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

Vorwort

TODO

Kurzfassung

TODO

Abstract

TODO engl.

Kapitel 1

Einleitung

In der Bachelorarbeit wurde ein Programm entwickelt, das mit Hilfe von reinforcement learning trainiert um das Spiel Vier Gewinnt zu erlernen. Es wird ein reinforcement learning Ansatz gewählt, da hierdurch automatisiert individuelle Gegner geschaffen werden können, die das Können unterschiedlicher Spieler widerspiegeln.

1.1 Motivation

Warum spielen wir? Wir spielen um Fähigkeiten zu erlernen, ein Kind lernt Objekte nach Formen zu sortieren oder eine Katze lernt spielerisch das Jagen von Beute. Doch auch erwachsene Menschen spielen noch und können hierdurch ihre Fähigkeiten verbessern. Um den bestmöglichen Effekt zu haben muss das Spiel einen gewissen Schwierigkeitsgrad haben der aber noch zu bewältigen sein muss. Da aber jeder Mensch, egal wie erfahren, das Spiel spielen soll ist es oft Hilfreich einen Gegner zu haben der ein ähnliches Level an Erfahrung mitbringt. Wenn nun diese Rolle des Gegners nicht von einem Menschen besetzt werden kann ist es naheliegend einen Computergegner zu erschaffen. Die Frage die ich in dieser Bachelorarbeit versuche zu beantworten ist: Ist es möglich, das Spiel Vier Gewinnt durch reinforcement learning zu erlernen und wie hilfreich ist dabei das Lernen vom Gegner.

1.2 Problemstellung

Ein Computergegner der für das Spiel Vier Gewinnt erschaffen wird kann auf verschiedenen Algorithmen basieren er könnte zum Beispiel in jedem Zug einfach zufällig eines der sieben möglichen Einwurflöcher bedienen oder er könnte durch einen Minimax Algorithmus zu jedem Zeitpunkt den bestmöglichen Zug errechnen. Das Problem hieran ist, dass jeder einzelne Algorithmus programmiert werden muss und somit ein enormer Arbeitsaufwand entsteht. Ein Lösungsansatz für diese Problem sind selbstlernende Algorithmen die mit Hilfe von reinforement learning und neuronalen Netzen selbstständig ähnlich wie ein Mensch erlernen wie das Spiel zu gewinnen ist. Hierraus können dann automatisch Computergegner generiert werden.

1.3 Umsetzung

Es wird eine digitale Version des Spiels Vier Gewinnt genutzt und eine Schnittstelle erschafft 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 dann trainiert werden ohne jedes einzelne Spiel gegen einen Menschen spielen zu müssen. Spielt der Computer soll zwischen verschieden effektiven Varianten von einem Minimax Algorithmus ausgewählt werden können. Somit kann der reinforcement learning Agent schnell viele Spiele gegen einen guten Gegner spielen und so effektiv lernen. Der Agent wird mit verschiedenen Voraussetzungen lernen um vergleichen zu können was besser und schlechter beim lernen hilft.

1.4 Evaluation

Je nachdem wie gut der selbstlernende Agent nach dem Lernen gegen unterschiedlich effektive Varianten des Minimax Algorithmus ist zeigt dann wie gut das reinforcement learning funktioniert hat. Hieran wird dann verglichen wie gut verschiedene Techniken des reinforcement learnings sind um den bestmöglichen Agenten zu erschaffen. Endgültig soll sich dann zeigen, dass durch das reinforcement learning Gegner auf automatisierte Weise geschaffen werden können die einen fein granularen Schwierigkeitsanstieg bieten können. TODO

1.5 Related work

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 [4], 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 [2], welche N-Tupel-Systeme nutzt, 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 2

Grundlagen

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

2.1 Vier Gewinnt

Vier Gewinnt ist ein Spiel für 2 Spieler in dem abwechselnd Spielsteine in ein vertikales Spielfeld der größe 7 * 6 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 bestimmtest Einwurfloch gesteckt wurden 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 [9]

2.2 Minimax Algorithmus

Ein Minimax Algorithmus beschreibt einen rekursiven Ansatz zum Finden einer optimalen Lösung für Probleme von zwei Parteien mit widersetzlich 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, dass 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 von den Spielern. 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, 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.

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.

2.2.2 Alpha-Beta Suche

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 klar besseren Pfad gibt.

