Computer spielen Computerspiele mit Hilfe von reinforcement learning am Beispiel Vier Gewinnt

Pablo Lubitz

BACHELORARBEIT

eingereicht am
Universitäts-Bachelorstudiengang

Informatik

in Bremen

im März 2020

Betreuung:

Holger Schultheis, Thomas Barkowsky

Erklärung

Ich erkläre eidesstattlich, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen entnommenen Stellen als solche gekennzeichnet habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Bremen, am 9. März 2020

Pablo Lubitz

Inhaltsverzeichnis

Kurzfassung

In dieser Bachelorarbeit wurde sich mit dem Thema Reinforcement-Learning in Spielen beschäftigt. Es wurde eine Implementierung dieses Verfahrens für das Spiel Vier Gewinnt genutzt. Mit der Implementation wurde getestet, ob dieses Verfahren durch das Beobachten des gegnerischen Verhaltens optimiert werden kann. Hierbei wurde eine minimale Verbesserung festgestellt.

Abstract

In this bachelor thesis I am exploring reinforcement learning in the area of games. Therefore, an implementation of this procedure was used for the game Connect Four. Furthermore, a test was made to proof that the implementation could be optimized with the addition of a way to learn what your enemie did. This was shown to have a minimal positive impact.

Kapitel 1

Einleitung

1.1 Motivation

Spiele machen Spaß und können genutzt werden, um neue Fähigkeiten zu lernen oder zu verbessern. Das Lernen beim Spielen funktioniert dann am besten, wenn das Spiel einen gewissen Schwierigkeitsgrad hat, welche den Spieler fordert aber nicht überfordert. Ist das Spiel zu einfach, wird der Spieler gelangweilt, ist es zu schwer, wird er frustriert. In beiden Fällen sucht er sich wahrscheinlich sehr bald eine andere Beschäftigung. Will man dies vermeiden, kann es helfen einen anderen Spieler zu haben, welcher ein ähnliches Vorwissen mitbringt. Wenn nun die Rolle des Gegners nicht von einem Menschen besetzt werden kann, ist es naheliegend einen Computergegner zu erschaffen. Solch ein Computergegner wird normalerweise mit Algorithmen umgesetzt, die einen optimalen Zug errechnen. Seit einigen Jahren ist es, durch den Fortschritt in Technologie und Wissenschaft, möglich Computergegner zu erschaffen, die ihr Können durch künstliche neuronale Netze antrainiert haben.

1.2 Problemstellung

Computergegner, die mit Hilfe von künstlichen neuronalen Netzen trainiert wurden, haben das Problem, dass sie nur gut werden können, wenn es auch einen guten Gegner gibt oder eine enorme Menge an Spielaufzeichnungen existiert, an denen sie trainieren können. Da dieses Training erst richtigen Effekt zeigt, wenn sehr viele Spiele gespielt wurden, ist es zeitlich nicht wirklich sinnvoll einen Menschen als Trainer zu nutzen. Hier wird normalerweise auf einen weiteren Computergegner zurückgegriffen, welcher das Spiel beherrscht ohne dies mit künstlichen neuronalen Netzen erlernt zu haben. Optimal wäre es, wenn die Anzahl der benötigten Spiele beim Training reduziert werden könnte, um schneller bessere Ergebnisse zu erzielen oder sogar nur mit einem Menschen lernen zu können.

1.3 Umsetzung

Meine Idee ist es, das Training durch das Mitlernen der Züge des Gegners zu verbessern. Dies erscheint sinnvoll, da der Gegner das Spiel schon beherrscht und daher tendenziell gute Züge machen wird.

Kapitel 2

Grundlagen

In dieser Arbeit werden einige Technologien genutzt, um die Problemstellung zu lösen. Die wichtigsten dieser Technologien werden hier erklärt, um das weitere Verständnis des Lesers zu gewährleisten.

2.1 Vier Gewinnt

Vier Gewinnt ist ein Spiel für zwei Spieler, in dem abwechselnd Spielsteine in ein vertikales Spielfeld der Größe $7X6$ (42 Felder) eingeworfen werden. Jeder Spieler kann sich in seinem Zug zwischen einem von sieben Einwurflöchern entscheiden. Wird ein Spielstein eingeworfen, so fällt dieser auf die niedrigste der sechs Positionen, die noch nicht durch andere Spielsteine gefüllt ist. Hierdurch ergibt sich ein Spielfeld mit 42 Feldern. Sind schon sechs Spielsteine in ein bestimmtes Einwurfloch gesteckt worden, so darf hier kein weiterer Spielstein eingeworfen werden. Ein Spieler gewinnt das Spiel, wenn er es geschafft hat, dass sich vier seiner Spielsteine in einer Reihe befinden. Eine Reihe kann horizontal, vertikal oder diagonal gebildet werden. Werden alle 42 Positionen mit Spielsteinen befüllt, ohne eine Reihe von vier Steinen des selben Spielers zu bilden, geht die Partie unentschieden aus.

Abbildung 2.1: Das Spiel Vier Gewinnt [26]

2.2 Minimax-Algorithmus

Ein Minimax-Algorithmus beschreibt einen rekursiven Ansatz zum Finden einer optimalen Lösung für Probleme von zwei Parteien mit widersetzlichen Zielen. Dies sind in der Regel Nullsummenspiele mit perfekter Information, es gibt also für jeden Gewinn des einen Spielers genauso viel Verlust für den anderen und es gibt kein Spielelement, das nur für einen Spieler einsehbar ist. Der Minimax-Algorithmus berechnet sich hierfür den Suchbaum des Spiels, welcher alle möglichen Züge beider Spieler in nachvollziehbarer Reihenfolge enthält. Hierbei werden in jedem Rekursionsschritt alle möglichen Züge des derzeitigen Spielers betrachtet und bewertet. Wenn der derzeitige Zug nicht zum Gewinn führt, findet eine Bewertung durch das Betrachten des nächsten Rekursionsschrittes statt. Der Name Minimax ergibt sich durch das Betrachten der abwechselnden besten Züge der Spieler. Da dies aus der Perspektive von einem Spieler betrachtet wird, ergibt sich hieraus ein abwechselnd minimaler und maximaler Zug. Dieses Verfahren führt bei einfachen Spielen wie Tic-Tac-Toe zu einem perfekten Spieler[6], da hier eine überschaubare Menge an möglichen Zügen betrachtet wird (9 Felder, die von 2 Spielern gefüllt werden). Für komplexere Probleme gibt es die Möglichkeit eine Suchtiefe zu bestimmen, wodurch die Rekursion nur bis zu dieser Tiefe durchgeführt wird. Dies ist sinnvoll da die Berechnung einzelner Züge sonst sehr lange dauern kann. Der Minimax-Algorithmus kann in bestimmten Spielen noch verbessert werden. Diese Verbesserungen sind der Negamax-Algorithmus und die Alpha-Beta-Suche, welche im Folgenden einmal beschrieben werden. Der Einfachheit halber wird trotz dieser Erweiterungen in dieser Arbeit im Bezug auf diesen Algorithmus immer von Minimax geredet.[6]

2.2.1 Negamax-Algorithmus

Der Negamax-Algorithmus basiert auf der Annahme, dass der maximal beste Zug für den einen Spieler der minimal beste Zug für den anderen bedeutet. Ist dies so in einem Spiel, in dem beide Spieler in jedem Zug die selben Dinge tun können, kann hierdurch die Rekursion vereinfacht werden, indem für beide Spieler die selbe Formel benutzt wird. Hierfür muss dann in jedem Rekursionsschritt der Input negiert werden. Wegen dieser Negierung heißt dieser Algorithmus auch Negamax.[5]

2.2.2 Alpha-Beta-Suche

Die Alpha-Beta Suche ist eine Variante des Minimax-Algorithmus, die die Berechnung des Suchbaumes beschleunigt, indem sie bestimmte Teile der Suche nicht ausführt, wenn es schon einen deutlich besseren Pfad gibt.[1]

