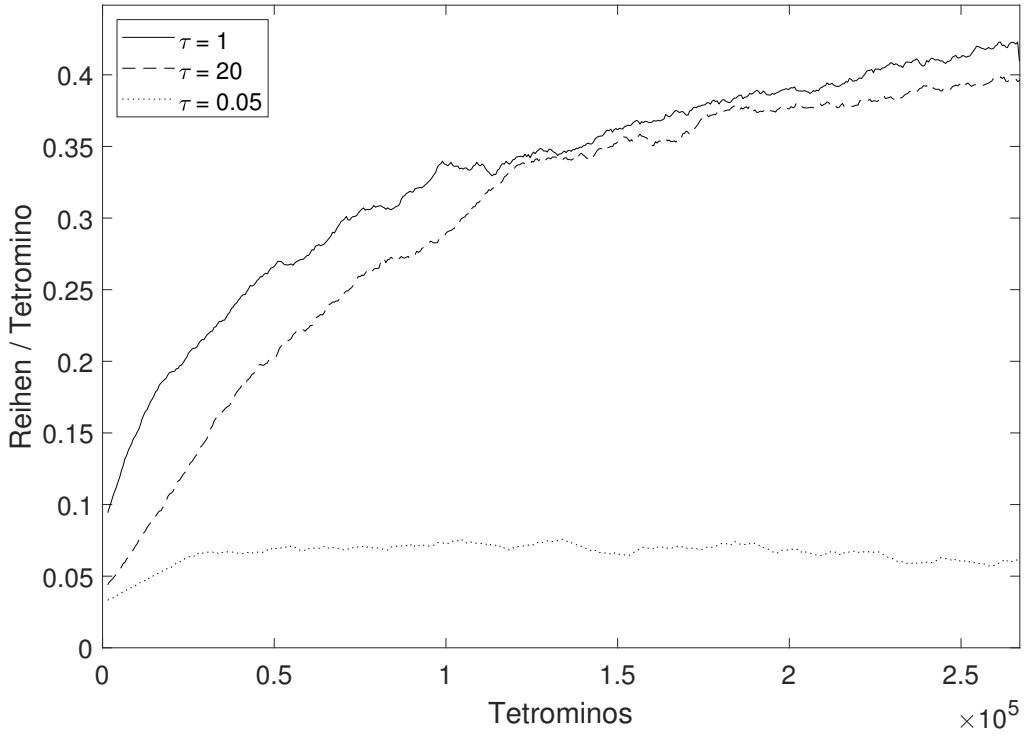

Abbildung 14: τ

sein. Diese ist jedoch nicht zu erkennen.

Der Diskontierungsfaktor Gamma scheint zunächst nur geringe Auswirkungen auf das Lernverhalten des Agenten zu haben, wie in Abbildung 17 zu erkennen ist. Bis $3 \cdot 10^5$ ist das Lernverhalten der Agenten sehr ähnlich. Gegen Ende der Testreihe scheint $\gamma = 0{,}6$ dennoch als bester getesteter Wert. Es wird davon ausgegangen, dass sich der gegen Ende des Tests beobachtete Vorteil mit $\gamma = 0{,}6$ bei längeren Testzeiten noch ausbauen würde.

6.3. Batches

Die in Abschnitt 5 beschriebenen Methoden zur Aggregation der Batches sollen im Folgenden getestet werden. Bei den vorher diskutierten Ergebnissen wurde ausschließlich die Methode B1 verwendet. Bei den im folgenden getesteten Batches wurde ausschließlich die Belohnungsfunktion R6 verwendet. Der in Abbildung 18 dargestellte Lernverlauf des Agenten B1 ist der in vorigen Tests verwendete. Die Methode des Bildens über B3, ist die unter Verwendung der *heapq.nlargest* gespeicherten Elemente, also einer festen Anzahl aller besten und schlechtesten Daten. Zunächst erscheint es nicht intuitiv, dass B3 schlechter als B1 lernt. Es ist allerdings festzustellen, dass nur selten neue Daten zu den Besten und Schlechtesten hinzukommen und der Agent so immer zu einem Teil auf den gleichen Daten lernt. Weiterhin ist anzumerken, dass die *heapq.nlargest* Funktion den Zeitaufwand pro Lernzyklus verschlechtert und somit nicht effektiv zu verwenden ist.

Ähnlich wie bei B3 verhält es sich bei dem in Abbildung 19 dargestellten B2. Hier werden zufällige Werte, welche schlechter, beziehungsweise besser, als 90% des Minimums, beziehungsweise Maximums sind, ausgewählt. Tatsächlich erfüllen nur wenige Datensätze dieses strenge Kriterium, womit dieser

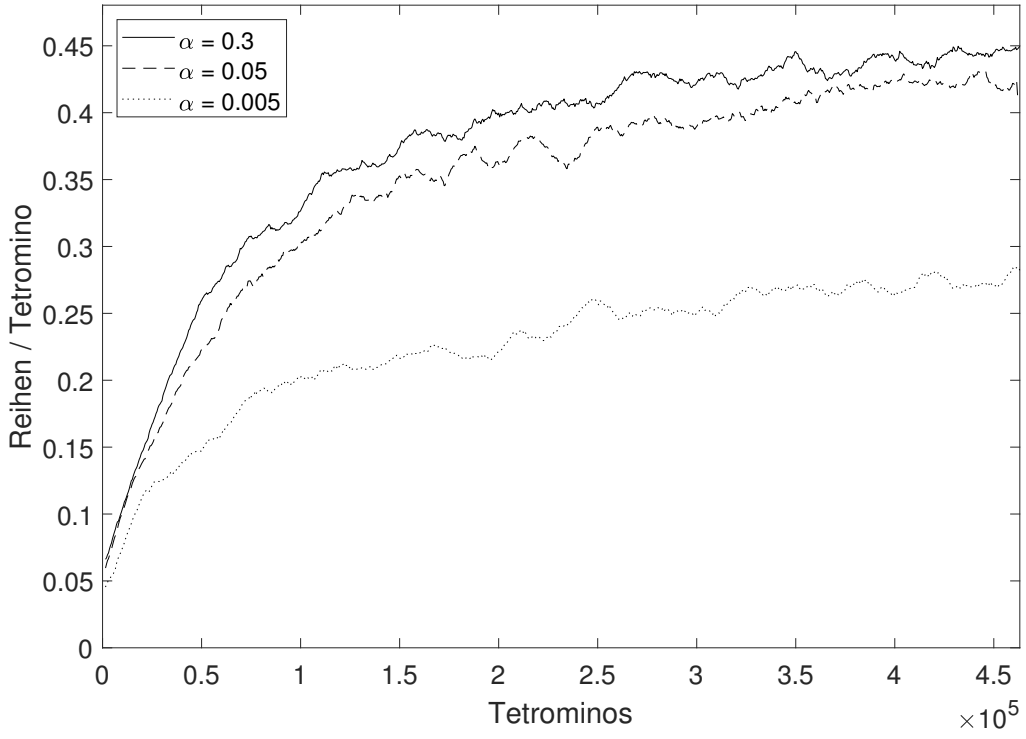

Abbildung 15: $\alpha \leq 0{,}3$

Teil des Batches zunächst nicht einmal mit der begrenzenden Anzahl von 3.000 Werten gefüllt werden kann und es nur wenige neue Werte gibt, nachdem die Grenze von 3.000 überschritten wurde, sodass auch hier häufig auf den selben Daten gelernt wird. Mit B4 und B5 wird versucht, zwischen Anzahl und Auswahl Kriterium abzuwägen, sodass häufig neue Datensätze hinzukommen, aber auch gleichzeitig die relevantesten Daten verwendet werden.