2.3 Reinforcement Learning

Reinforcement learning (Bestärkendes Lernen) ist eine Maschine-Learning Methode bei der ein Agent mit Hilfe eines Environments, Policy, State, Actions und eines Rewards lernt, eine ihm gegebene Aufgabe zu lösen. Hierfür bieten sich Probleme an die als Markow-Entscheidungsproblem[3] formuliert werden können, da diese alles abdecken was für ein Environment wichtig ist. Es wird dabei Grundsätzlich wie in Abbildung 2.2 beschrieben vorgegangen. Der Agent bekommt also einen State vom Environment, und durch diesen sucht er sich mittels seiner Policy eine Action aus. Diese Action beinflusst 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 reinfocement learnings werden hier nun einmal genauer beschrieben.

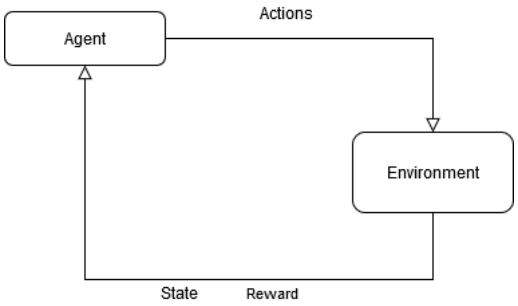


Abbildung 2.2: grundsätzliches Konzept von reinforcement learning

2.3.1 Environment

Das Environment beschreibt das gesamte Problem dass der Agent versucht zu lösen. Es beinhaltet alles was für das Spielen wichtig ist außer dem Spieler, welcher von dem Agenten behandelt wird. Dies ist bei den meisten Spielen ein Spielfeld mit den Positionen aller Akteure, Spielsteine oder sonstige 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 im Falle von physikalischen Spielen 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 optimalerZug 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 genutzt.

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.
TODO

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 einer bestimmten Zeitpunkt ausführen kann.

2.3.6 Reward

Der Reward ist die Belohnung die der Agent für seine Aktionen bekommt. Diese Belohnung beschreibt, wie gut es ist eine bestimmte Aktion auszuführen. Hieran passt der Agent dann wärend des Trainings seine Q-Values entsprechend seiner Policy an. Der Reward wird also benutzt um das Verhalten des Agenten zu beeinflussen. Möchte man ein bestimmtes Verhalten wird hierfür ein positiver Reward vergeben. Möchte man ein anderes Verhalten nicht 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 oder 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, denn sonst kann dies dazu führen, dass die Auswirkung von dem Teil mit kleineren Rewards von dem Teil mit größeren Rewards überschattet wird.

Bei Komplexeren Environments ist es schwer für jeden Zug vorauszusehen wie Zielführend er ist. Um diesem credit assignment problem entgegen zu wirken wird Q-

Learning genutzt. Deswegen wird hier dann eine Q-Funktion benutzt die anhand der gelernten Zustände versucht Fehlende zu Approximieren.TODO

2.3.7 Q-Funktion

TODO

2.3.8 Q-Values

TODO

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 neuronales Netz genutzt welches versucht eine Policy für diese erwarteten Rewards zu bestimmen. Dieses Verfahren nennt sich Deep Q-Learning. Hierbei wird ein Deep Q-Network erstellt, welches durch eine Sammlung zufälligen gespeicherter Spielen mit der aktuellen Policy diese anpasst. TODO

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 (Probabillity) 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.

2.3.11 Episode

Eine Episode beschreibt einen einzelnen Durchlauf des Agenten im Environment.

2.3.12 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 welches als ein OpenAI Gym Environment fungiert. An diesem Environment trainiert dann ein Agenten der mit Hilfe von Keras erschaffen wurde.

2.4.1 Python

Als Programmiersprache wurde sich für Python[7] entschieden da ein gutes Vorwissen in dieser Sprache vorhanden war und alle für diese Arbeit wichtigen Packages in die-

ser Programmiersprache existieren. Der Schöpfer von Python beschreibt es selbst als einfach zu erlernende objektorientierte Programmiersprache, welche Leistung mit einer klaren Syntax verbindet.[1]

2.4.2 Tensorflow

Bei Tensorflow [8] handelt es sich um ein open source Framework für Maschine-Learning welches ursprünglich für den internen Bedarf bei Google, für zum Beispiel Spracherkennung oder Google Maps, entwickelt wurde.