2.2.3 Wahrscheinlichkeitsverteilung

Da der Minimax-Algorithmus im Normalfall immer deterministisch den besten Zug wählt, wurde er noch um eine Wahrscheinlichkeitsverteilung ergänzt. Die Wahrscheinlichkeitsverteilung basiert auf der Annahme, dass in jedem State (siehe 2.3.4) die Chancen der Menge aller möglichen Aktionen addiert 100 Prozent ergibt. Die Chancen der durch den Minimax-Algorithmus errechneten besseren Züge sind hierbei höher als die der schlechteren. Somit kann dann z.B. eine zufällige Aktion genommen werden, die aber sehr stark Züge mit besseren Chancen bevorzugt. Hierdurch wird der Minimax-Algorithmus probabilistisch und kann somit in seiner Stärke angepasst werden.[7]

2.3 Reinforcement-Learning

Das Reinforcement-Learning (Bestärkendes Lernen) ist eine Machine-Learning Methode, bei der ein Agent mit Hilfe eines Environments, einer Policy, States, Actions und eines Rewards lernt, eine ihm gegebene Aufgabe zu lösen. Hierfür bieten sich Probleme an, die als Markow-Entscheidungsproblem formuliert werden können, da diese alles abdecken, was für ein Environment wichtig ist.[12] Es wird dabei grundsätzlich wie in Abbildung 2.2 dargestellt vorgegangen. Der Agent bekommt also einen State vom Environment, und durch diesen sucht er sich mittels seiner Policy eine Action aus. Diese Action beeinflusst dann das Environment und dieses gibt dem Agenten einen Reward für seine Action. Durch den Reward kann der Agent dann die Gewichtungen für seine Policy anpassen. Dazu bekommt der Agent dann den neuen State und kann dann, mit der angepassten Policy, wieder eine neue Action auswählen. Diese Teile des Reinforcement-Learnings werden im Folgenden einmal genauer beschrieben.[8; 15]

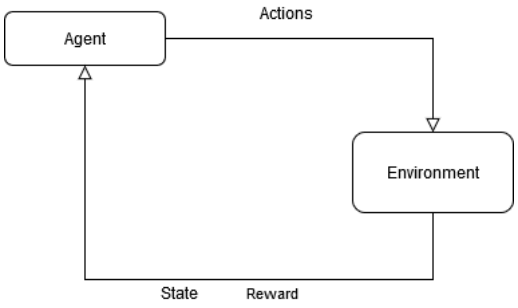


Abbildung 2.2: grundsätzliches Konzept von reinforcement learning

2.3.1 Environment

Das Environment beschreibt das gesamte Problem, das der Agent versucht zu lösen. Es beinhaltet alles, was für das Spielen wichtig ist außer dem Spieler, welcher von dem Agenten behandelt wird. Dies sind bei den meisten Spielen ein Spielfeld mit den Positionen aller Akteure, Spielsteine oder sonstigen Elemente, die zum Spielen genutzt werden.

Dazu kümmert sich das Environment auch noch um die Spiellogik. Die Spiellogik beschreibt alle Abläufe, die durch die Regeln des Spiels definiert wurden. In dem Beispiel Vier Gewinnt wäre das unter anderem die Funktion, dass die Spielsteine immer auf das untereste freie Feld fallen oder die Überprüfung, ob ein Spieler gewonnen hat. Das letzte wichtige Element im Environment der meisten Spiele ist der Gegenspieler, welcher normalerweise die selben Möglichkeiten besitzt wie der Spieler. Für das Verhalten des Gegenspielers werden im Normalfall Algorithmen genutzt, die den bestmöglichen Zug errechnen, wie hier der Minimax-Algorithmus. Für sehr komplexe Spiele, bei denen sich kein optimaler Zug berechnen lässt, wird oft auf menschliche Spieler zurückgegriffen. Da ein Computer aber viel schneller im Spielen ist und Reinforcement-Learning viele Durchläufe absolvieren muss um Wirkung zu zeigen, wird versucht dies zu vermeiden oder bestehende Datensätze von Spielen mit Menschen zu nutzen.

2.3.2 Agent

Bei dem Agenten handelt es sich um den Akteur, der die ihm gegebene Aufgabe meistern soll. Er sieht das Environment und wählt mittels der Policy die ihm am besten erscheinende Action aus, um den Reward zu maximieren.

2.3.3 Policy

Mit der Policy wird das Verhalten beschrieben, nach dem der Agent entscheidet, welche der möglichen Aktionen die aktuell beste ist, um den Reward zu maximieren.

2.3.4 State

Der State beschreibt den aktuellen Zustand des Environments, welcher in jedem Zug an den Agenten weitergereicht wird, damit dieser seine Auswahl treffen kann.

2.3.5 Actions

Actions sind die möglichen Aktionen, zwischen denen sich der Agent je nach State entscheiden muss. Diese sind alle möglichen Züge, die ein Spieler zu einem bestimmten Zeitpunkt ausführen kann.

2.3.6 Reward

Der Reward ist die Rückmeldung, die der Agent für seine Aktionen bekommt. Diese Rückmeldung kann positiv aber auch negativ sein und beschreibt, wie gut es ist eine bestimmte Aktion in einem bestimmten State auszuführen. Hieran passt der Agent dann während des Trainings seine Q-Values (siehe 2.3.8) entsprechend seiner Policy an. Der Reward wird also benutzt, um das Verhalten des Agenten zu beeinflussen. Möchte man ein bestimmtes Verhalten erzielen, wird hierfür ein positiver Reward vergeben. Möchte man ein Verhalten vermeiden, wird ein negativer Reward vergeben. Je besser

die Reward-Funktion gewählt wird, desto eher wird der Agent gewolltes Verhalten zeigen. Bei der Wahl einer Reward-Funktion wird zwischen diskreten und kontinuierlichen Funktionen unterschieden. Es ist aber auch eine Mischung dieser beiden möglich.
Bei einer diskreten Reward-Funktion werden nur feste, im Vorhinein definierte Rewards gegeben, die zu bestimmten Events verteilt werden. Solch ein Event kann zum Beispiel das Gewinnen und Verlieren am Ende eines Spiels sein oder das Betreten des Agenten in einen gewollten oder ungewollten Bereich.
Bei einer kontinuierlichen Reward-Funktion wird der Reward zu jedem Schritt durch verschiedene Faktoren aus dem Environment berechnet. Solche Faktoren können zum Beispiel die verstrichene Zeit, die Anzahl der Züge oder die aktuelle Position des Agenten sein. Eine gute kontinuierliche Reward-Funktion hilft dabei, den Agenten schneller dem gewünschten Verhalten anzunähern, indem der Reward zu den gewünschten Zielstates hin immer weiter angehoben wird.
Da diese beiden Arten von Reward-Funktionen das Verhalten des Agenten an unterschiedlichen Stellen beeinflussen, ist es oft sinnvoll eine Mischung dieser zu nutzen. Wird sich für eine Mischung entschieden ist es wichtig, dass die aus beiden Teilen kommenden Rewards in Relation zueinander stehen. Dies kann sonst dazu führen, dass die Auswirkung von dem Teil mit kleineren Rewards von dem Teil mit größeren Rewards überschattet wird.