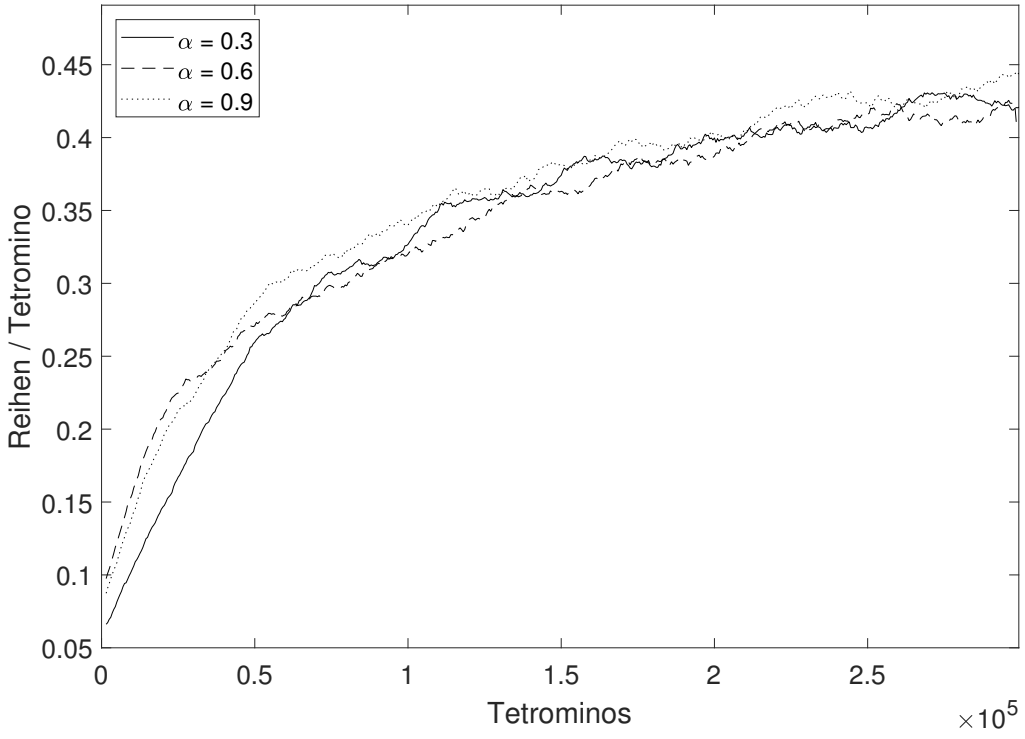

Abbildung 16: $\alpha \geq 0{,}3$

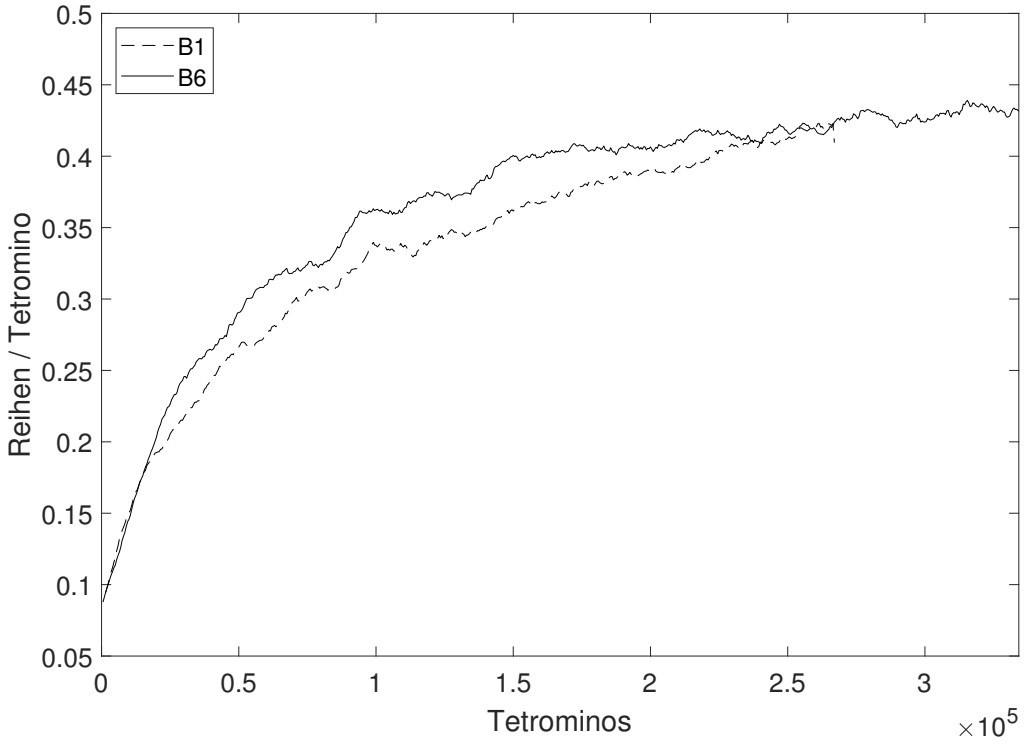

Abbildung 21: Batches B1 und B6

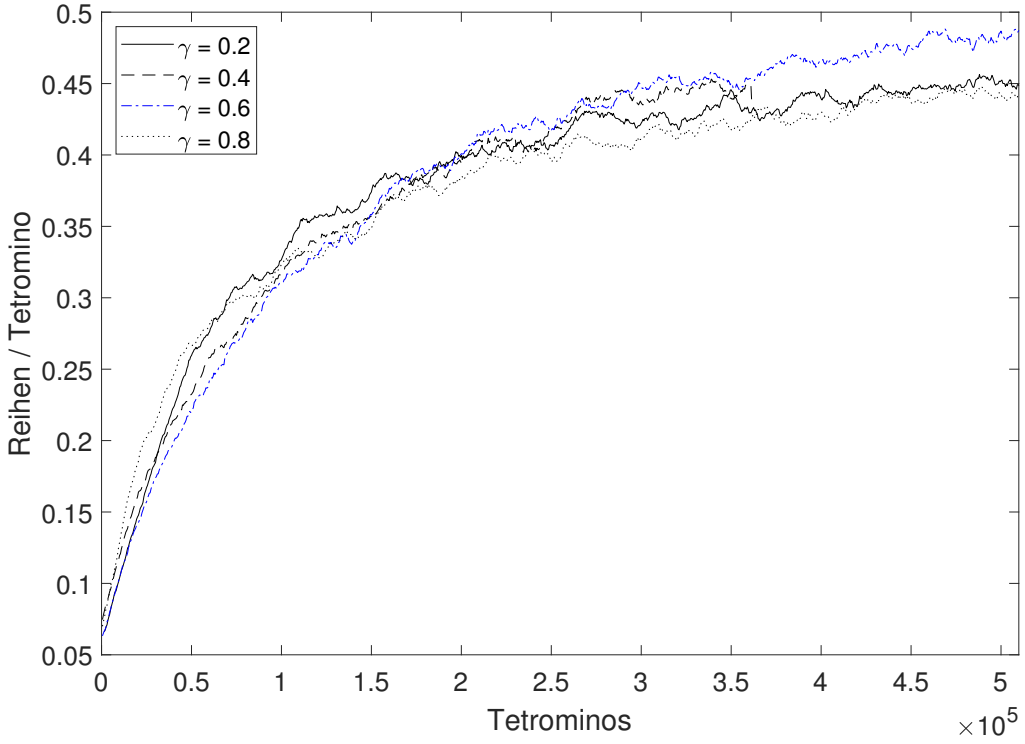

Abbildung 17: γ

In Abbildung 21 ist zu erkennen, dass das Bilden der Batches über die neu hinzugekommenen Daten erfolgversprechend ist. Es wird daraufhin die Methode B7 entwickelt, bei der die neusten Daten immer zum Bilden des Batches verwendet werden und nicht durch Zufall aussortiert werden können, wie es bei einem größeren Trainingsdatensatz geschieht. Dies ist in Abbildung 21 bei über $2{,}5 \cdot 10^5$ Tetrominos zu erkennen. Der Unterschied zwischen B6 und B1 und damit auch zu B4 verschwindet.