2.4.3 Keras

Keras[5] ist eine open source Bibliothek die auf Tensorflow aufbaut welche die Möglichkeit bietet vereinfacht neuronale Netze zu erstellen um diese in reinforcement learning Algorithmen zu benutzen.

2.4.4 OpenAI Gym

Gym [6] 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.

Kapitel 3

Konzept

Im Folgenden wird nun beschrieben was mit dieser Bachelorarbeit erreicht werden soll und wie dafür vorgegangen werden soll.

3.1 Zielsetzung

TODO mehr
Es soll gezeigt werden ob sich reinforcement learning zum erstellen von indviduellen Gegnern von Spielen eignet und wie sinnvoll hierbei das Lernen von einem guten Gegner ist.

3.2 Methode

Es wird ein Programm geschrieben, das durch reinforcement learning erlernt das Spiel Vier Gewinnt zu meistern. Um dies zu ermöglichen wird ein Gegner genutzt, der möglichst optimal spielt. Hierfür wird ein Minimax Algorithmus sorgen. Es wird eine optimierung der Reward-Funktion stattfinden um das Lernverhalten zu verbessern. Das Verhalten des guten Gegners soll mit in den Lernprozess des Agenten einfließen. Um den Lernfortschritt besser beurteilen zu können wird der Minimax-Algorithmus um eine Wahrscheinlichkeitsverteilung erweitert. Hierdurch soll dann das erlernte Verhalten an leicht schlechteren Gegnern getestet werden können. TODO mehr

Kapitel 4

Implementierung

DQN Algoritmus zum lernen des Spiels

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.
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.
Diese Implementierungen werden hier einmal genauer beschrieben:

4.1 Gegner mit Minimax Algorithmus

Der Minimax Algorithmus wurde als Teil des Environments implementiert und besteht im Groben aus zwei Teilen. Einmal die Funktion *check_next_actions* die den Suchbaum für die nächsten Züge aufbaut. Und die 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 einer Wahrscheinlichkeitsverteilung gearbeitet um den Gegner schlagbarer zu machen.

4.1.1 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.

4.1.2 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. Dazu wurde noch eine Funktion zur Wahrscheinlichkeitsverteilung implementiert, die dafür sorgt, dass der Minimax Algorithmus in bestimmten Situationen mit einer gewissen Chance nicht optimal spielt.

4.1.3 Wahrscheinlichkeitsverteilung

TODO

4.2 Gegner mit schlechten Strategien

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

4.2.1 Gegner mit deterministischen Aktionen

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

4.2.2 Gegner mit zufälligen Aktionen

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

4.3 Reinforcement Learning

TODO link

4.3.1 Environment

In Vier Gewinnt ist dies das Spielfeld und der Gegner. Das Spielfeld ist hier ein zweidimensionales Array der Länge 6 * 7. Der Gegner ist während des Lernens ein Computergegner der nach dem Minimax Algorithmus handelt.

4.3.2 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.

4.3.3 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 finden.

4.3.4 Actions

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

4.3.5 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:

$$\frac{Reward}{AnzahlZüge - 3}$$

Wie man in Abbildung 4.1 gut sehen kann wird somit das schnelle Gewinnen mehr belohnt (obere Linie) und das schnelle Verlieren mehr bestraft (untere Linie). Da nur wenn das Spielfeld komplett voll ist unentschieden gespielt wird, wird hier ein Reward von 0 gegeben (einzelner Punkt). Diese Reward-Funktion wird im weiteren Text als zugabhängige Reward-Funktion benannt.

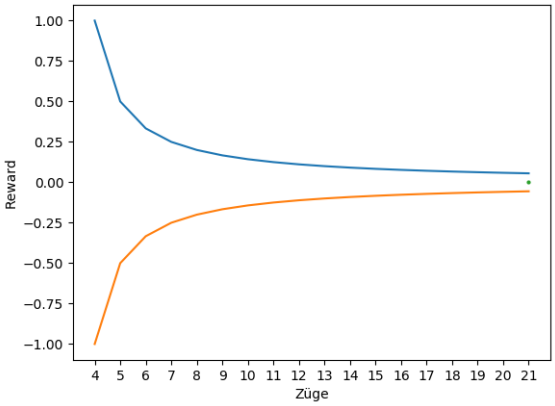


Abbildung 4.1: Reward Funktion

4.3.6 Policy

Da in Vier Gewinnt nur am Ende eines Spiels ein Reward ausgegeben wird, 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 Aktion, 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 Trainingalgorithmuses stark verlangsamen kann. Buch Seite 469 Für diese Proben werden dann 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 ziehlführende Policy.