2.3.7 Q-Funktion

Bei komplexeren Environments ist es schwer für jeden Zug vorauszusehen wie zielführend er ist. Um diesem Credit Assignment Problem(siehe 2.3.11) entgegenzuwirken wird eine Q-Funktion benutzt, die anhand der gelernten Zustände versucht die ungenauen Q-Values an ihre potenziellen Werte anzunähern.[15]

2.3.8 Q-Values

Die Q-Values beschreiben für jedes Paar von State und Action einen erwarteten Reward. Leider sind diese Q-Values für die meisten Probleme nicht bekannt und müssen daher mit einer Q-Funktion antrainiert werden.[15]

2.3.9 Deep Q-Network

Da es bei einem Spiel mit einer Komplexität wie Vier Gewinnt nicht möglich ist für alle Paare von State und Action einen erwarteten Reward zu bestimmen, wird ein künstliches neuronales Netz genutzt, welches versucht eine Q-Funktion für diese erwarteten Rewards zu bestimmen. Dieses Verfahren nennt sich Deep Q-Learning. Hierbei wird ein Deep Q-Network erstellt, welches während des Trainings eine Policy erstellt. Hierfür werden alle gespielten Spiele abgespeichert. Von diesen Spielen werden dann zufällige Züge betrachtet und somit die aktuelle Policy durch die hier bestehenden Q-Values angepasst.[13]

2.3.10 Markow-Entscheidungsproblem

Das Markow-Entscheidungsproblem beschreibt eine Menge von State-Action-Probability-Reward Tupeln. Es gibt also für jeden State, Actions, die eine Chance (Probability) haben in einen anderen State zu wechseln. Jede dieser Transitionen hat einen Reward, der dieser Transition einen Wert zuweist. Das Entscheidungsproblem bezieht sich hier auf die Transition von einem Start-State zu einem Ziel-State, wobei versucht wird den Reward zu maximieren.[12]

2.3.11 Credit Assignment Problem

Das Credit Assignment Problem beschreibt ein häufig auftretendes Problem in Environments, welches sich dadurch auszeichnet, dass ein Ablauf aus mehreren Schritten erst nach seinem Abschluss einen Reward zugewiesen bekommt. Es ist dann nicht klar zu beurteilen, welcher der Schritte wie hilfreich zur Erfüllung des Ablaufs war.[2]

2.3.12 Episode

Eine Episode beschreibt einen einzelnen Durchlauf des Agenten vom Start- bis zum Ziel-State.

2.3.13 Epoche

Eine Epoche ist eine Sammlung von Episoden und beschreibt einen gesammten Lerndurchlauf.

2.4 Software

Um die Problemstellung zu bearbeiten wurde eine Implementierung des Spiels in Python genutzt, welche als ein OpenAI Gym Environment fungiert. An diesem Environment trainiert dann ein Agent, der mit Hilfe von Keras erschaffen wurde.

2.4.1 Python

Als Programmiersprache wurde sich für Python[22] entschieden, da alle für diese Arbeit wichtigen Packages in dieser Programmiersprache existieren. Der Schöpfer von Python beschreibt sie selbst als einfach zu erlernende, objektorientierte Programmiersprache, welche Leistung mit einer klaren Syntax verbindet.[9]

2.4.2 OpenAI Gym

Gym [20] ist ein Werkzeug zum Entwickeln und Vergleichen von Reinforcement-Learning-Algorithmen, welches von OpenAI, einem Forschungslabor aus San Francisco, zur Verfügung gestellt wird. Es bietet einen Standard zwischen dem Environment und dem Agenten. Hierdurch können neue Agenten und Environments nach gewissen Vorgaben erstellt werden. Somit ist es mit Gym einfacher möglich, verschiedene Agenten an einem Environment zu trainieren und diese zu vergleichen oder den selben Agenten an verschiedenen Environments auf seine Anpassungsfähigkeit zu testen.

2.4.3 Tensorflow

Bei Tensorflow [23] handelt es sich um ein Open Source Framework für Maschine-Learning, welches ursprünglich für den internen Bedarf bei Google, beispielsweise fürl Spracherkennung oder Google Maps, entwickelt wurde.

2.4.4 Keras

Keras[19] ist eine Open Source Bibliothek, die auf Tensorflow aufbaut. Sie bietet die Möglichkeit vereinfacht künstliche neuronale Netze zu erstellen, um diese in Reinforcement-Learning-Algorithmen oder anderen Szenarien zu benutzen.

Kapitel 3

Stand der Technik

In den letzten Jahrzehnten wurde für viele Spiele, die bis dahin als zu komplex galten, Programme geschaffen, die diese spielen können. Das Spielniveau der hierdurch erschaffenen Spieler ist dabei gleich gut oder besser als das von den besten menschlichen Spielern. Dies liegt zu einem großen Teil an den Fortschritten in der Hardware, welche es ermöglichen, große künstliche neronale Netzwerke zu erstellen und zu trainieren. Einige der bekanntesten Erfolge sind zum Beispiel Deep Blue für Schach und Alpha Go für das Spiel Go.[15]
Weitere Erfolge aus den letzten Jahren sind Agenten, die Computerspiele meistern, wie zum Beispiel MarI/O [25], welcher Super Mario spielt oder AlphaStar [24], welcher Starcraft II spielen kann. Die Veröffentlichung von OpenAI Gym[20], welche unter anderem eine Menge an Atari-Spielen zur Verfügung stellt, eröffnete das Thema Reinforcement-Learning für Spiele für die gesamte Welt. Wie in dem Buch von Sutton und Barto [15] gezeigt wurde, lassen sich Markow-Entscheidungsprobleme mit Hilfe von reinforcement learning lösen. Dass das Lernen durch das Imitieren des Verhalten eines Profis verbessert werden kann, wurde unter anderm von Oliver Amantier gezeigt. [4; 10; 14]

3.1 Arbeiten mit ähnlichem Inhalt

Ich habe zwei Arbeiten gefunden, die auch mit Hilfe von Reinforcement-Learning versuchen, das Spiel Vier Gewinnt zu meistern. Einmal die Arbeit von Lukas Stephan [16], welcher das Problem mit einer Java Implementierung angegagen ist und den Lernalgorithmus mit einer Mustererkennung ausgestattet hat. Zweitens die Arbeit von Markus Thill, Patrick Koch und Wolfgang Konen [11], welche N-Tupel-Systeme nutzten, um einen Agenten zu trainieren. Anders als diese Arbeiten ist mein Ansatz nicht das Erkennen von Mustern, sondern das Lernen durch das Verhalten des Gegners.

Kapitel 4

Konzept

Im Folgenden wird nun beschrieben, was das inhaltliche Ziel dieser Bachelorarbeit ist und wie vorgegangen werden soll um dieses Ziel zu erreichen.

4.1 Zielsetzung

Es soll am Beispiel des Spiels Vier Gewinnt gezeigt werden wie ein Agent funktioniert, welcher die Spielstrategie mit Hilfe von Reinforcement-Learning erlernt hat und ob diese Technik durch das Imitieren des Gegners verbessert werden kann.

4.2 Methode

Es wird eine digitale Version des Spiels Vier Gewinnt genutzt und eine Schnittstelle erschaffen die es ermöglicht auszuwählen, ob ein Mensch oder der Computer die Rolle der Spieler übernimmt. Dies ist wichtig, da ein selbstlernender Algorithmus viele Spiele durchlaufen muss, um einen Lernfortschritt aufzuzeigen. Somit kann in der Simulation trainiert werden, ohne jedes einzelne Spiel gegen einen Menschen spielen zu müssen. Spielt der Computer, soll zwischen verschiedenen Strategien ausgewählt werden können, welche unterschiedlich effektiv sind. Die Strategien sind: komplett zufällige Züge wählen, immer so weit wie möglich links ins Spielfeld werfen und ein Minimax-Algorithmus, welcher verschieden starke Varianten unterstützt. Es wurde sich für diese Strategien entschieden, da bei den einfachen Strategien das Lernverhalten besser zu erkennen ist. Für den Minimax-Algorithmus wurde sich entschieden um dann das Lernverhalten gegen einen sehr guten Gegner zu zeigen. An diesen Strategien soll der Agent mit verschiedenen Voraussetzungen trainieren, um dann vergleichen zu können, was besser und schlechter beim Lernen hilft.