Nach der Beschreibung der verwendeten Batches wird ein möglicher Grund deutlich, wieso die Belohnungsfunktion R1 nicht lernfähig war. Die in Abbildung 8 und 9 dargestellte Testreihe wurde mit dem Verfahren zur Bildung der Batches B1 aufgenommen. Mit dem größer werdenden Trainingsdatensatz wird bei einer zufälligen Auswahl der Testdaten die Wahrscheinlichkeit geringer, einen Datensatz zu finden, welchem eine positive Belohnung beim Löschen einer Reihe zugeordnet wird.

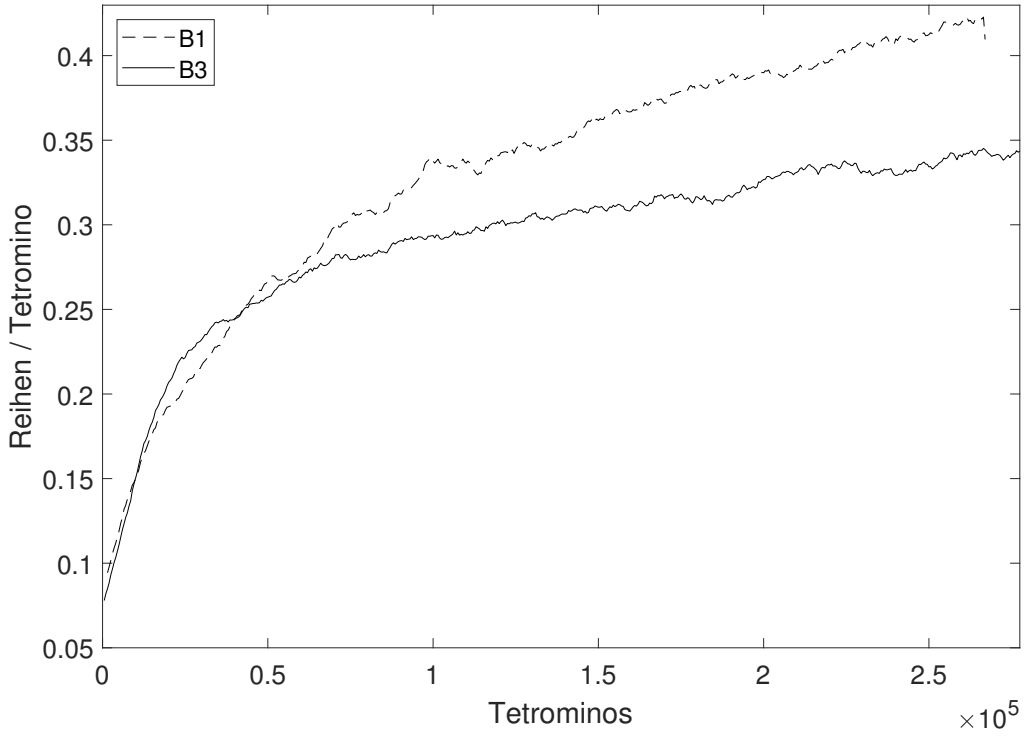

Abbildung 18: Batches B1 und B3

6.4. Information über den nächsten Tetromino

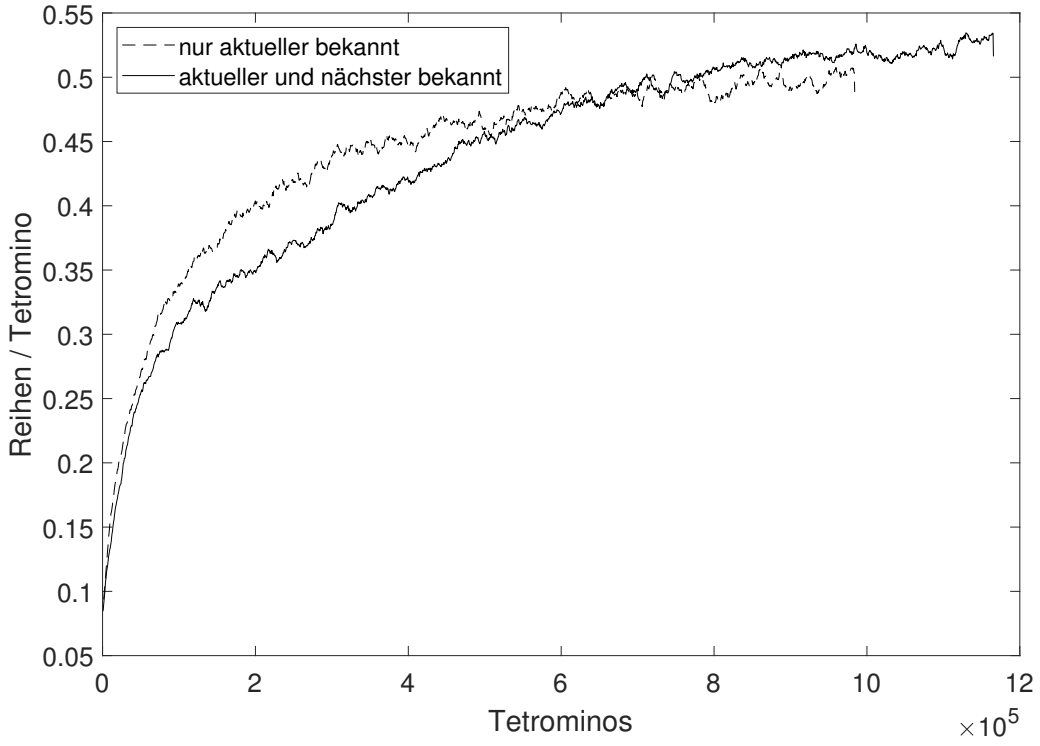

Abbildung 22: Information über nächsten bekannten Tetromino

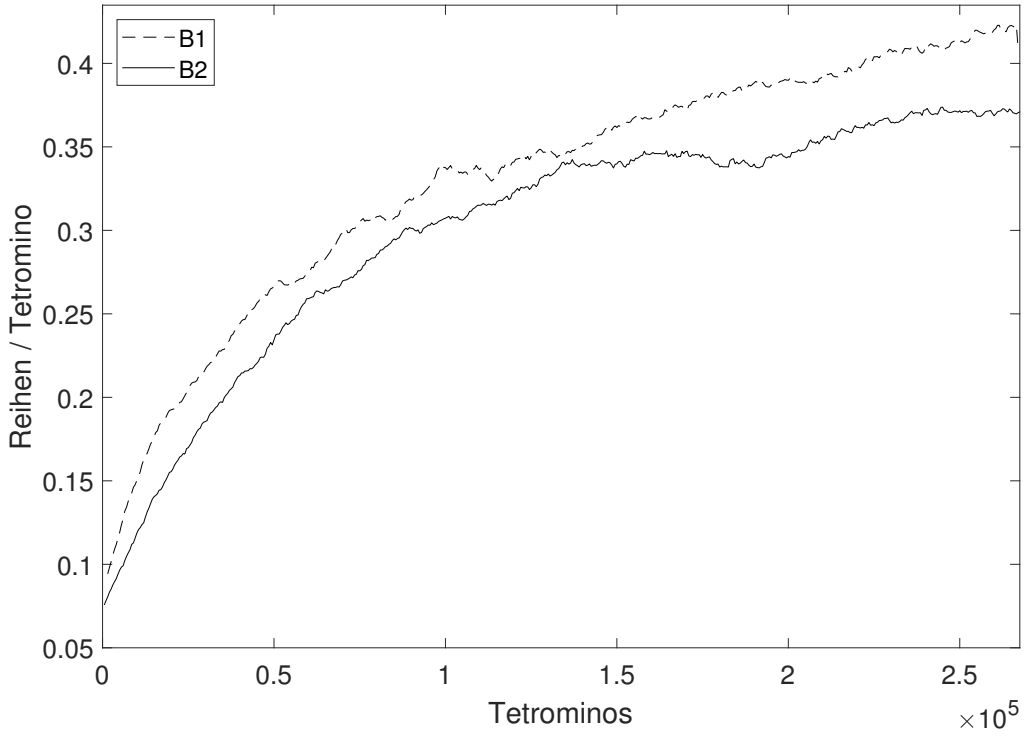

Abbildung 19: Batches B1 und B2

Es wird im Weiteren versucht, den Agenten so zu trainieren, dass er eine möglichst gute, eventuell sogar eine perfekte, Strategie erreicht. Beim klassischen Tetris ist der nächste zu platzierende Tetromino dem Spieler bekannt. Diese Information wird nun auch dem Agenten bekannt gemacht. Die in Abbildung 22 dargestellten Graphen zeigen das Lernverhalten des Agenten mit R9. Es ist eine zuvor noch nicht beobachtete Beziehung zwischen den Lernkurven der Agenten zu erkennen. Bis zu diesem Punkt ist es Agenten, welche zu Beginn langsamer gelernt haben, nicht gelungen andere Agenten einzuholen. Dies könnte auf die Beschränkung der Tests auf kürzere Zeiträume zurückzuführen sein. Eine alternative Hypothese ist, dass der Agent, welchem der nächste Tetromino bekannt ist, einen größeren Zustandsraum hat und deswegen zunächst langsamer lernt.

Die zusätzliche Information ist aber relevant für die Adaption einer besseren Strategie. Nach [14] kann die Information über den nächsten Stein die Güte der Strategie um mehrere Größenordnungen verbessern, da nun auch Strategien verfolgt werden, welche zunächst einen äußerst schlechten Zustand herbeiführen, wenn bekannt ist, dass der nächste Stein die Situation wieder beheben kann. Diese Beobachtung konnte im konkreten Fall nicht gemacht werden.