Epsilon-Greedy

Als Unterstützung des Deep Q-Network wurde sich für die Epsilon-Greedy Policy entschieden, da diese sehr gut für Deep Q-Networks funktioniert Source. 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 natürlich Sinn, dass der Agent ganz viel erkundet, da er noch kein Wissen über das Spiel besitzt. Hat der Agent dann einige Spiele hinter sich kann er auf die ausbeutende Strategie umschalten. Um dieses Verhalten des erstigen 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 wärend der Lernphase langsam erhöht bis er etwas weniger als 1 erreicht hat. Somit ist das Ergebniss der Formel am Anfang meistens $True$ und später meistens $False$. Somit kann das DQN am Anfang viele Daten bekommen um hieraus eine Policy zu erarbeiten und dann diese später nach und nach verbessern, wenn durch die Ausbeutung dann nach dieser Policy vorgegangen wird.

4.4 Lernen vom Gegner

Um das Lernen des Agenten zu verbessern wurde sich dazu 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. Um diese Informationen aufnehmen zu können wurden die Daten über das aktuelle Spielfeld und den gemachten Zug angepasst. Hierfür wird das gesammte Spiel einen halben Zug in der Vergangenheit betrachtet. Das heißt der Agent versetzt sich in die Rolle seines Gegners. Da der Agent weiß, welche Zug sein Gegner gemacht hat kann er somit das negierte Spielfeld als Startzustand und den Zug seines Gegners als Aktion für sich nutzen.

4.5 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 gemacht. Beispiele hierfür findet man im Kapitel 5 Evaluierung.

4.6 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 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. Hierzu wird noch der Hintergrund Blau eingefärbt um dem Design des Spiels möglichst nah zu kommen. Abbildung 4.3 ist ein durch diese Weise erstelltes Spielfeld für das Interne Array 4.2

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

Abbildung 4.2: Array-Representation

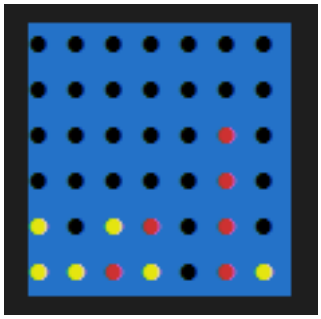


Abbildung 4.3: Das Spielfeld auf der Konsole

Kapitel 5

Evaluierung

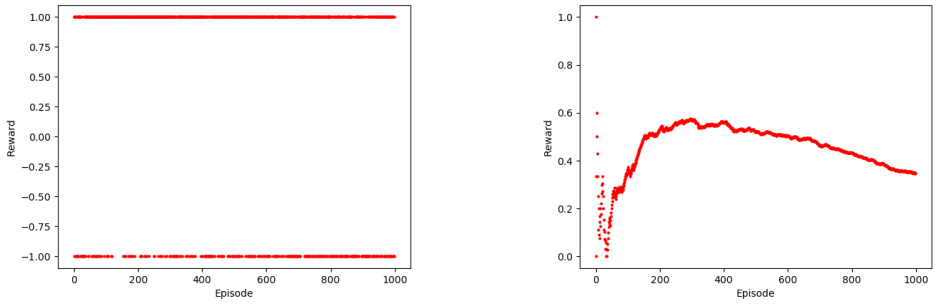
In diesem Kapitel wird nun der implementierte reinforcement learning Agent evaluiert. Hierfür wird zuerst, mit Hilfe von einem deterministischen Gegner und einem Gegner der in jedem Zug eine zufällige Aktion auswählt, gezeigt, ob die neu gewählte Reward-Funktion einen Vorteil beim Lernen gibt. Danach wird dann an diesen beiden Gegnern gezeigt, ob das Lernen mit Einbezug der gegnerischen Züge eine Veränderung im Lernverhalten zeigt. Hiernach wird dann 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 ein Paar von Graphen die selbe Lernepoche beschreibt. Der eine Graph beschreibt den absoluten Reward, also den genauen Wert, zu jeder Episode. Der andere Graph 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 Berrechnen von längeren Epochen einen sehr hohen Zeitaufwand bedeuten würde.