4.3 Evaluierung

Je nachdem wie gut der selbstlernende Agent nach dem Trainieren gegen die unterschiedlich effektiven Strategien ist, zeigt sich dann, wie gut das Reinforcement-Learning mit den verschiedenen Voraussetzungen funktioniert hat. Somit kann dann gezeigt werden welche Veränderungen besser sind, um einen guten Reinforcement-Learning Agenten zu erschaffen. Es sollte sich dann auch zeigen ob das Lernen des Verhalten des Gegners zu einer Verbesserung des Lernverhaltens des Reinforcement-Learning-Agenten führt.

Kapitel 5

Implementierung

Es wurde sich für eine bestehende grundlegende Implementierung entschieden, welche schon die Spiellogik von Vier Gewinnt sowie einen sehr einfachen Reinforcement-Learning Ansatz enthält. In dieser Version lernt der Agent gegen einen Gegner zu gewinnen, der zufällige Züge macht.[18]
Diese Implentierung wurde um einen Gegner erweitert, der seine Züge nach dem Minimax-Algorithmus auswählt. Weitergehend wurde die Reinforcement-Learning Implementierung überarbeitet, um gegen den Minimax-Algorithmus gewinnen zu können. Hierfür wurde es ermöglicht, dass der Agent von den Zügen seines Gegners lernen kann. Diese Implementierungen werden in diesem Kapitel einmal genauer beschrieben:

5.1 Vier Gewinnt

Die Implementierung des Spiels Vier Gewinnt deckt alle für das Spiel wichtigen Regeln ab. Es wird sich darum gekümmert, welcher Spieler aktuell dran ist. Hat dieser sich dann für eine Action entschieden, übernimmt das Spiel die korrekte Platzierung der Spielsteine sowie die Überprüfung auf den Abschluss eines Spiels.
Es wurde sich dafür entschieden, dass der Reinforcement-Learning-Agent immer mit einer Partie beginnt, um somit die Menge an möglichen States zu halbieren, wodurch das gesamte Problem vereinfacht wird.

5.2 Gegner mit Minimax-Algorithmus

Der Minimax-Algorithmus wurde als Teil des Environments implementiert und besteht im Groben aus zwei Teilen. Einmal aus der Funktion *check_next_actions*, die den Suchbaum für die nächsten Züge aufbaut. Und aus der Funktion $find\_best\_move$, die den Suchbaum nach dem besten Zug durchsucht. Hierdurch kann sich dieser Gegner zu jedem State den besten nächsten Zug errechnen. Da dies im Optimalfall ein unschlagbarer Gegner ist wird hier noch mit eine Funktion zur Wahrscheinlichkeitsverteilung implementiert, die dafür sorgt, dass der Minimax-Algorithmus in bestimmten Situationen nicht optimal spielt.

5.2.1 check_next_actions

Das Erstellen des Suchbaums funktioniert nach dem Negamax-Algorithmus, es wird also eine Funktion für beide Spieler genutzt und immer abwechselnd das Maximum und das Minimum gesucht.

5.2.2 find_best_move

Das Suchen des besten Zuges wird durch die Alpha-Beta-Suche beschleunigt, indem bestimmte Äste des Suchbaumes, die sicher nicht zum gewünschten besten Zug führen, nicht weiter betrachtet werden.

5.2.3 Wahrscheinlichkeitsverteilung

Durch die Wahrscheinlichkeitsverteilung wird es ermöglicht einen zufälligen Zug aus den vom Minimax-Algorithmus errechneten Zügen zu wählen. Die zuvor errechnete Gewinnchance eines Zuges bestimmt, wie wahrscheinlich es ist, dass dieser ausgewählt wird. Diese Wahrscheinlichkeit kann mit einer $Variable$ angepasst werden, wodurch die Wahrscheinlichkeit steigt, dass einer der suboptimalen Züge, die vom Minimax-Algorithmus berechnet wurden, ausgewählt wird.

5.3 Gegner mit schlechten Strategien

Um das Lernverhalten zu beobachten, wurden zwei Gegner mit schlechten Strategien geschaffen.

5.3.1 Gegner mit deterministischen Aktionen

Es wurde ein Gegner geschaffen, der bei jedem Zug immer so weit links wie möglich in das Spielfeld einwirft.

5.3.2 Gegner mit zufälligen Aktionen

Es wurde ein Gegner geschaffen, der bei jedem Zug in ein zufälliges Loch des Spielfeldes einwirft, das noch frei ist.

5.4 Reinforcement-Learning

Für das Reinforcement-Learning wird ein Deep Q-Network genutzt, welches einen Agenten an dem Environment trainiert. Im Folgenden wird die Implementierung der hierfür notwendigen Komponenten beschrieben.

5.4.1 Environment

In Vier Gewinnt beschreibt das Environment das Spielfeld und den Gegner. Das Spielfeld ist hier ein zweidimensionales Array der Länge $7X6$. Der Gegner ist während des Lernens ein Computergegner, der nach unterschiedlichen Strategien, wie zum Beispiel

dem Minimax-Algorithmus handelt.

5.4.2 Agent

Der Agent ist einer der beiden Spieler von Vier Gewinnt, welcher lernt, seine Züge durch die ihm gegebene Policy auszuwählen, um einen möglichst hohen Reward zu erhalten.

5.4.3 State

Für Vier Gewinnt beschreibt der State, welche der 42 Felder mit welchen Steinen befüllt sind. Ein Beispiel für so einen State kann man in dem Abschnitt Visualsierung des Spiels 5.7 finden.

5.4.4 Actions

Die Actions in Vier Gewinnt beschreiben die sieben Löcher, in die der Agent Steine werfen kann.

5.4.5 Reward

Zuerst wurde eine einfache diskrete Implementierung erstellt, welche den Reward immer zum Ende einer Partie ausgibt, wobei dieser beim Gewinn 1, beim Verlieren -1 und wenn die Partie unentschieden ausgeht 0 beträgt. Dies wurde durch die Anzahl der Züge um eine kontinuierliche Komponente erweitert. Hierfür wird der Reward, der durch die diskrete Implementierung ermittelt wurde durch die Anzahl der Züge geteilt. Da es nicht möglich ist vor dem vierten Zug zu gewinnen, wird erst ab diesem Zug gezählt. Es ergibt sich also die Formel:

$$\frac{diskreter\,Reward}{anzahl\,Z\ddot{u}ge - 3} = kontinuierlicher\,Reward$$

Wie man in Abbildung 5.1 gut sehen kann wird somit das schnelle Gewinnen mehr belohnt (obere Linie) und das schnelle Verlieren mehr bestraft (untere Linie). Da nur unentschieden gespielt wird wenn das Spielfeld komplett voll ist, wird hier ein Reward von 0 gegeben (einzelner Punkt). Diese Reward-Funktion wird im weiteren Text zugabhängige Reward-Funktion genannt.

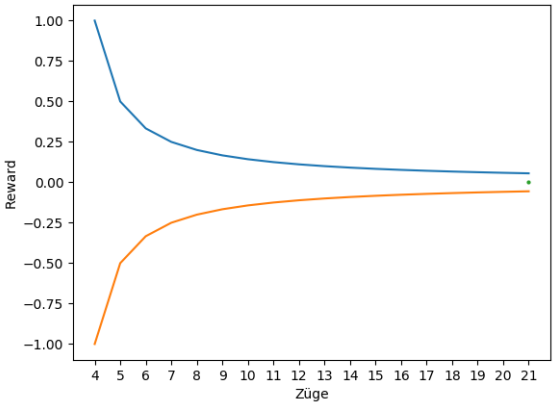


Abbildung 5.1: Reward Funktion

5.4.6 Policy

Da in Vier Gewinnt nur am Ende eines Spiels ein Reward ausgegeben werden kann, wird hier ein Deep Q-Network genutzt um das Credit Assignment Problem zu lösen. Dieses Deep Q-Network wird nach einer Epsilon-Greedy Policy trainiert.