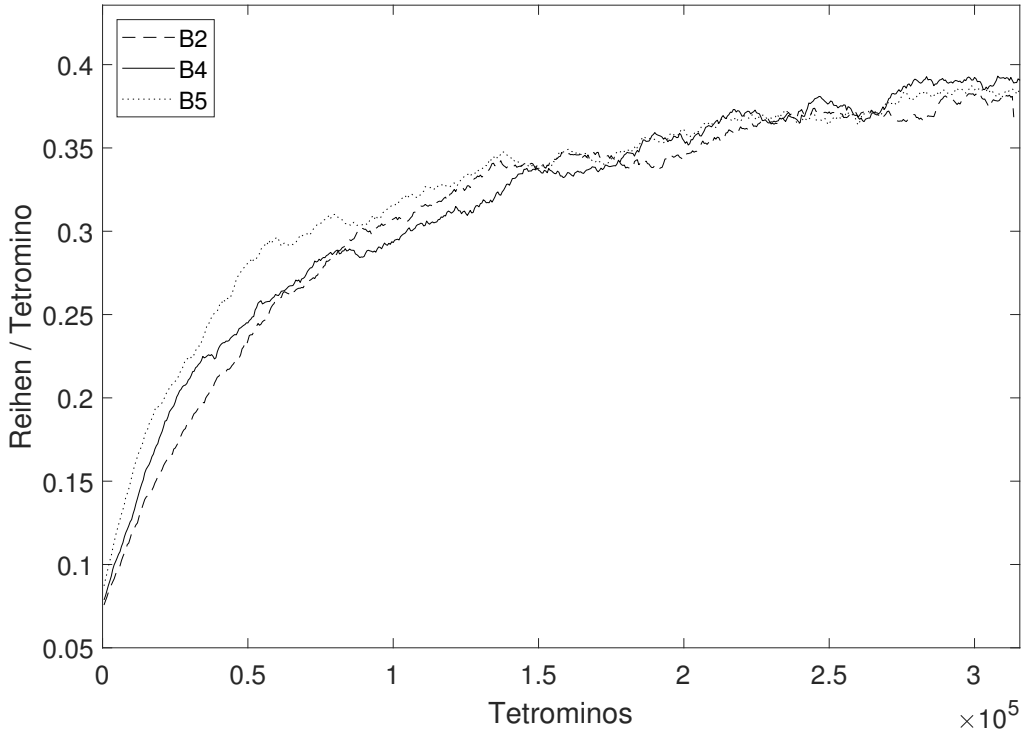

Abbildung 20: Batches B2, B4 und B5

6.5. Frequenz der Lernzyklen

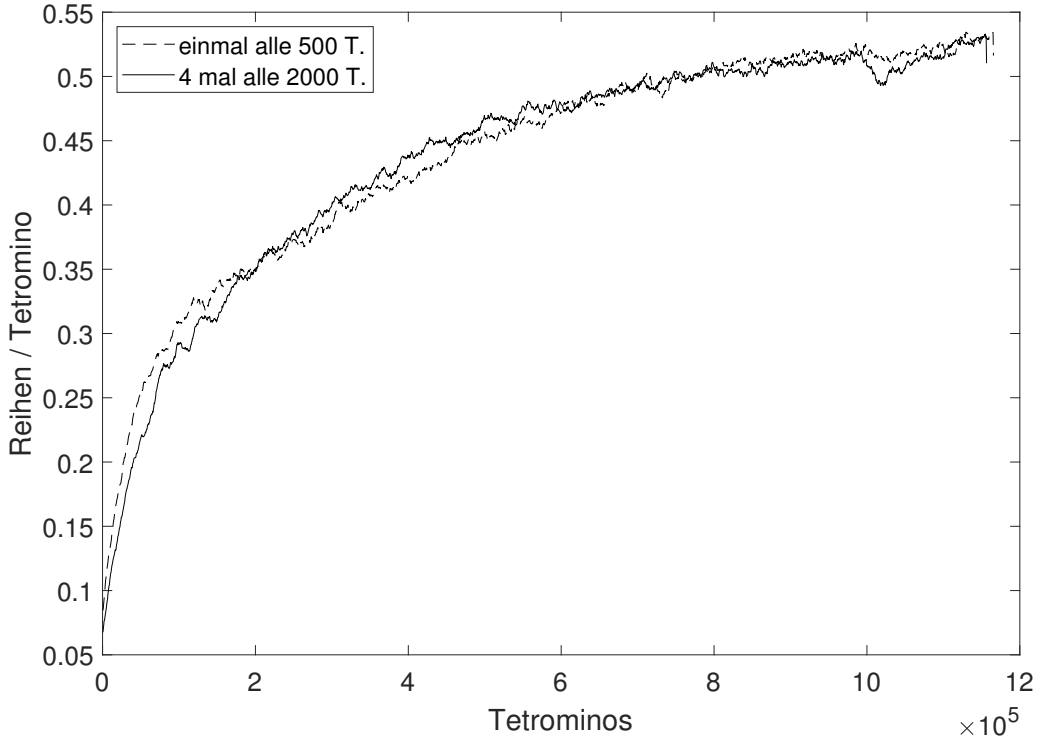

Abbildung 23: Frequenz der Lernzyklen

In den zuvor diskutierten Ergebnissen wird die Q-Funktion aktualisiert, sobald jeweils 500 Tetrominos hinzugekommen sind. Dies beinhaltet das Bilden der Batches, wie oben beschrieben, das Erzeugen der Q-Funktion und das fitten des neuronalen Netzes an diese. In Abbildung 23 wird die Q-Funktion aktualisiert, sobald jeweils 2.000 Tetrominos hinzugekommen sind. Der mittlere Zeitaufwand pro platziertem Tetromino bleibt dabei konstant und ermöglicht den Vergleich von der zuvor dargestellten Messreihe, bei welcher die Information des nächsten Tetrominos hinzugenommen wurde. Aus den bis hierhin dargestellten Tests wird weiterhin deutlich, dass kürzere Messreihen weniger relevant werden, da sie sich zu Beginn des Trainings nicht mehr deutlich unterscheiden lassen. Im Folgenden soll die im Rahmen dieser Arbeit maximal zu erreichende Güte der Strategie ermittelt werden. Der in Abbildung 23 dargestellte Agent erreicht nach 10^6 platzierten Tetrominos bereits einen Durchschnitt von 0,533 gelöschten Reihen pro Tetromino.

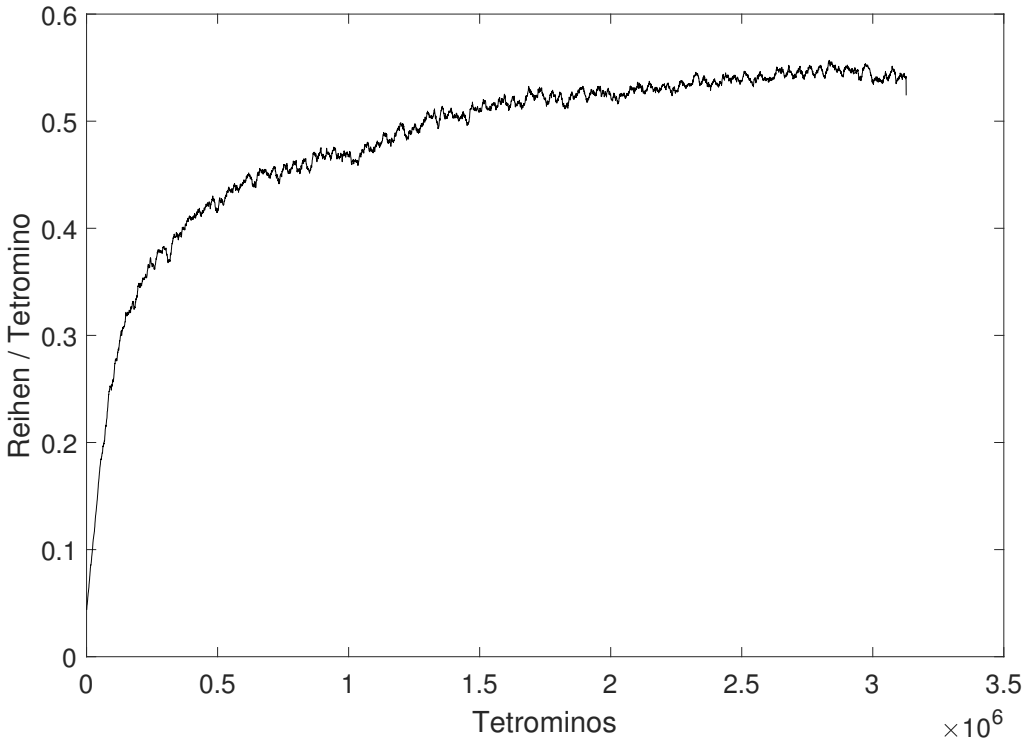

Abbildung 24: Lernverhalten über $3 \cdot 10^6$ platzierte Tetrominos