Es sei darauf hin zu weisen, dass der durchschnittliche Reward (immer der rechte Graph) meist nicht das komplette Spektrum von 1 bis -1 abdeckt. Dies wurde gemacht um eine bessere Erkennbarkeit des Ausschnittes zu gewähren.

5.1 Reward-Funktion

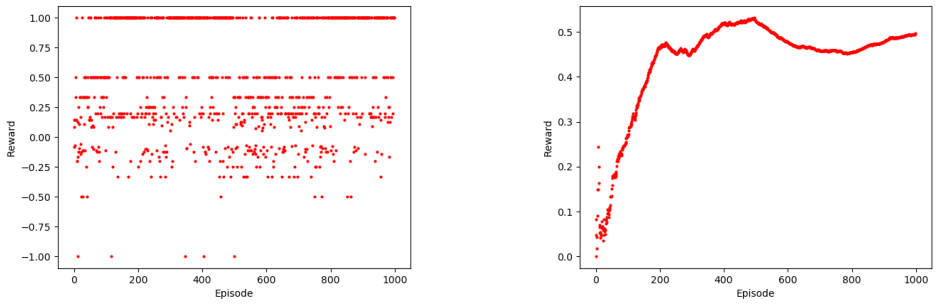
Als erstes wird die Veränderung durch die zugabhängige Reward-Funktion evaluiert. Hierfür wird Abbildung 5.1 mit Abbildung 5.2 sowie Abbildung 5.3 mit Abbildung 5.4 verglichen.
Abbildung 5.1 (b) und Abbildung 5.2 (b) verhalten sich fundamental sehr ähnlich. Beide erreichen einen durchschnittlichen Reward von 0.5 nach etwa 250 Episoden. Wobei die einfache Reward-Funktion aber einen abfallenden, und die zugabhängige Reward-Funktion einen konstanteren Trent aufweißt.

Abbildung 5.3(b) zeigt einen starken Abfall in gewonnenen Spielen nach etwa der 300. Episode. Dieses Verhalten ist auch sehr stark in Abbildung 5.3(a) zu erkennen.
Im Vergleich zu der einfachen Reward-Funktion wird in Abbildung 5.4(b) eine Annähe-



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

Abbildung 5.1: Reinforcement learning an zufällig werfendem Gegner mit einfacher Reward-Funktion und ohne die Züge des Gegners mitzulernen.



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

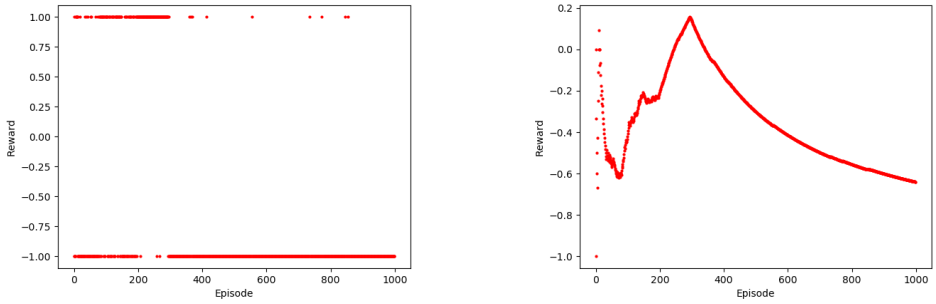
Abbildung 5.2: Reinforcement learning an zufällig werfendem Gegner mit verbesserter Reward-Funktion und ohne die Züge des Gegners mitzulernen.

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.