Deep Q-Network

Das Deep Q-Network speichert zu jedem gespielten Zug den derzeitigen State, die ausgeführte Action, den erhaltenen Reward, der daraus folgende State und ob der neue State die Episode beendet hat in sein Memory. Aus diesem Memory wird dann, mit der Funktion *experience_replay*, eine gewisse Menge an zufälligen Proben genommen. Es ist sehr sinnvoll hier eine zufällige Probe zu nehmen, da sonst durch die in Reihe betrachteten Züge ein ungewollter Bias entstehen kann, welcher die Konvergenz des Trainingsalgorithmus stark verlangsamen kann [8, Seite 469]. Für diese Proben werden die Q-Values vorhergesagt. Mit diesen Q-Values wird dann das Q-Network trainiert. Dieser Prozess wird so häufig wiederholt, dass hierdurch die Q-Values immer genauer werden und dadurch ergibt sich dann eine zielführende Policy.

Epsilon-Greedy

Als Unterstützung des Deep Q-Networks wurde sich für die Epsilon-Greedy Policy entschieden, da diese sehr gut für Deep Q-Networks funktioniert [3]. Epsilon-Greedy beschreibt das Verhalten des Agenten, welches entweder explorativ oder ausbeutend ist. Ist das Verhalten explorativ, entscheidet sich der Agent für einen Zug, den er vorher noch nicht oder selten gemacht hat, um diesen zu erlernen. Ist das Verhalten ausbeutend, so wird der Zug genommen, welcher derzeitig die höchste Gewinnchance bietet. Zum Anfang des Lernens macht es Sinn, dass der Agent viel erkundet, da er noch kein Wissen über das Spiel besitzt. Hat der Agent dann einige Spiele gespielt, kann er auf die ausbeutende Strategie umschalten. Um dieses Verhalten des anfänglichen Erkundens und späteren Ausbeutens zu ermöglichen, nutzt die Epsilon-Greedy Policy die Formel:

$np.random.rand()) < 1 - \epsilon$. Gibt diese Formel $True$ zurück, so wird erkundet, gibt sie $False$ aus, wird ausgebeutet. Der Wert von Epsilon ist am Anfang etwas mehr als Null und wird während der Lernphase langsam erhöht, bis er etwas weniger als Eins erreicht hat. Somit ist das Ergebnis der Formel am Anfang meistens $True$ und später meistens $False$. Dadurch kann das Deep Q-Network am Anfang viele Daten bekommen um hieraus eine Policy zu erarbeiten und diese später nach und nach verbessern, wenn dann durch die Ausbeutung nach dieser Policy vorgegangen wird.

5.5 Lernen vom Gegner

Um das Lernen des Agenten zu verbessern wurde entschieden, dass der Agent auch die Züge seines Gegners zum Lernen nutzen kann. Dies ist für Vier Gewinnt angemessen, da es sich um ein Spiel mit perfekter Information handelt. Es sind also jedem Spieler zum Zeitpunkt einer Entscheidung alle Informationen über das Spiel ersichtlich. Da ein menschlicher Anfänger in diesem Spiel bei einer Niederlage an den Spielzügen seines Gegners lernen kann, wurde dies auch als sinnvoll für den Agenten angesehen. Hierfür mussten einige Anpassungen vorgenommen werden. Da der Reinforcement-Learning-Agent immer der erste der beiden Spieler ist, der seinen Zug macht, fehlen dem Deep Q-Network Informationen über den Zug nach seinem Gegner. Um diese Informationen zu erhalten, wird das gesamte Spiel einen halben Zug in der Vergangenheit betrachtet. Somit kann der alte Zug des Gegners als der eigene betrachtet werden und der eigene Zug als Antwort auf diesen. Damit der Reinforcement-Learning-Agent dies verarbeiten kann, muss noch das ganze Spielfeld negiert werden. Es werden also die Steine beider Spieler getauscht, damit es wieder wie eine ganz normale Zugabfolge aussieht.

5.6 Graphische Darstellung

Um die Effektivität der verschiedenen Versionen zu vergleichen, wird der Reward in einem Graphen geplottet. Dies wird mit dem Paket pyplot von matplotlib [21] gemacht. Beispiele hierfür findet man im Kapitel 6, Evaluierung.

5.7 Visualsierung des Spiels

Da das Spielfeld im Code nur als zweidimensionales Array, welches mit 1, 0 und -1 gefüllt ist, existiert, ist es für den Menschen nicht so einfach zu erkennen, was gerade im Spiel passiert ist. Hierfür wurde mit Hilfe des Python Pakets colorama [17] eine Visualisierung erschaffen. Diese wandelt alle Felder in den Unicode Character 'BLACK CIRCLE' (U+25CF) um, wobei die 1 gelb und die -1 rot eingefärbt werden. Es wird noch der Hintergrund blau eingefärbt, um dem Design des Spiels möglichst nahe zu kommen.
Abbildung 5.3 ist ein durch diese Weise erstelltes Spielfeld für das Interne Array 5.2

$$\begin{bmatrix} [& 0 & 0 & 0 & 0 & 0 & 0 & 0 &] \\ [& 0 & 0 & 0 & 0 & 0 & 0 & 0 &] \\ [& 0 & 0 & 0 & 0 & 0 & -1 & 0 &] \\ [& 0 & 0 & 0 & 0 & 0 & -1 & 0 &] \\ [& 1 & 0 & 1 & -1 & 0 & -1 & 0 &] \\ [& 1 & 1 & -1 & 1 & 0 & -1 & 1 &] \end{bmatrix}$$

Abbildung 5.2: Array-Representation

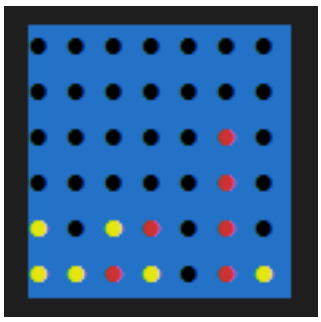


Abbildung 5.3: Das Spielfeld auf der Konsole

Kapitel 6

Evaluierung

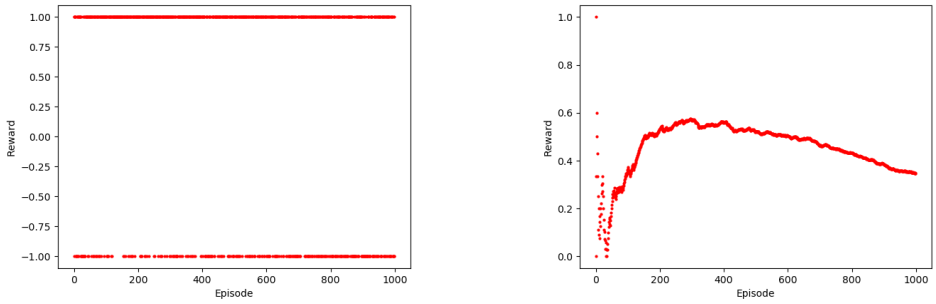
In diesem Kapitel wird der implementierte Reinforcement-Learning-Agent evaluiert. Hierfür wird zuerst, mit Hilfe von einem deterministischen Gegner und einem Gegner, der in jedem Zug eine zufällige Aktion auswählt, gezeigt, ob die neu gewählte Reward-Funktion einen Vorteil beim Lernen darstellt. Danach wird an diesen beiden Gegnern gezeigt, ob das Lernen mit Einbezug der gegnerischen Züge eine Veränderung im Lernverhalten aufweist. Dann wird gegen den Minimax Algorithmus trainiert, um zu zeigen, wie gut der Reinforcement-Learning-Agent werden kann.
Es folgen nun die Graphen dieser Evaluierung, wobei immer die zwei nebeneinander stehenden Graphen die selbe Lernepoche beschreiben. Der erste Graph (a,c,e oder g) beschreibt den absoluten Reward, also den genauen Wert zu jeder Episode. Der zweite Graph (b,d,f oder h) beschreibt den durchschnittlichen Reward zu jeder Episode im Bezug auf alle vorherigen Episoden dieser Epoche. Es wurde sich für eine Epoche von 1000 Episoden entschieden, da sich hier schon das Verhalten des Agenten abzeichnet und das Berechnen von längeren Epochen einen sehr hohen Zeitaufwand bedeuten würde.