Im Folgenden sind diverse längere Testreihen mit unterschiedlichen Parametern aufgenommen worden. Dabei ist die perfekte Strategie von 0,66 gelöschten Reihen pro Tetromino nicht erreicht worden. Die beste Strategie erreichte der Agent mit dem in Abbildung 24 abgebildeten Lernverhalten. Der Durchschnitt über 15.000 platzierte Tetrominos lag dabei bei 0,557 gelöschten Reihen pro Tetromino. Es ist dabei anzumerken, das andere getestete Agenten ebenfalls eine ähnlich gute Strategie, teilweise sogar nach weniger platzierten Tetrominos erreichten. Der Agent aus Abbildung 24 erzielte aber den besten Lernerfolg, da dieser auch am längsten getestet wurde. Dies war möglich, da er nur alle 2.000 platzierte Tetrominos eine Aktualisierung der Q-Funktion durchführte und somit viermal schneller pro platziertem Tetromino ist, als die zuvor getesteten Agenten. Der dargestellte Agent arbeitete mit der Belohnungsfunktion R9 und bildet die Batches wie in B7 beschrieben. Die Hyperparameter waren $\alpha = 0{,}3$, $\gamma = 0{,}8$ und $\tau = 5$. Bei einem Spiel, bei dem der Agent verlieren konnte, schaffte er es ca. 60 Reihen pro Spiel zu löschen. Dies ermöglicht allerdings nur unter Einschränkungen einen Vergleich mit den in Abschnitt 3 beschriebenen Agenten, da die bis hierhin diskutierte Implementierung mit einer Spielfeldbreite von 6 arbeitet.