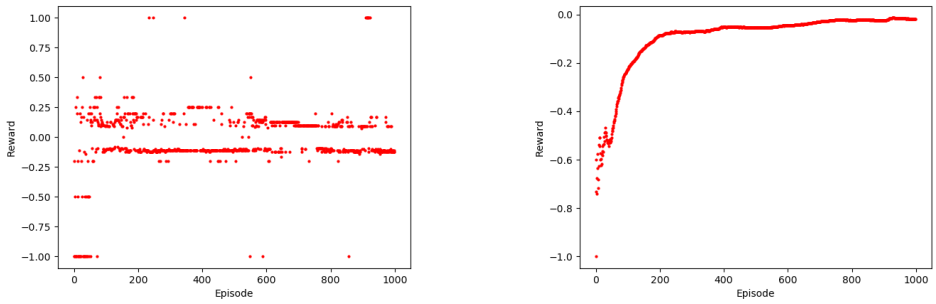
5.2 Lernen vom Gegner

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 5.5 (b) zu sehen ist, ist das Ergebnis hier im Vergleich zu Abbildung



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

Abbildung 5.3: Reinforcement learning an deterministisch werfendem Gegner mit einfacher Reward-Funktion und ohne die Züge des Gegners mitzulernen.

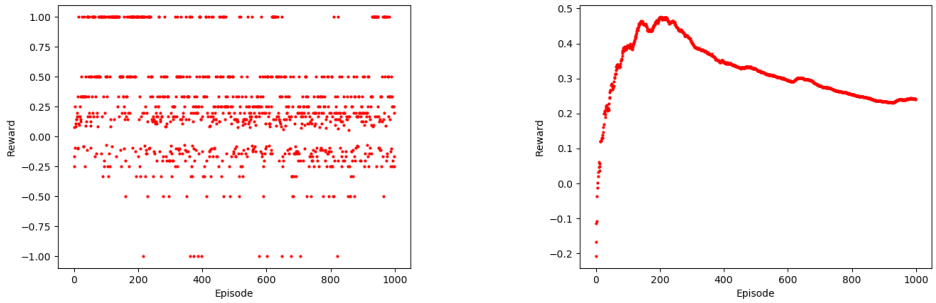


(a) absoluter Reward **(b)** durchschnittlicher Reward

Abbildung 5.4: Reinforcement learning an deterministisch werfendem Gegner mit verbesserter Reward-Funktion und ohne die Züge des Gegners mitzulernen.

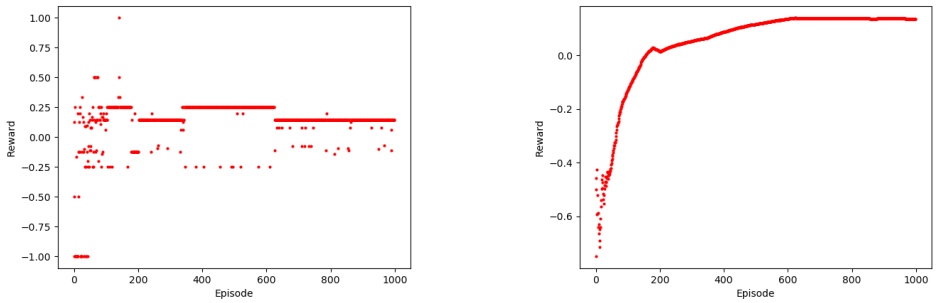
5.2 (b) wesentlich schlechter. Dies ergibt aber Sinn, da der Gegner ja komplett zufällig spielt und somit das Lernen seiner Züge eher als Nachteil betrachtet werden muss. Abbildung 5.6 zeigt hingegen eine sehr starke Verbesserung im vergleich zu Abbildung 5.4. Wie in Abbildung 5.6(a) zu sehen ist sind die meisten Rewards oberhalb von 0, dies war in Abbildung 5.4(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 5.5: Reinforcement learning an zufällig werfendem Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernent werden.

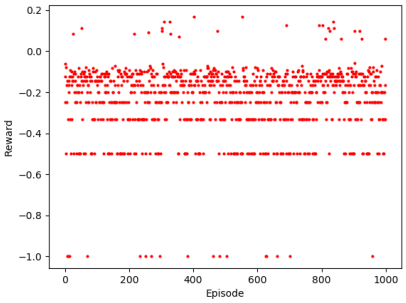


(a) absoluter Reward **(b)** durchschnittlicher Reward

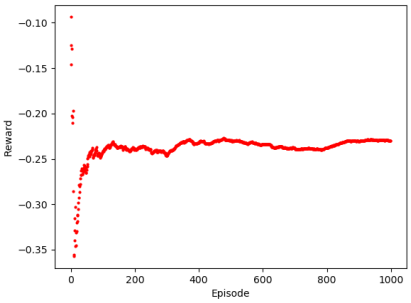
Abbildung 5.6: Reinforcement learning an zufällig werfendem Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernent werden.