Es sei darauf hingewiesen, dass der durchschnittliche Reward (b,d,f oder h) meist nicht das komplette Spektrum von 1 bis -1 abdeckt. Dies wurde gemacht um eine bessere Erkennbarkeit des Ausschnittes zu gewähren.

6.1 Reward-Funktion

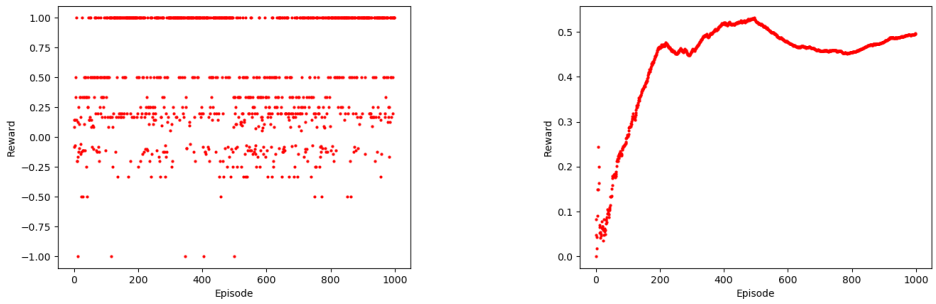
Als erstes wird die Veränderung durch die zugabhängige Reward-Funktion evaluiert. Hierfür wird Abbildung 6.1 mit Abbildung 6.2 sowie Abbildung 6.3 mit Abbildung 6.4 verglichen.
Abbildung 6.1 (b) und Abbildung 6.2 (b) verhalten sich fundamental sehr ähnlich. Beide erreichen einen durchschnittlichen Reward von 0.5 nach etwa 250 Episoden wobei die einfache Reward-Funktion aber einen abfallenden, und die zugabhängige Reward-Funktion einen konstanteren Trend aufweist.

Abbildung 6.3(b) zeigt einen starken Abfall in gewonnenen Spielen nach etwa der 300. Episode. Dieses Verhalten ist ebenfalls sehr stark in Abbildung 6.3(a) zu erkennen. Im Vergleich zu der einfachen Reward-Funktion wird in Abbildung 6.4(b) eine Annähe-



(a) absoluter Reward **(b)** durchschnittlicher Reward

Abbildung 6.1: Reinforcement-Learning an zufällig ziehendem Gegner mit einfacher Reward-Funktion und ohne die Züge des Gegners mitzulernen.



(a) absoluter Reward **(b)** durchschnittlicher Reward

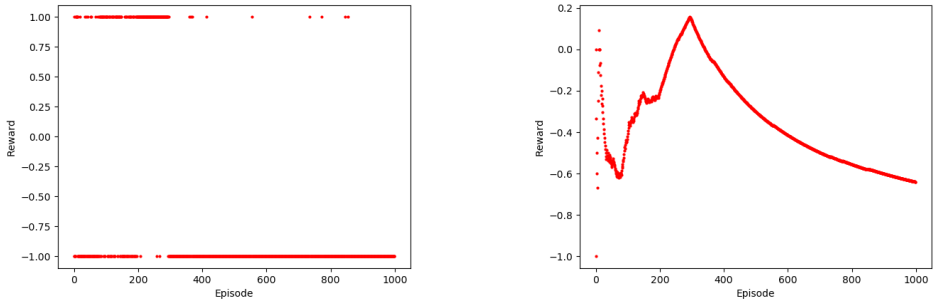
Abbildung 6.2: Reinforcement-Learning an zufällig ziehendem Gegner mit verbesserter Reward-Funktion und ohne die Züge des Gegners mitzulernen.

rung an 0 aufzeigt, was auf eine 50% Gewinnchance hinweist.

Es wurde gezeigt, dass durch die zugabhängige Reward-Funktion besser gegen den zufälligen und den deterministischen Gegner gespielt wurde. Hieraus schließe ich, dass die zugabhängige Reward-Funktion einen positiven Effekt auf das Lernverhalten des Reinforcement-Learning-Agenten hat. Aus diesem Grund wird in allen fortlaufenden Evaluierungen mit der verbesserten Reward-Funktion gearbeitet.

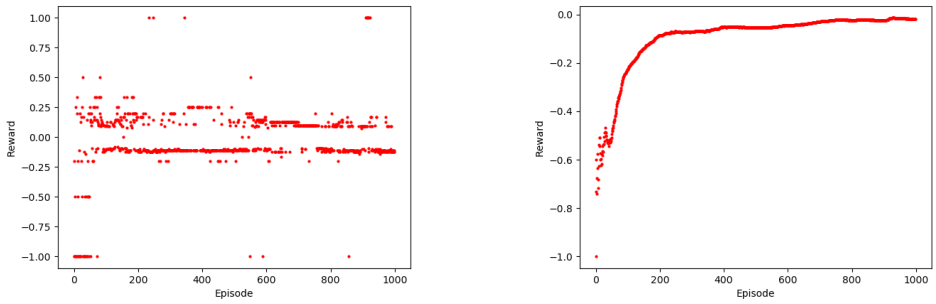
6.2 Lernen vom Gegner

Nun wird evaluiert, ob das Lernen von den Zügen des Gegners einen positiven Einfluss auf das Lernverhalten des Agenten hat. Hierfür werden wieder der deterministisch und der zufällig ziehende Gegner genutzt.
Wie in Abbildung 6.5 (b) zu sehen ist, ist das Ergebnis hier im Vergleich zu Abbildung



(a) absoluter Reward **(b)** durchschnittlicher Reward

Abbildung 6.3: Reinforcement-Learning an deterministisch ziehendem Gegner mit einfacher Reward-Funktion und ohne die Züge des Gegners mitzulernen.

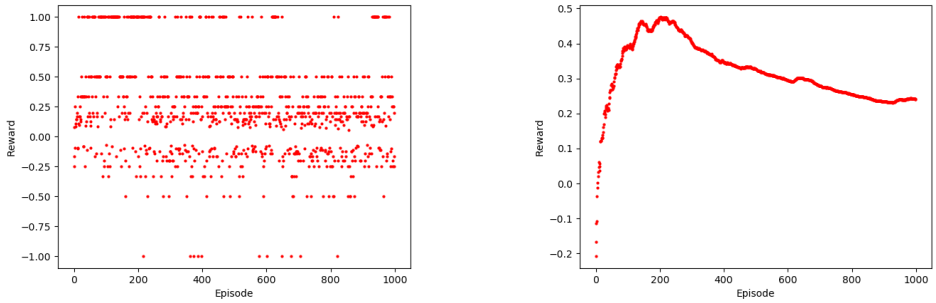


(a) absoluter Reward **(b)** durchschnittlicher Reward

Abbildung 6.4: Reinforcement-Learning an deterministisch ziehendem Gegner mit verbesserter Reward-Funktion und ohne die Züge des Gegners mitzulernen.

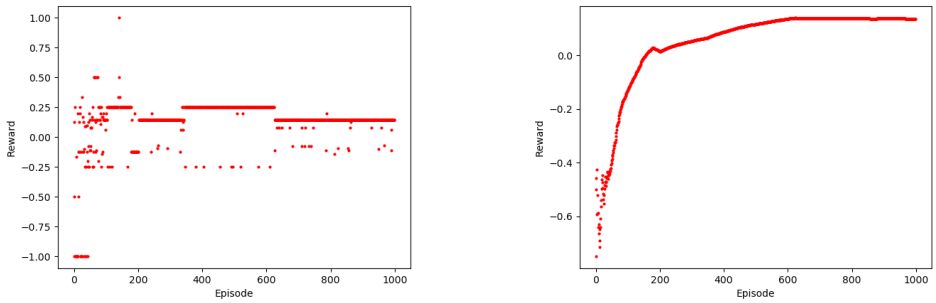
6.2 (b) wesentlich schlechter. Dies ergibt aber Sinn, da der Gegner komplett zufällig spielt und somit das Lernen seiner Züge eher als Nachteil betrachtet werden muss. Abbildung 6.6 zeigt hingegen eine sehr starke Verbesserung im Vergleich zu Abbildung 6.4. Wie in Abbildung 6.6(a) zu sehen ist, sind die meisten Rewards oberhalb von 0, dies war in Abbildung 6.4(a) nicht der Fall.

Somit wurde gezeigt, dass das Nutzen der Züge eines Gegners mit zielführender Strategie zu einer Verbesserung des Lernverhaltens des Reinforcement-Learning-Agenten führt. Da in den folgenden Tests nur noch gegen Gegner mit einer zielführenden Strategie gelernt wird, wird das Lernen von gegnerischen Zügen für alle folgenden Evaluationen genutzt.



(a) absoluter Reward **(b)** durchschnittlicher Reward

Abbildung 6.5: Reinforcement-Learning an zufällig ziehendem Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernent werden.

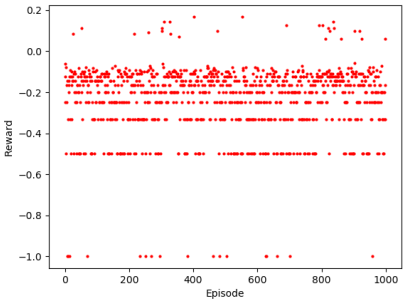


(a) absoluter Reward **(b)** durchschnittlicher Reward

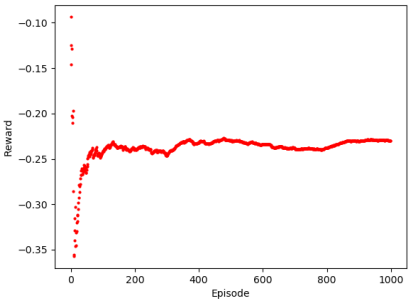
Abbildung 6.6: Reinforcement-Learning an zufällig ziehendem Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernent werden.

6.3 Training am Gegner mit Minimax-Algorithmus

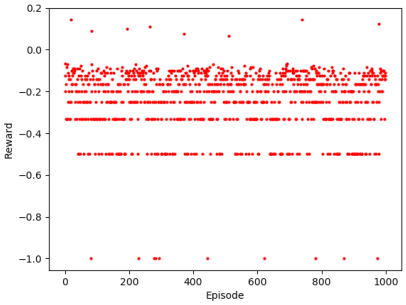
Die letzte Evaluierung sollte zeigen, dass ein Reinforcement-Learning-Agent mit den zuvor gezeigten Verbesserungen unterschiedlich schnell zu einer guten Strategie finden kann. Hierfür wurde der Reinforcement-Learning-Agent gegen den Minimax-Algorithmus mit verschieden starker Wahrscheinlichkeitsverteilung trainiert.
Die in Abbildung 6.7 und Abbildung 6.8 gezeigten Graphen zeigen das Trainingsverhalten gegen den Minimax-Algorithmus. Diese sind nach steigender Wahrscheinlichkeit den besten Zug zu wählen geordnet. Das erste Paar in Abbildung 6.7 (a und b) hat somit die niedrigste Wahrscheinlichkeit den besten Zug zu wählen und das letzte Paar in Abbildung 6.8 (c und d) die höchste.
Leider ließ sich hier kein Lernfortschritt feststellen, nicht mal bei der niedrigsten Wahrscheinlichkeit.



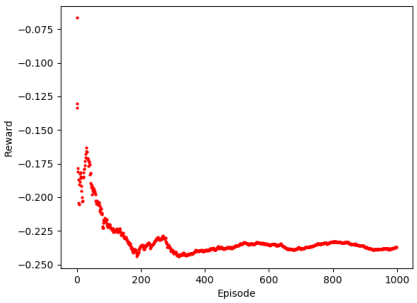
(a) absoluter Reward



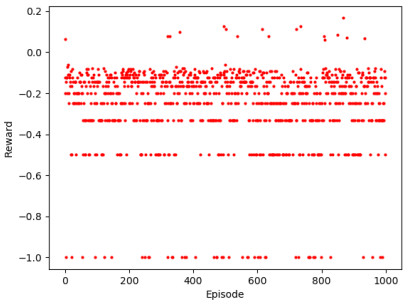
(b) durchschnittlicher Reward



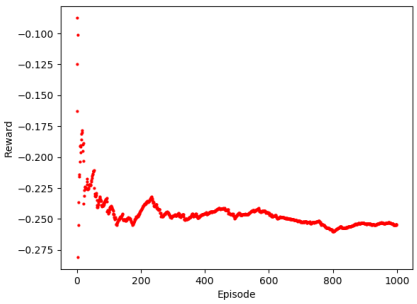
(c) absoluter Reward



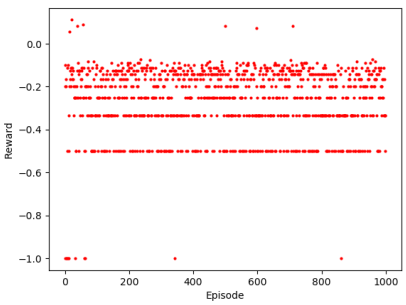
(d) durchschnittlicher Reward



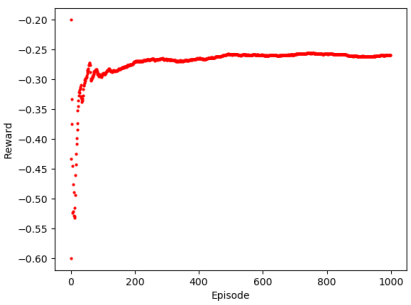
(e) absoluter Reward



(f) durchschnittlicher Reward

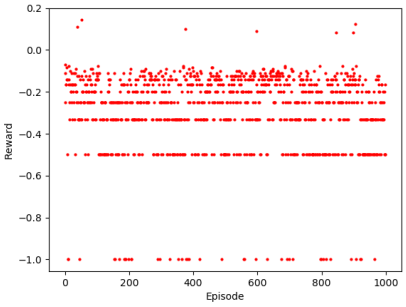


(g) absoluter Reward

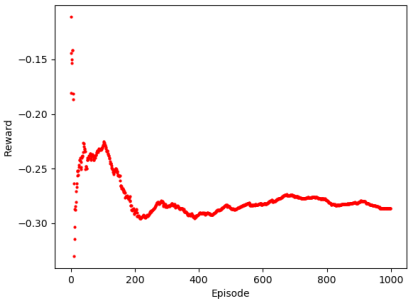


(h) durchschnittlicher Reward

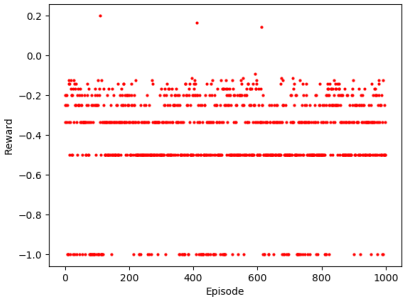
Abbildung 6.7: Reinforcement-Learning an Minimax Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernt werden. Teil 1



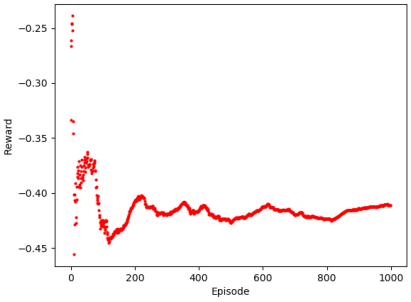
(a) absoluter Reward



(b) durchschnittlicher Reward



(c) absoluter Reward



(d) durchschnittlicher Reward

Abbildung 6.8: Reinforcement-Learning an Minimax Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernent werden. Teil 2

6.4 Besonderheiten

In den Graphen der Evaluation existiert eine Auffälligkeit, die bei ungefähr 250 bis 300 Episoden auftritt. Ist diese Anzahl an Episoden überschritten wird das Lernverhalten des Agenten oft schlagartig schlechter. Dies ist am deutlichsten in der Abbildung 6.3(b) zu sehen. Ich vermute, dass dies mit der Epsilon-Greedy Policy zusammenhängt, die in dem Raum etwa den Maximalen Wert für ϵ erreicht hat.

Kapitel 7

Fazit

Mit dieser Bachelorarbeit sollte geprüft werden, ob das Lernen der Züge des Gegners eine positive Auswirkung auf das Lernverhalten des Reinforcement-Learning-Agenten hat. Hierzu wurde die für diese Arbeit entwickelte Implementierung des Spiels Vier Gewinnt genutzt. In der Evaluierung konnte beim Vergleich der verschiedenen Versionen (mit und ohne Lernen der gegnerischen Züge) an Gegnern mit einfachen Strategien ein positiver Effekt festgestellt werden. Des weiteren wurde gezeigt, dass die gewählte Reward-Funktion angemessen war und somit zur Verbesserung des Lernverhaltens beigetragen hat. Diese Verbesserungen konnten vorerst leider nur am Beispiel von Gegnern mit schlechten und sehr einfach zielführenden Strategien, jedoch nicht am Gegner mit sehr guter Strategie gezeigt werden. Die Vermutung liegt nahe, dass dieses Verhalten auch bei der sehr guten Strategie eintritt, wenn eine längere Lernepoche absolviert wird. Da das Training des Agenten, welches in dieser Arbeit gemacht wurde, trotz der vorgenommenen Optimierungen schon sehr viel Zeit in Anspruch genommen hat, wurde davon abgesehen das Verhalten in längeren Lernepochen zu beobachten. Es spricht aber tendeziell nichts dagegen, dass sich ein Lernerfolg gegen den Minimax-Algorithmus zeigen kann.

In weiteren Untersuchungen könnte versucht werden, ob sich das hier gezeigte positive Verhalten auch beim Trainieren mit besseren Gegnern zeigen lassen kann. Hierfür müsste dann gegebenenfalls die Problemstellung vereinfacht oder der Algorithmus verbessert werden, sodass längere Lernepochen in überschaubarer Zeit absolviert werden können.

Quellenverzeichnis

Literatur

[1] Ashraf M Abdelbar. „Alpha-Beta Pruning and Althöfer's Pathology-Free Negamax Algorithm". *Algorithms* 5.4 (2012), S. 521–528 (siehe S. 4).

[2] Adrian K Agogino und Kagan Tumer. „Unifying temporal and structural credit assignment problems". In: *Proceedings of the Third International Joint Conference on Autonomous Agents and Multiagent Systems-Volume 2*. IEEE Computer Society. 2004, S. 980–987 (siehe S. 8).

[3] Meire Fortunato et al. „Noisy Networks for Exploration". Machine Learning (cs.LG). Ithaca, New York: Cornell University, 2017. URL: https://arxiv.org/pdf/1706.10295.pdf?source=post_page---------- (siehe S. 16).

[4] Olivier Armantier. „Does observation influence learning?" *Games and Economic Behavior* 46.2 (2004), S. 221–239. URL: http://www.sciencedirect.com/science/article/pii/S0899825603001246 (siehe S. 10).

[5] Hendrik Baier. „Der alpha-beta-algorithmus und erweiterungen bei vier gewinnt". *Bachelor Arbeit* (2006) (siehe S. 4).

[6] Plamenka Borovska und Milena Lazarova. „Efficiency of parallel minimax algorithm for game tree search". In: *Proceedings of the 2007 international conference on Computer systems and technologies*. 2007, S. 1–6 (siehe S. 4).

[7] Adele Diederich und Jerome R Busemeyer. „Simple matrix methods for analyzing diffusion models of choice probability, choice response time, and simple response time". *Journal of Mathematical Psychology* 47.3 (2003), S. 304–322 (siehe S. 5).

[8] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn and Tensorflow*. 2. Aufl. Gravenstein Highway North, Sebastopol, CA95472: O'Reilly Media Inc., 2017 (siehe S. 5, 16).

[9] Fred L. Drake Guido Van Rossum. *Python 3 Reference Manual*. 1. Aufl. Scotts Valley: CreateSpace, 2009 (siehe S. 8).

[10] Jonathan Ho und Stefano Ermon. „Generative Adversarial Imitation Learning". In: *Advances in Neural Information Processing Systems 29*. Hrsg. von D. D. Lee u. a. Curran Associates, Inc., 2016, S. 4565–4573. URL: http://papers.nips.cc/paper/6391-generative-adversarial-imitation-learning.pdf (siehe S. 10).

[11] Patrick Koch und Wolfgang Konen Markus Thill. „Reinforcement Learning with N-tuples on theGame Connect-4“. Part of the Lecture Notes in Computer Science book series (LNCS, volume 7491). Gummersbach, Germany: Department of Computer Science, Cologne University of Applied Sciences, 2012. URL: https://www.researchgate.net/profile/Wolfgang_Konen/publication/235219697_Reinforcement_Learning_with_N-tuples_on_the_Game_Connect-4/links/0912f510811c0942f6000000.pdf (siehe S. 10).

[12] Marco Wiering Martijn van Otterlo. „Reinforcement Learning and Markov Decision Processes“. Reinforcement Learning. Adaptation, Learning, and Optimization, vol 12. Berlin, Heidelberg: Springer, 2012. URL: https://www.ai.rug.nl/~mwiering/Intro_RLBOOK.pdf (siehe S. 5, 8).

[13] Volodymyr Mnih u. a. „Playing atari with deep reinforcement learning“. *arXiv preprint arXiv:1312.5602* (2013) (siehe S. 7).

[14] Bob Price und Craig Boutilier. „Implicit imitation in multiagent reinforcement learning“. In: *ICML*. Citeseer. 1999, S. 325–334 (siehe S. 10).

[15] Andrew G. Barto Richard S. Sutton. *Reinforcement Learning An Introduction*. 2. Aufl. Cambridge, Massachusetts: The MIT Press, 2018 (siehe S. 5, 7, 10).

[16] Lukas Stephan. „Implementierung und Evaluation eines intelligenten Spielagenten mit Reinforcement Learning“. Bachelorarbeit. Bremen, Deutschland: Universität Bremen, Informatik, Juni 2018. URL: unpublished (siehe S. 10).

Software

[17] *colorama*. URL: https://pypi.org/project/colorama/ (siehe S. 17).

[18] *connect-four-tensorflow by alexisjanvier*. URL: https://github.com/marmelab/connect-four-tensorflow (siehe S. 13).

[19] *Keras*. URL: https://keras.io/ (siehe S. 9).

[20] *OpenAI Gym*. URL: https://gym.openai.com/ (siehe S. 9, 10).

[21] *pyplot*. URL: https://matplotlib.org/api/pyplot_api.html (siehe S. 17).

[22] *Python*. URL: https://www.python.org/ (siehe S. 8).

[23] *Tensorflow*. URL: https://www.tensorflow.org/ (siehe S. 9).

Online-Quellen

[24] *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. Jan. 2019. URL: https://deepmind.com/blog/article/alphastar-mastering-real-time-strategy-game-starcraft-ii (besucht am 05. 03. 2020) (siehe S. 10).

[25] *MarI/O - Machine Learning for Video Gamesl*. Juni 2015. URL: https://youtu.be/qv6UVOQ0F44 (besucht am 05. 03. 2020) (siehe S. 10).

[26] *Vier Gewinnt Spiel*. März 2020. URL: https://cdn.anyfinder.eu/assets/5a6903838df2f5833889cbf45eb7599de9b858bb1821dc321540fb0bf4dca517 (besucht am 05. 03. 2020) (siehe S. 3).