6.6. Spielfeldgröße

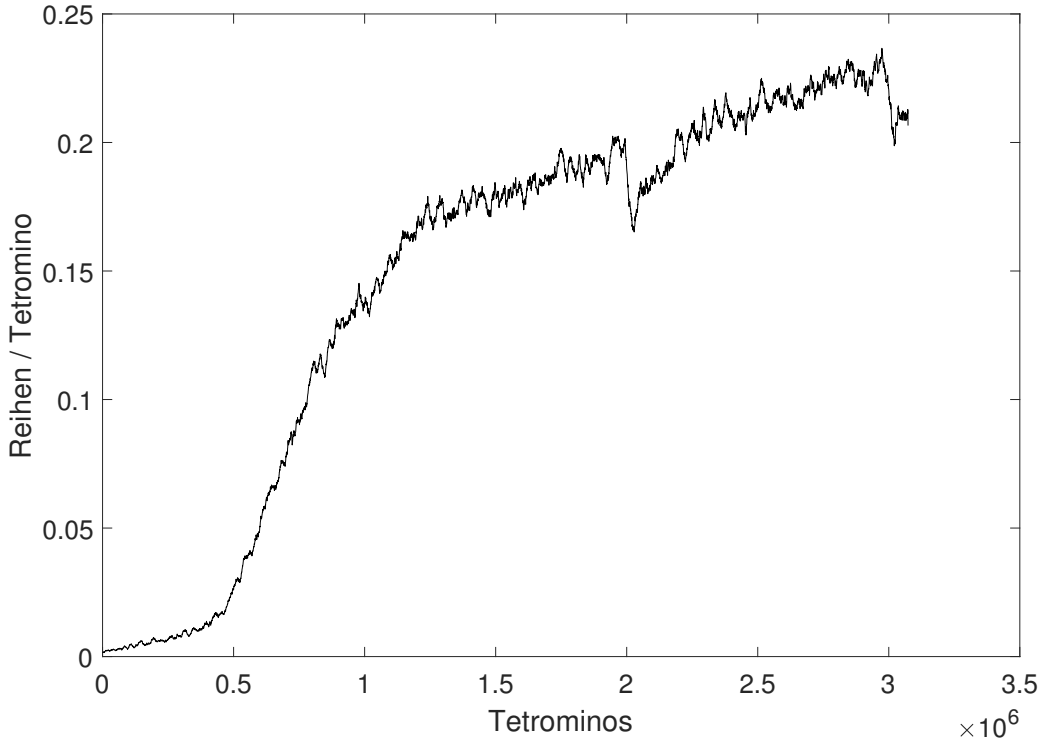

Abbildung 25: Spielfeldbreite 10

Abschließend soll getestet werden, ob der Agent auch auf dem originalen Spiel mit einer Spielfeldbreite von 10 lernen kann. Dazu wird der im vorigen Abschnitt vorgestellte Agent verwendet. Abbildung 25 stellt das Lernverhalten des Agenten dar. Es fällt auf, dass der im Graphen dargestellte Lernverlauf eher einem beschränktem logistischen Wachstum zugeordnet werden kann, als, wie bei dem kleineren Spielfeld, einem beschränktem exponentiellen Wachstum. Dies ist darauf zurückzuführen, dass es bei der Initialisierung mit zufällig platzierten Tetrominos seltener eine gute Aktion gibt, wenn die Spielfeldbreite größer wird. Auffällig ist, dass der Agent bei $5{\cdot}10^5$ platzierten Tetrominos einen Wendepunkt zu erreichen scheint und Strategie danach deutlich schneller besser wird. Auch fallen die Einbrüche in der Güte der Strategie bei $2 \cdot 10^6$ und $3 \cdot 10^6$ auf, welche mit dem Zeitpunkt des Überlaufes des Speichers korrelieren und auf einen Bug im Code zurückzuführen sind, welcher bei einer Spielfeldbreite von 6 bereits behoben wurde, wie in Abbildung 24 zu erkennen ist. Der Agent erreicht im über 15.000 platzierte Tetrominos gebildeten Durschnitt 0,237 gelöschte Reihen pro Tetromino bei maximal möglichen 0,4. Der Agent erreicht im Durchschnitt 16 gelöschte Reihen vor dem Verlust, welches deutlich schlechter als ein vom Menschen gespieltes Spiel ist.

7. Fazit

Das formulierte Ziel zu Beginn dieses Projektes war:

"[...] die Implementierung eines lernenden Agenten, der mittels Bestärkenden Lernens das Spiel Tetris erlernt."

Ein solcher Agent konnte im Rahmen dieses Projektes entwickelt und implementiert werden. Der Lernerfolg für verschiedene Parameter wurde in Kapitel 6 vorgestellt und beschrieben. Es wurde der Einfluss ausgewählter Freiheitsgrade des Systems auf den Lernerfolg des Agenten untersucht. Den größten Einfluss hat die Belohnungsfunktion.

In dem Spiel Tetris werden Punkte dafür vergeben, wie viele Reihen gelöscht wurden. Es konnte gezeigt werden, dass eine solche Punktevergabe nicht als Belohnungsfunktion geeignet ist. Es muss das ganze Spielfeld berücksichtigt werden, um die vom Agenten gewählte Aktion zu bewerten. Dabei hat sich herausgestellt, dass die in Tabelle 4 vorgestellte Belohnungsfunktion R6 das beste Ergebnis erzielte. Das Löschen einzelner Reihen wurde dabei nicht direkt belohnt. Vielmehr wurde eine möglichst glatte Oberfläche und eine geringe Bauhöhe belohnt, wodurch natürlich auch indirekt das Löschen von Reihen beachtet wird.

Für die Hyperparameter konnte gezeigt werden, dass eine hohe Lernrate (großes Alpha) für die ausgeführten Aktionen zu besseren Ergebnissen geführt hat. Dagegen war der Einfluss des Parameters Gamma kaum zu bemerken, erst nach sehr langen Lernzeiten konnte ein Gamma von 0,6 als bester getesteter Wert ermittelt werden.

Im theoretisch besten Fall ist es möglich 0,66 Reihen pro platzierten Tetromino zu löschen. Es konnte gezeigt werden, dass einer der getesteten Agent ca. 0,53 Reihen mit jedem Tetromino löscht. Der Agent hat das Spiel somit noch nicht perfekt gelernt, aber er kommt schon verhältnismäßig nahe an den optimalen Fall heran. Allerdings lassen die in Kapitel 6 dargestellten Abbildungen der Lernkurven darauf schließen, dass der Agent mit steigender Lernzeit nicht mehr wesentlich besser werden wird und gegen eine obere Schranke konvergiert.

Eine mögliche Fehlerquelle die den Agenten nicht die optimale Strategie lernen lässt ist, dass nicht auf dem tatsächlichen Spielfeld gelernt wird, sondern auf der Kontur. Die Kontur erlaubt es dem Agenten nicht in der zweiten Ebene liegende Löcher zu sehen. Ein Agent kann ggf. von der Kontur her einen Stein so platzieren, dass scheinbar eine Reihe vervollständigt ist, allerdings muss das nicht wirklich so sein, wenn für die Kontur nicht sichtbare Löcher vorhanden sind. Weiterhin wird bei der Kontur nur eine Maximale Höhendifferenz von 3 Feldern berücksichtigt.

Die Beschreibung der vom Agenten wahrgenommenen Status über Konturen resultiert in zwei negative Effekte. Zunächst kann der Agent nie wissen, an welcher Stelle beim Löschen von Reihen, eine Stelle freigelassen werden muss, damit das darunter liegende Loch freigelegt wird. Das Wissen über Positionen von Löchern spielt bei einem menschlichen Spieler eine wesentliche Rolle.

Zweitens sind die vom Agenten wahrgenommenen Zustandsübergänge nicht mehr deterministisch. Beim Ausführen von gleichen Aktionen bei gleichen Zuständen können unterschiedliche Zustände resultieren. Außerdem resultieren entsprechend stark unterschiedliche Rewards. Dies ist auch ein weiterer Grund weshalb das ausschließliche Belohnen von gelöschten Reihen nicht Ziel führend ist.

Gegen Ende der Testreihen wurde untersucht, wie viele Reihen der Agent löschen kann, bevor er das Spiel verliert. Dabei konnte ein durchschnittliches Ergebnis von ca. 60 gelöschten Reihen bis

zum Verlieren erreicht werden. Dabei sollte beachtet werden, dass das Spielfeld eine Breite von sechs Blöcken hat. Es ist demnach wesentlich schwieriger das Spiel zu spielen. Das Ergebnis ist mit dem eines ungeübten Spielers vergleichbar und von der Geschwindigkeit der fallenden Tetrominos abhängig, welche für den Softwareagenten natürlich irrelevant ist.