5.3 Minimax

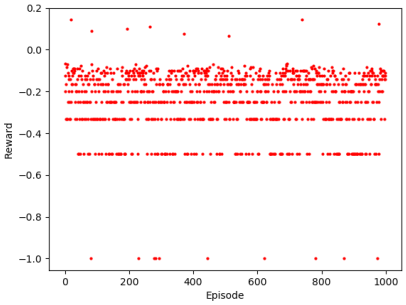
Die letzte Evaluierung sollte zeigen, dass ein reinforcement learning Agent mit den zuvor gezeigten Verbesserungen unterschiedlich schnell zu einer guten Strategie finden kann. Hierfür wurde der reinforcement learning Agent gegen den Minimax-Algorithmus mit verschieden starker Wahrscheinlichkeitsverteilung trainiert.
Die in Abbildung 5.7 und Abbildung 5.8 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 5.7 (a und b) hat somit die niedrigste Wahrscheinlichkeit den besten Zug zu wählen und das letzte Paar in Abbildung 5.8 (c und d) den höchsten.
Leider ließ sich hier kein Lernvortschritt feststellen nicht mal bei der niedrigsten Wahrscheinlichkeit.



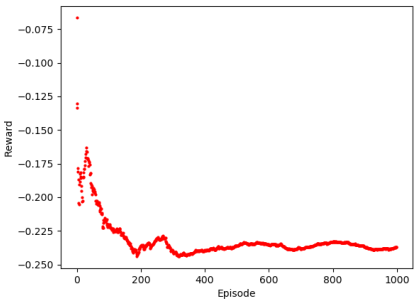
(a) absoluter Reward



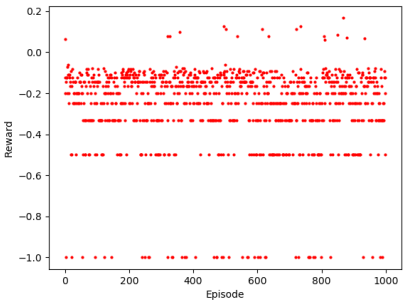
(b) durchschnittlicher Reward



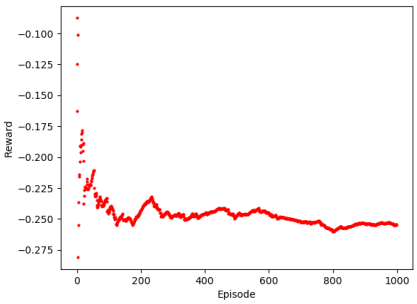
(c) absoluter Reward



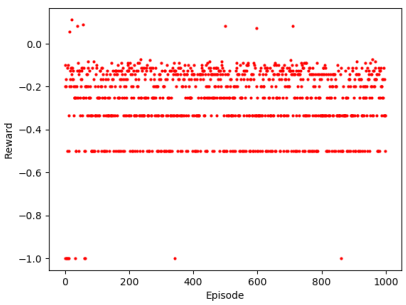
(d) durchschnittlicher Reward



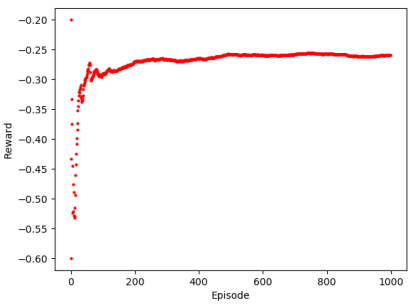
(e) absoluter Reward



(f) durchschnittlicher Reward

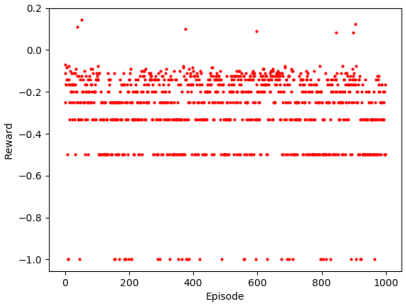


(g) absoluter Reward

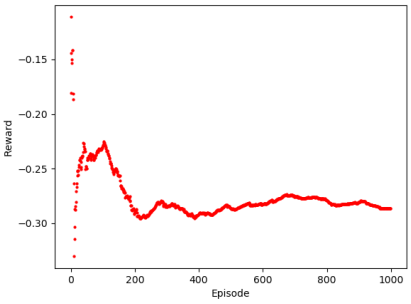


(h) durchschnittlicher Reward

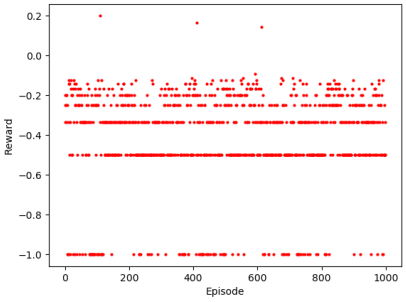
Abbildung 5.7: Reinforcement learning an Minimax Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernt werden. Teil 1



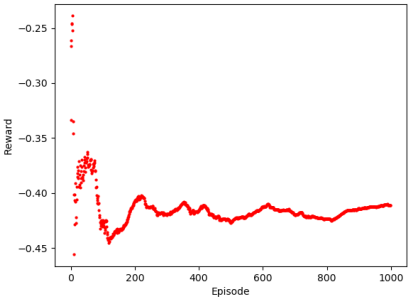
(a) absoluter Reward



(b) durchschnittlicher Reward



(c) absoluter Reward



(d) durchschnittlicher Reward

Abbildung 5.8: Reinforcement learning an Minimax Gegner mit verbesserter Reward-Funktion, wobei die Züge des Gegners mitgelernent werden. Teil 2

5.4 Besonderheiten