Bei einer Spielfeldbreite von zehn konnte der Agent dagegen gerade einmal 16 Reihen löschen, was im Vergleich zu menschlichen Spielern sehr schlecht ist. Hier ist interessant, dass für einen menschlichen Spieler das spielen auf einem Spielfeld mit der Breite sechs deutlich schwieriger ist, als auf einem Spielfeld mit der Breite zehn. Für den lernenden Agenten scheint sich dies auf Grund des größeren Zustandsraumes genau gegenteilig zu verhalten. Der Agent muss wesentlich länger lernen um einen anfänglichen Lernerfolg aufzuweisen.

Aus diesem Grund ist die Motivation entstanden die Hauptarbeit in Bezug auf Parameteroptimierung und Untersuchung diverser Ergebnisse auf der Basis des auf der Spielfeldbreite von sechs trainierten Agenten durchzuführen. Hierdurch ließen sich eine größere Anzahl von verschiedenen Tests durchführen. Die gewonnenen Erkenntnisse lassen sich direkt auch auf eine größere Spielfeldbreite übertragen. Weiterhin war der durch die Untersuchung einer größeren Menge an Ergebnissen durchlaufene, didaktische Prozess ergiebiger.

Beim Vergleich mit der besten dem Autoren bekannten Implementierung, welche Aktionen direkt über die Q-Funktion bestimmt, erscheint die erreichte Leistung des Agentens auf dem Spielfeld mit der Breite 10 akzeptabel. Unter [16] wurden 18 gelöschte Reihen erreicht. Der hauptsächliche Unterschied im Konzept war hier, dass auf dem gesamten Spielfeld, welches über Convolutional Neural Networks verarbeitet wird, gelernt wurde. Das hier erzielte Ergebnis von 16 gelöschten Reihen scheint dieses Ergebnis zu bestätigen.

8. Ausblick

8.1. Spielfeldbreite 10

Das Training über den anfänglichen Lernerfolg hinaus konnte bei dem mit der Spielfeldbreite von zehn trainierten Agenten aufgrund des zeitlichen Rahmens des Projektes nicht extensiv untersucht werden und nicht bis zur Konvergenz gegen eine Schranke in der Güte der Strategie beobachtet werden. Ebenfalls war es aufgrund der zeitlichen Beschränkung nicht mehr möglich eine umfassende Untersuchung der Einflüsse diverser Hyperparameter, ähnlich der mit Spielfeldbreite 6, durchzuführen. Es lassen sich allerdings verschiedene Aspekte diskutieren, welche einen Ausblick auf den zukünftigen Lernerfolg des Agenten ermöglichen.

Zum Beispiel lassen sich Schlüsse aus der Initialisierung des Agenten mit zufällig platzierten Tetrominos ziehen. Beide Agenten wurden mit 500 zufällig platzierten Tetrominos initialisiert. Bei einer Spielfeldbreite von sechs werden dabei im Schnitt etwa zwölf Reihen gelöscht, bei der Spielfeldbreite von zehn zwei. Daraus lässt sich schließen, dass auf einem kleineren Spielfeld, alleine durch Zufall eine bessere Strategie erspielt werden kann. Die 12 gelöschten Reihen auf 500 Tetrominos entsprechen 0,024 gelöschten Reihen pro Tetromino. Betrachtet man die nicht geglätteten original Daten eines trainierten Agenten, so wird man feststellen, dass dieser Wert deutlich innerhalb der Standardabweichung liegt. Der Effekt, dass bei einer Spielfeldbreite von 10 seltener Reihen durch Zufall gelöscht werden, ist also nur für die in der Initialisierungsphase erzeugten Datensätze und den kurzfristigen Lernerfolg relevant.

Deutlich größeren Einfluss auf die bei einer Spielfeldbreite von 10 zu erwartende Schranke für die Güte der Strategie hat die Wahrnehmung des Spielfelds über die Kontur. Bei einem größeren Spielfeld müssen mehr Steine zum Löschen einer Reihe platziert werden. Die Wahrscheinlichkeit, dass eine verdeckte Reihe ein Loch enthält ist demnach wesentlich größer. Dies ist in der resultierenden Strategie des Agenten zu beobachten. Anstatt Tetrominos möglichst „tief" in die Struktur einzubauen und so mehrere Reihen auf einmal zu löschen versucht der Agent eher die für ihn sichtbare oberste Reihe zu löschen. Diese Strategie ist bei dem auf der Spielfeldbreite von sechs trainierten Agenten ebenfalls zu beobachten, allerdings deutlich weniger ausgeprägt.

8.2. Optimierungspotential

Abschließend lässt sich das weitere Optimierungspotential im Implementierten Q-Learning Agenten diskutieren. Im Verlaufe der Arbeit wurden bereits diverse Ansätze diskutiert, welche vielversprechend für die konkrete Anwendung erscheinen.

Bei Betrachtung des in Abbildung 25 dargestellten Lernverhaltens bei der Spielfeldbreite 10 werden zunächst offensichtliche Maßnahmen zur Optimierung der besten Strategie deutlich. Im Lernverhalten ist zwischen $1,5 \cdot 10^6$ und $3 \cdot 10^6$ ein annähernd linearer Anstieg der über die gelöschten Reihen pro platziertem Tetromino definierten Güte der Strategie zu erkennen. Auch ohne Behebung der Fehler beim Speicherüberlauf wäre schon durch längeres Training eine bessere Strategie zu erzielen. Auch bei der Implementierung mit der Spielfeldbreite sechs sind Implementierungen vorhanden, welche durch weiteres Training verbessert werden könnten.

Weiteres Optimierungspotenzial ist bereits bei der Ausarbeitung identifiziert und bei der Implementierung berücksichtigt worden. So wurde zum Beispiel das Initialisieren des auf einer Spielfeldbreite

von 10 spielenden Agenten mit vom Menschen eingespielten Daten nicht getestet, obwohl diese Funktion bereits implementiert ist. Dies könnte die anfänglichen Schwierigkeiten des Agenten beim Lernen verhindern.

Weiterhin wurde bereits die Rotationssymmetrie der Tetrominos diskutiert. Es würde sich entsprechend eine Begrenzung der Explorationsstrategie in Abhängigkeit der Tetrominoform und eine entsprechende Einschränkung des Aktionsraums anbieten. Die konkrete technische Umsetzung müsste im Weiteren untersucht werden.

Die Änderung der Approximationsfunktion von einem neuronalen Netzwerk zu einem Decision Tree wurde bereits als vielversprechend identifiziert, aber noch nicht implementiert. Auch generelle Änderungen an der Struktur oder dem Lernverhalten des Netzwerks sind nicht umfassend untersucht worden und könnten Optimierungspotential bieten. Eine der größten Einschränkungen des implementierten Agenten ist die Wahrnehmung des Spielfelds über Konturen. Diese könnte durch das Lernen auf dem gesamten Spielfeld als Status eliminiert werden. Letzteres würde zur Umsetzung voraussichtlich Deep Learning Ansätze erfordern.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Wikipedia: *Tetris*. https://en.wikipedia.org/wiki/Tetris. – Eingesehen am 10.1.2019

[2] Erik D. Demaine, Susan H. ; Liben-Nowell, David: *Tetris is Hard, Even to Approximate*. http://erikdemaine.org/papers/Tetris_COCOON2003/paper.pdf. – Eingesehen am 10.1.2019

[3] Kuna, Karol: *Learning to Play Tetris using Reinforcement Learning*. https://nlp.fi.muni.cz/uiprojekt/ui/kuna_karol2016/tetris-documentation.pdf. Version: 2016. – Eingesehen am 10.1.2019

[4] Melax: *Reinforcement Learning Tetris Example*. http://melax.github.io/tetris/tetris.html. – Eingesehen am 10.1.2019

[5] Carr, Donald: *Applying reinforcement learning to Tetris*. https://pdfs.semanticscholar.org/644b/3b345759b7e36c87f9c6d247e9733553dff7.pdf. Version: 2005. – Eingesehen am 10.1.2019

[6] Aggarwal, Charu C.: *Neural Networks and Deep Learning*. Cham, Switzerland : Springer International Publishing AG, 2018. – ISBN 978–3–319–94462–3

[7] Frochte, Jörg: *Maschinelles Lernen - Grundlagen und Algorithmen in Python*. New York : Hanser, Carl GmbH + Company, 2018. – ISBN 978–3–446–45291–6

[8] Lapan, Maxim: *Deep Reinforcement Learning Hands-On*. Packt Publishing Ltd., 2018. – ISBN 978–1–78883–424–7

[9] : *Outlines and Highlights for Artificial Intelligence by Stuart Russell, Isbn - 9780136042594 0136042597*. New York : Cram101 Incorporated, 2011. – ISBN 978–1–614–90130–3

[10] Kulkarni, Parag: *Reinforcement and Systemic Machine Learning for Decision Making*. IEEE Press, Wiley, 2012. – ISBN 978–0–470–91999–6

[11] Buduma, Nikhil: *Fundamentals of Deep Learning*. O'Reilly, 2018. – ISBN 978–1–491–92561–4

[12] Böhm, Niko ; Kokai, Gabriella ; Mandl, Stefan: *An Evolutionary Approach to Tetris*. https://www2.informatik.uni-erlangen.de/publication/download/mic.pdf. – Eingesehen am 30.1.2018

[13] Yijuan, Lee: *Tetris AI – The (Near) Perfect Bot*. https://codemyroad.wordpress.com/2013/04/14/tetris-ai-the-near-perfect-player/. – Eingesehen am 30.1.2019

[14] Fahey, Colin: *Tetris*. https://www.colinfahey.com/tetris/tetris.html. Version: 2003. – Eingesehen am 30.1.2019

[15] Lundgaard, Nicholas ; McKee, Brian: *Reinforcement Learning and Neural Networks for Tetris*. http://www.mcgovern-fagg.org/amy/courses/cs5033_fall2007/Lundgaard_McKee.pdf. – Eingesehen am 30.1.2019

[16] Stevens, Matt ; Pradh, Sabeek: *Playing Tetris with Deep Reinforcement Learning*. http://cs231n.stanford.edu/reports/2016/pdfs/121_Report.pdf. – Eingesehen am 30.1.2019

[17] Burgiel, Heidi: *How to lose at Tetris*. https://www.jstor.org/stable/3619195?origin=crossref&seq=1#page_scan_tab_contents. Version: 1997. – Eingesehen am 31.1.2019

[18] pygame: *pygame*. https://www.pygame.org/news. – Eingesehen am 04.02.2019

[19] team keras: *Keras*. https://keras.io/. – Eingesehen am 04.02.2019

I. Anhang

I.I. Weitere Darstellungen

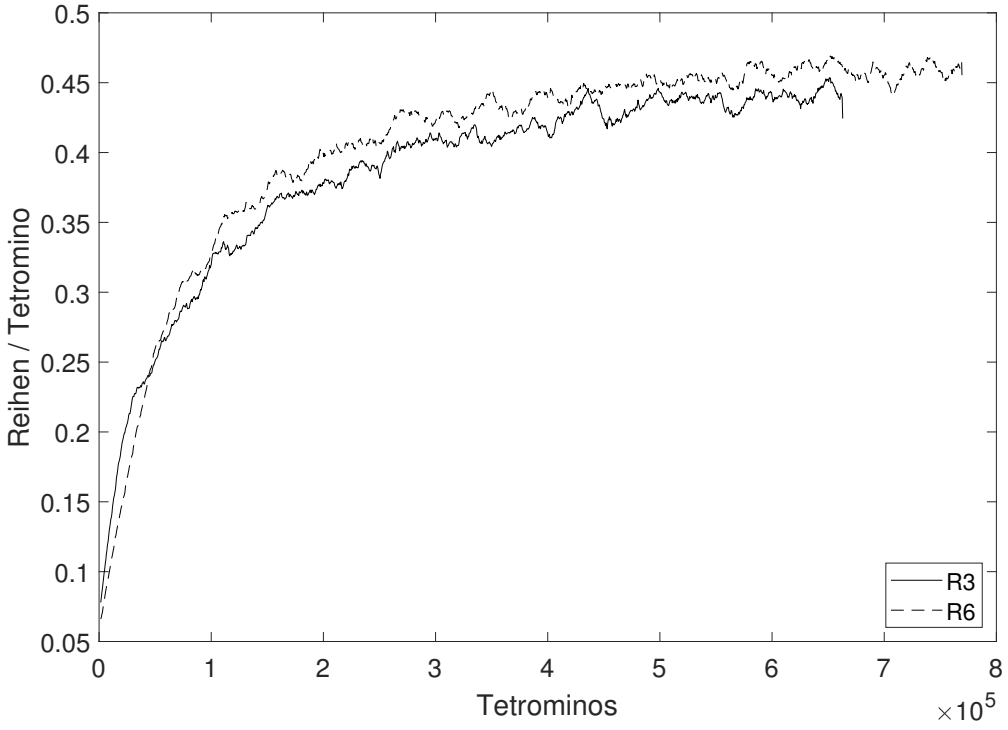

Abbildung 26: R3 und R6 $8 \cdot 10^5$

I.II. Bedienung und Installation

Neben der für der Masterveranstaltung *Angewandte KI* empfohlenen Pakete, wie Numpy, Keras und Tensorflow werden für die Ausführung des Programs lediglich Pygame für Python3 und heapq benötigt.

Es wird empfohlen, das Programm über das Terminal mit *python3 main.py* aufzurufen. Nun startet die Pygame GUI und das Tetris Spiel könnte wie gewohnt über die Pfeiltasten gesteuert werden. Im Hintergrund werden in diesem Fall die eingegebenen Aktionen gespeichert und mit Rewards versehen, so wie es bei dem Q-Learning Agenten geschehen würde.

Alternativ kann mittels Leertaste der Q-Learning Agent gestartet werden. Mittels Backspace kann der Zeichenmodus ein und ausgeschaltet werden. Bei ausgeschaltetem Zeichenmodus werden die Tetrominos schneller platziert und der Agent lernt schneller.

Eine gespeicherte Datei mit Gewichten des neuronalen Netzes kann mit L geladen werden. Die Exploration und das Learning wird in diesem Fall ausgestellt und es kann die vom Agenten erlernte Strategie beobachtet werden. Mit I lernt der Agent weiter und es wird zusätzlich eine bereitzustellende *gameData.csv*, welche gespeicherte Aktionen, Stati und Rewards enthält geladen. Diese kann durch Drücken von S zuvor gespeichert werden. Letzteres speichert auch die beim menschlichen Spiel aufgezeichneten Züge.

Zum Beobachten einer vergleichsweise guten Strategie sind die bei dieser Ausarbeitung angehängten Gewichte des vom in Abbildung 24 dargestellten Agenten zu laden. Hierfür ist es nötig das Programm zu starten und dann unmittelbar Leertaste, gefolgt von L zu drücken.