In den Graphen der Evaluation existiert eine Auffälligkeit die bei ungefähr 250 bis 300 Episoden auftritt. Hiernach wird das Lernverhalten des Agenten oft schlagartig schlechter. Dies ist am deutlichsten in der Abbildung 5.3(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 6

Fazit

Es wurde festgestellt, dass das Lernen der Züge des Gegners einen positiven Effekt auf das Lernverhalten eines reinforcement learning Agenten haben kann. Des weiteren wurde gezeigt, dass die gewählte Reward-Funktion angemessen war und somit zur Verbesserung des Lernverhalten beigetragen hat. Dies konnte leider nur am Beispiel von Gegnern mit schlechten und sehr einfach zielführenden Strategien, und nicht am Gegner mit sehr guter Strategie gezeigt werden. Ob sich dieses Verhalten auch gegen bessere Gegner zeigt, müsste in weiteren Untersuchungen durch Vereinfachung der Problemstellung oder durch Verlängerung der Lernepoche getestet werden. Diese Veränderungen waren auf Grund des Rahmens der Bachelorarbeit leider nicht möglich. TODO

Quellenverzeichnis

Literatur

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

[2] 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, Juni 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. 2).

[3] 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. 4).

[4] Lukas Stephan. „Implementierung und Evaluation eines intelligenten Spielagenten mit Reinforcement Learning“. Bachelorarbeit. Bremen, Deutschland: Universität Bremen, Informatik, Juni 2018. URL: unpublished (siehe S. 2).

Software

[5] *Keras*. URL: https://keras.io/ (siehe S. 8).

[6] *OpenAI Gym*. URL: https://gym.openai.com/ (siehe S. 8).

[7] *Python*. URL: https://www.python.org/ (siehe S. 7).

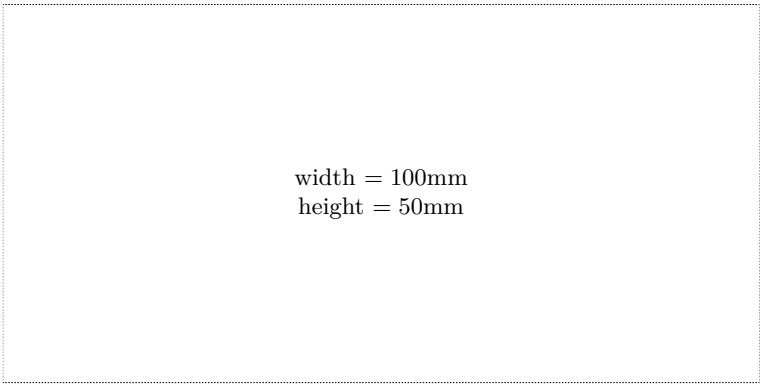
[8] *Tensorflow*. URL: https://www.tensorflow.org/ (siehe S. 8).

Online-Quellen

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

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —