Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Benjamin Wagener

Neuronale Netze als Näherungsverfahren für große Zustandsräume beim Reinforcement Learning

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Benjamin Wagener

Neuronale Netze als Näherungsverfahren für große Zustandsräume beim Reinforcement Learning

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

im Studiengang Bachelor of Science Angewandte Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. rer. nat. Michael Neitzke
Zweitgutachter: Prof. Dr. Olaf Zukunft

Eingereicht am: 23. Oktober 2015

Benjamin Wagener

Thema der Arbeit
Neuronale Netze als Näherungsverfahren für große Zustandsräume beim Reinforcement Learning

Stichworte
Künstliche Intelligenz, maschinelles Lernen, bestärkendes Lernen, künstliche neuronale Netze

Kurzzusammenfassung
Diese Arbeit untersucht, wie die Grenzen vom Reinforcement Learning bei großen Zustandsräumen überschritten werden können mittels künstlicher neuronaler Netze als Näherungsverfahren. Dies geschieht auf Basis einer Projektarbeit, in welcher ein Reinforcement Learing Agent entwickelt wurde, um Super Mario zu spielen. Als Spiel wird die frei verfügbare Java Version von MarioAI verwendet. Es wird erklärt, wie die beiden Lernverfahren kombiniert werden und welche Nutzen daraus gezogen werden können. Hierbei zeigt sich, dass die erreichte Verbesserung des Agenten insgesamt gesehen nicht den Aufwand rechtfertigt.
Benjamin Wagener

Title of the paper
Neural Networks as an approximation method for large state spaces in reinforcement learning

Keywords
Artificial Intelligence, Machine Learning, Reinforcement Learning, Artificial Neural Networks

Abstract
This thesis examines, how the limits of Reinforcement Learning can be exceeded for large state spaces by using artificial neural networks as an approximation method. This is done on the basis of a project, in which a reinforcement learning agent was developed to play Super Mario. The freely available Java version of MarioAI is used as game. It will be explained, how the two learning methods can be combined and what benefits can be drawn from it. Furthermore it will be shown, that the achieved improvements as a whole don't justify the needed effort.

Inhaltsverzeichnis

Tabellenverzeichnis

Abbildungsverzeichnis

1 Einleitung

Künstliche Intelligenz ist häufig Bestandteil von Science Fiction. Jedoch gibt es heutzutage schon Möglichkeiten, ein Computerprogramm intelligent handeln zu lassen und Probleme selbstständig durch Lernen zu erfüllen. Mittels maschinellen Lernens ist es möglich, Programmen die Möglichkeit zu geben, eigenständig Entscheidungen zu treffen oder auch die Lösung für ein Problem selbständig zu finden. Reinforcement Learning ist ein Teilbereich des maschinellen Lernens. Hierbei erlernt ein Programm die Lösung für eine Aufgabe selbständig durch Ausprobieren. Für durchgeführte Aktionen erhält das Programm Belohnungen, aufgrund dieser kann die Richtigkeit einer Aktion bewertet werden. Programme, die eigenständig handeln und lernen können werden auch Agenten genannt. Aber auch Reinforcement Learning hat seine Grenzen. Um diese zu überwinden, wird in dieser Arbeit versucht, mittels künstlicher neuronaler Netze in Kombination mit Reinforcement Learning die Grenzen zu überschreiten. Neuronale Netze können die Funktionsweise eines menschlichen Gehirns nachbilden. Sie werden häufig dafür eingesetzt, Vorhersagen zu treffen in der Wirtschaft und Statistik, eignen sich jedoch auch z.B. für Handschriftenerkennung und vieles mehr. Neuronale Netze erlernen ihr Wissen in speziellem Training und können sich die Lösung nicht selbstständig durch Ausprobieren erarbeiten wie ein Reinforcement Learning Agent.

1.1 Motivation

In einer Projektarbeit an der HAW Hamburg von Ruben Christian Buhl, Alexander Sawadski, Wlad Timotin und dem Verfasser wurde versucht, einem Agenten mittels Reinforcement Learning beizubringen, das Spiel Super Mario zu spielen. Wie gut der Computer spielt, wird anhand der erreichten Siege über eine bestimmte Anzahl an durchlaufenden Leveln bewertet. Diese Arbeit baut auf der Projektarbeit auf und versucht, die erreichten Ergebnisse zu verbessern. Hierzu wird eine Kombination von Reinforcement Learing und neuronalen Netzen verwendet, um die Grenzen vom Reinforcement Learing zu überschreiten.

1.2 Aufbau der Arbeit

In Kapitel 2 werden zunächst die Grundlagen der einzelnen Themenbereiche vermittelt. Dies beinhaltet eine Einführung in Reinforcement Learning und neuronaler Netze sowie eine kurze Erklärung von Super Mario.

Kapitel 3 analysiert das Spiel tiefer im Bereich des Reinforcement Learnings nach dem aktuellen Stand der Projektarbeit und zeigt auf, in welchen Bereichen Verbesserungsmöglichkeiten, falls vorhanden, bestehen und wie diese umgesetzt werden können.

Darauf folgend beschreibt Kapitel 4, wie genau die Verbesserungsmöglichkeiten umgesetzt und implementiert werden.

Im vorletzten Kapitel 5 werden die Ergebnisse der Tests aufgezeigt und näher betrachtet.

Im letzten Kapitel 6 wird die Arbeit zusammengefasst und ein abschließendes Fazit über die Veränderungen gezogen.

2 Grundlagen

Dieses Kapitel vermittelt die wichtigen theoretischen Grundlagen, die für diese Arbeit benötigt werden.

2.1 Reinforcement Learning

Die Grundlagen zum Reinforcement Learning basieren auf dem Werk 'Reinforcement Learning An Introduction' von Richard S. Sutton und Andrew G. Barto (Sutton und Barto (1998)).

Die natürlichste Art, etwas zu lernen, ist Lernen durch sogenanntes 'try and error'. Schon Kleinkinder und auch Tiere fangen so an, z.B. das Laufen zu erlernen. Dies ist ein natürlicher Prozess, welcher meistens unterbewusst vonstatten geht. Auch Gesprächsführung wird so erlernt. Es wird dabei mit der Umgebung interagiert und die Reaktion beobachtet. In einem Gespräch wäre dies die Antwort des Gesprächspartners. So kann z.B. erlernt werden, das Gespräch zu einem gewünschtem Ergebnis zu führen. Bei der Reaktion der Umwelt wird hier meistens von Belohnungen oder auch Bestrafungen gesprochen. Ein Kind, welches beim Versuch zu laufen hinfällt, wird Schmerzen verspüren und so lernen, dass der unternommene Versuch nicht richtig war. Es wird etwas anders probieren, um erneutes Hinfallen zu vermeiden. Dabei wird ganz automatisch versucht, Belohnungen zu maximieren und Bestrafungen zu minimieren.

Reinforcment Learning setzt diesen Ansatz um bezogen auf Computer / Programme und ist ein Bereich des maschinellen Lernens. Ein Programm versucht hierbei wie in der Natur durch Interaktion mit seiner Umwelt erhaltene Belohnungen zu maximieren und Bestrafungen zu minimieren. Da ein Computer jedoch keine Schmerzen verspüren kann, handelt es sich bei den Belohnungen um Zahlenwerte. Der Prozess des Lernens kann als einfacher Kreislauf dargestellt werden, Abbildung 2.1 zeigt so einen.

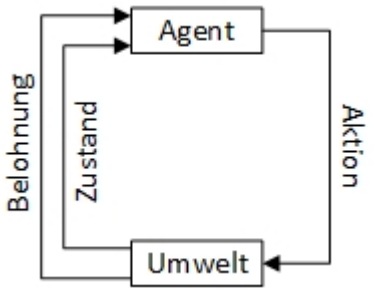

Abbildung 2.1: Reinforcement Learning Kreislauf

Das Programm, welches mit der Umwelt interagiert und lernen soll, wird Agent genannt (weiterhin auch als 'RL- Agent' bezeichnet für Reinforcement Learning Agent). Die Umwelt teilt dem Agenten mit, in welchem Zustand er sich befindet. Was genau ein Zustand beschreibt, ist von der jeweiligen Anwendung abhängig. Bei einem Programm, welches Autofahren lernen soll, könnte dies z.B. die aktuelle Geschwindigkeit des Autos sowie Pedal- und Lenkradstellung sein. Der Agent wählt daraufhin eine passende, ihm zur Verfügung stehende Aktion aus und teilt diese der Umwelt mit. Diese reagiert auf die Aktion und teilt dem Agenten den neuen Zustand mit sowie die Höhe der Belohnung. Der Agent nimmt die Belohnung war und nutzt diese, um das weitere Verhalten ggf. anzupassen. Es wird hierbei pro Zustand jeweils ein Wert für jede mögliche Aktion gespeichert. Dieser Wert wird Q- Wert genannt und gibt dem Agenten Auskunft darüber, wie groß die zu erwartende Belohnung ist, wenn die entsprechende Aktion ausgeführt wird. Wie genau dieser Wert berechnet wird, ist vom verwendeten Reinforcement Learning Algorithmus abhängig. In dieser Arbeit wird der sogenannte SARSA- Algorithmus verwendet. Dieser verwendet folgende Updatefunktion für Q- Werte:

$$Q_{(s,a)} = Q_{(s,a)} + \alpha[r + \gamma Q_{(s',a')} - Q_{(s,a)}]$$

$Q_{(s,a)}$ ist der Q- Wert für die Aktion a im Zustand s. $Q_{(s',a')}$ entsprechend für den Q- Wert der Folgeaktion im Folgezustand. Bei γ handelt es sich um den sogenannten Discount- Faktor. Dieser ist notwendig bei Anwendungen, in denen es zu keinem Ende kommt. Hierbei könnte ohne diesen die theoretische Belohnung unendlich groß werden. r ist der Wert der erhaltenen Belohnung für die Aktion a im Zustand s. α bezeichnet die Lernrate des Agenten. Diese bestimmt, wie viel Einfluss neu erlerntes Wissen auf das schon vorhandene hat. Durch diese Funktion entstehen so für jede Aktion in jedem Zustand entsprechende Werte. Wenn der Agent die Auswahl der Aktion treffen muss, ist es jedoch nicht optimal, wenn er immer

die Aktion mit dem höchsten Wert wählt. Es ist auch wünschenswert, dass der Agent auch optimale Aktionen ausprobiert und diese 'erkundet', um sich an neue Situationen anpassen zu können oder auch eine optimale Strategie zu finden. Als Beispiel: Es gibt 2 Aktionen zur Auswahl in einem beliebigen Zustand, eine Aktion ist mit dem Wert 1 bewertet und die andere wurde noch nie ausprobiert und ist deshalb mit 0 bewertet. Wenn der Agent immer die am höchsten bewertete Aktion wählt, würde er nie herausfinden, dass die 2. Aktion z.B. mit 2 bewertet sein könnte, weil sie viel besser ist. Die Strategie, immer die Aktion mit der höchsten Bewertung zu wählen, nennt sich 'greedy'. Beim Ausprobieren von nicht optimalen Aktionen spricht man vom Erkunden bzw. 'exploration'. Wie viel ein Agent erkundet, wird mittels einer Erkundungsrate ϵ angegeben. Die sogenannte 'ϵ- greedy' Strategie vereint, wie der Name schon vermuten lässt, die greedy Strategie mit einer Explorationsrate. Hierbei führt ein Agent bei z.B. 10% seiner Entscheidungen eine zufällige Aktion aus und sonst immer die mit der höchsten Bewertung. Diese Strategie wird auch bei dem in dieser Arbeit verwendeten Agenten angewendet.

2.2 MarioAI

Super Mario ist eine sehr erfolgreiche 'Jump 'n' Run' (aus dem englischem 'Jump and Run') Spielreihe der Firma Nintendo. Bei Jump 'n' Run Spielen muss ein Spieler meistens eine Spielfigur innerhalb eines zweidimensionalen Levels vom linken Rand eines Levels zum rechten, durch geschicktes Laufen und Springen bewegen. Hierbei gilt es, Hindernisse zu überwinden und Gegnern auszuweichen oder sie auszuschalten z.B. durch Draufspringen auf diese. Bei der Spielfigur handelt es sich in diesem Fall um Mario. Das Spiel ist beendet, wenn Mario durch eine Berührung eines Gegners stirbt, das Ende des Levels erreicht oder die Zeit abgelaufen ist. Stirbt Mario z.B. durch das Berühren eines Gegners, wird ihm ein sogenanntes Leben abgezogen und das aktuelle Level kann vom Start nochmals versucht werden. Sind alle Leben verbraucht, ist das gesamte Spiel beendet. Durch das Einsammeln von 100 Münzen erhält Mario ein weiteres Leben dazu. Da es in MarioAI keine Leben gibt, werden Münzen nicht weiter beachtet. Mittels sogenannter Powerups wird Mario größer und stärker. Es gibt folgende Stadien, in denen sich Mario befinden kann:

- Klein
- Groß
- Feuermario

Marios Zustand	Gegner berühren	Pilz- Powerup	Feuerblumen- Powerup
Klein	tot	Groß	Feuermario
Groß	Klein	Groß	Feuermario
Feuermario	Groß	Feuermario	Feuermario

Tabelle 2.1: Powerup- Tabelle

Die Tabelle 2.1 zeigt das verhalten von Marios Zustand bei den verschieden Aktionen. Ist Mario klein, so stirbt er bei der Berührung eines Gegners. Wird ein Pilz- Powerup eingesammelt, wächst er und wird groß. Wenn ein Gegner in diesem Zustand berührt wird, stirbt Mario nicht, sondern wird klein. Durch das Einsammeln einer sogenannten Feuerblume (ebenfalls ein Powerup) wird Mario zu Feuermario. Hierbei erlangt er die Fähigkeit, Gegner auch durch das Verschießen von Feuerbällen zu erledigen. Wird ein Gegner berührt, gelangt Mario in den Zustand 'Groß' und kann somit 2 Gegner berühren, bevor er stirbt. Powerups sind in einem Level immer in speziellen Blöcken versteckt im Gegensatz zu Münzen, welche auch offen verteilt in einem Level zu finden sind.

MarioAI [1] ist eine frei verfügbare Java Version, basierend auf der Super Mario Reihe, die z.B. für den Einsatz mit Reinforcement Learning Agenten ausgelegt ist. Das Spiel stellt die Umwelt dar und der Agent übernimmt die Rolle des Spielers. Es ist zu beachten, dass der Agent nicht Mario ist, sondern nur Mario steuert. Ein Zustand wäre für einen Menschen das aktuell sichtbare Bild auf einem Bildschirm. Auf Basis dieses Bildes kann ein Spieler sich für seine nächste Aktion entscheiden. Dies bedeutet, dass diese Art von Spiel die Markov Eigenschaft besitzt. D.h., dass im aktuellen Zustand alle Informationen enthalten sind, um das Spiel fortführen zu können und keine Kenntnis über vorherige Aktionen benötigt wird. Schach besitzt z.B. auch diese Eigenschaft. Die aktuelle Situation auf einem Schachbrett reicht als Information aus, so dass ein anderer Spieler zu jederzeit übernehmen und weiterspielen könnte. Es wird keine Kenntnis über vorherige Züge benötigt (vgl. Sutton und Barto (1998) S.62).

[1] URL: http://www.marioai.org/

2.3 Neuronale Netze

Die Grundlagen zu neuronalen Netzen basieren auf der Einführung zu neuronalen Netzen von Günter Daniel Rey und Fabian Beck (KNN) und der Bachelorarbeit von Bianca Ott (Ott (2013)).

Das menschliche Gehirn besteht aus einem großen Netz von Neuronen. Angelehnt an dieses können Computer sogenannte künstliche neuronale Netze simulieren. Diese dienen der Forschung, um das Gehirn besser zu verstehen. Sie werden heutzutage jedoch auch viel in der Statistik und Technik verwendet, um verschiedenste Probleme zu lösen und sind auch ein Teilbereich des maschinellen Lernens. Abbildung 2.2 zeigt wie so ein Netz aufgebaut sein kann.

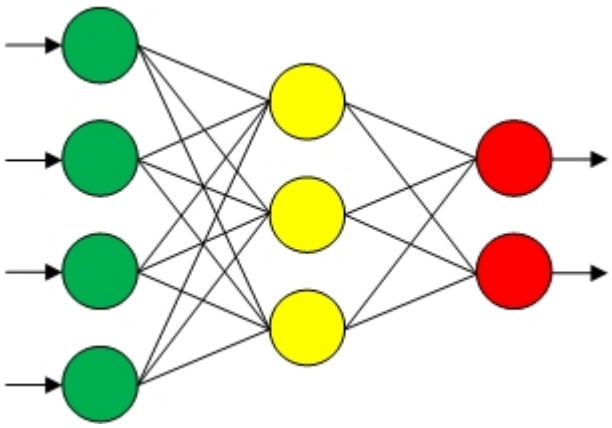

Abbildung 2.2: Künstliches neuronales Netz

Die Kreise stellen die einzelnen Neuronen dar. Die Kreise / Neuronen, welche untereinander stehen, bilden eine sogenannte Schicht. Jedes Neuron ist mit jedem Neuron der Folgeschicht verbunden. Die grün markierten Neuronen stellen die Eingangsschicht dar. Man spricht auch von der Input- Schicht und den Input- Neuronen. Die roten Neuronen bilden die Ausgangsschicht, auch Output- Schicht bzw. Output- Neuronen genannt. Ein minimales neuronales Netz besitzt mindestens eine Eingangsschicht und eine Ausgangsschicht. Weitere Schichten dazwischen werden Hidden- Schichten bzw. Hidden- Neuronen genannt, da sie im inneren des Netzes sozusagen verborgen sind. Aus wie vielen Schichten und Neuronen ein neuronales Netz besteht, ist vom tatsächlichen Anwendungsfall abhängig.
Um ein neuronales Netz zu verwenden, wird ein Eingangssignal auf die Eingangsneuronen gelegt. Neuronen besitzen eine Aktivierungsfunktion, mit deren Hilfe entschieden wird, ob ein Neuron aktiv ist oder nicht. Diese Funktionen können sehr verschieden sein, z.B. könnte es eine binäre oder auch lineare Funktion sein. Was für eine Funktion verwendet wird, ist wiederum

vom Anwendungsfall abhängig. Das Ausgangssignal eines Neurons wird über eine sogenannte gewichtete Verbindung an alle Neuronen der Folgeschicht geleitet. Diese Gewichtung kann man sich als Multiplikationsfaktor vorstellen. Dies dient dazu, um bestimmten Neuronen mehr oder weniger Wichtigkeit zuzuordnen. Wenn davon gesprochen wird, das ein neuronales Netz lernt, bedeutet dies meistens, dass die Verbindungsgewichte im Netzwerk angepasst werden.

Das Lernen von neuronalen Netzen ist sehr unterschiedlich zu dem eines RL- Agenten. Während dieser einfach anfangen kann und durch simples Ausprobieren selbstständig dazulernt, muss ein neuronales Netz mit Trainingsdaten trainiert werden, es wird hier auch von überwachtem Lernen gesprochen. Dies bedeutet, dass zu einem Eingangssignal das richtige Ausgangssignal bekannt sein muss. Beim Training wird das Eingangssignal angelegt und das Ausgangssignal mit dem gewünschtem Signal verglichen. Der Unterschied zwischen dem Ist- und Soll- Signal ist der Fehler eines neuronalen Netzwerks. Diesen versucht der Trainingsalgorithmus für das Netz möglichst zu verringern. Ein Trainingsdatensatz besteht meistens aus mehreren Trainingsdaten. Man kann sich dies wie eine Ansammlung von Fragen und Antworten vorstellen. Wenn alle Daten aus einem Satz durchgearbeitet sind, d.h. alle Fragen gestellt wurden, bedeutet dies, dass eine Trainingsinteration durchgeführt wurde. Wenn von einem Trainingsfehler gesprochen wird, so ist dies der Fehler über den ganzen Trainingsdatensatz. Dies könnte z.B. der Durchschnitt sein aus allen gestellten Anfragen an das Netz. Wann genau die Verbindungsgewichte verändert werden, ist vom Lernalgorithmus abhängig. Dies könnte z.B. nach jeder Anfrage oder auch erst nach einer kompletten Iteration geschehen. Als Trainingsalgorithmus wird meistens eine Form des sogenannten Backpropagation Verfahrens verwendet, besonders wenn Hidden- Schichten im Netzwerk benutzt werden. Nachdem ein Netz trainiert wurde, wird ein Test oder auch Validierung durchgeführt. In dieser werden dem Netz bisher unbekannte Anfragen gestellt um zu überprüfen, wie gut das Netz seine Aufgabe erlernt hat.

Wie viele Neuronen die Eingangs- und Ausgangs- Schicht haben, ergibt sich aus dem jeweiligen Anwendungsfall. Die Anzahl an Hidden- Schichten und den darin enthaltenen Neuronen ist jedoch frei wählbar. Eine optimale Anzahl dieser für den jeweiligen Anwendungsfall zu finden ist von größter Wichtigkeit, da die Hidden- Neuronen maßgeblich verantwortlich sind, wie gut ein Netzwerk lernt. Werden zu wenig verwendet, kann die Aufgabe nicht richtig erlernt werden, da das Netzwerk nicht komplex genug ist. Sind zu viele Hidden- Neuronen vorhanden, ist das Netz zu leistungsstark und lernt die Trainingsdaten auswendig. Dies führt zwar zu einem kleinen Trainingsfehler, jedoch können unbekannten Anfragen häufig nicht richtig beantwortet werden, welches an großen Fehlern bei der Validierung zu erkennen ist. Man

spricht hierbei von Over- und Underfitting. Es gibt keine Formel, welche immer eine optimale Anzahl an Hidden- Neuroen für ein Netzwerk berechnen könnte, sondern lediglich Faustformeln. Deshalb ist es nötig, bei der Entwicklung von Anwendungen mit neuronalen Netzen verschieden konfigurierte Netze vorher zu testen und so ein passendes Netz zu finden.

3 Analyse

In diesem Kapitel werden Aktionen, Zustände und Verbesserungsmöglichkeiten eines Reinforcement-Learning-Agenten in MarioAI analysiert.

3.1 Aktionsanalyse

In MarioAI gibt es vier Steuerungstasten.

- Links
- Rechts
- Springen
- Laufen/Schießen

Die Aktionsmöglichkeiten eines Spielers bestehen aus den vier Steuerungstasten sowie sämtlichen Kombinationsmöglichkeiten der Tasten. Tabelle 3.1 veranschaulicht dies.

Links	Rechts	Springen	Laufen/Schießen	Aktion
0	0	0	0	0
0	0	0	1	1
0	0	1	0	2
0	0	1	1	3
0	1	0	0	4
0	1	0	1	5
0	1	1	0	6
0	1	1	1	7
1	0	0	0	8
1	0	0	1	9
1	0	1	0	10
1	0	1	1	11
~~1~~	~~1~~	~~0~~	~~0~~	~~12~~
~~1~~	~~1~~	~~0~~	~~1~~	~~13~~
~~1~~	~~1~~	~~1~~	~~0~~	~~14~~
~~1~~	~~1~~	~~1~~	~~1~~	~~15~~

Tabelle 3.1: Aktionstabelle

Eine 1 steht hier für eine gedrückte Taste und eine 0 entsprechend für nicht gedrückt. Die untersten vier Einträge sind durchgestrichen, da bei diesen Kombinationen sowohl die 'Links'- als auch die 'Rechts'-Taste gedrückt werden und dies keine sinnvollen Aktionen im Spiel sind, da Mario sich immer nur in eine der beiden Richtungen bewegen kann. Somit gibt es zwölf sinnvolle Aktionen, die ein Agent in jedem Zustand zur Verfügung hat.

3.2 Zustandsanalyse

Für einen menschlichen Spieler ist der aktuelle Zustand des Spiels immer das komplette, sichtbare Bild wie in Abbildung 3.1.

Abbildung 3.1: Eine Situation in MarioAI

In MarioAI ist für Agenten ein Gitternetz eingebaut, durch welches die Informationen von Marios Umwelt im Programm verarbeitet werden können. Abbildung 3.2 zeigt dieses Gitternetz. Jeder Block des Gitternetzes kann auf seine Eigenschaften bzgl. 'LevelScene' sowie 'Enemies'[1] abgeprüft werden. Über 'LevelScene' erhält man Informationen über statische Objekte wie z.B. Hindernisse, über welche Mario springen müsste. Mit 'Enemies' kann geprüft werden, ob sich Gegner in diesem Block befinden. Diese sind meistens dynamisch und bewegen sich durch das Level.

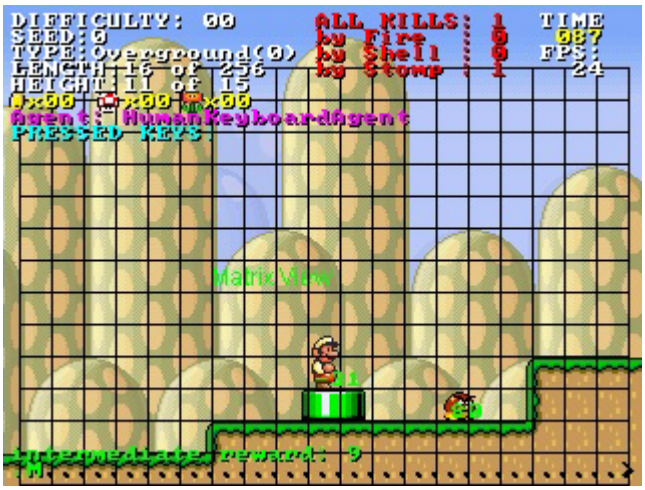

Abbildung 3.2: Eingebautes Gitternetz

[1] 'LevelScene' sowie 'Enemies' sind die Namen der entsprechenden Variablen im Quellcode.

3.3 Zustandsmodellierung

Die intuitivste Modelierung des Zustands wäre wohl einfach, jeden Block des Gitters abzuprüfen und aus diesen Informationen einen Zustand zusammenzusetzen, indem man eine lange Zahl aus den einzelnen Werten bildet. Das Gitternetz besteht aus 247 Blöcken, wenn man horizontal alle und vertikal bis 3 Blockreihen unter Mario mit einbezieht: Also alle sichtbaren Blöcke in Abbildung 3.2 ($19 \cdot 13 = 247$). 'LevelScene' liefert für einen Block 12 verschiedene Werte, 'Enemies' 13 Werte. Dies führt dazu, dass es eine sehr große Anzahl an möglichen Zuständen gibt: $12^{247} \cdot 13^{247} = 5 \cdot 10^{541} Zustände$. Diese können in einer Datenbank wie Tabelle 3.2 mit den 12 Q-Werten gespeichert werden. Die Anzahl der Zustände, multipliziert mit den 12 Aktionen, ergibt die Anzahl an Double-Werten (Dezimalzahlen) in der vollständigen Datenbank. $Zustände \cdot Aktionen = 5 \cdot 10^{541} \cdot 12 = 6 \cdot 10^{542} Double - Werte$. Pro Double- Wert werden in Java 8 Byte an Speicherplatz benötigt. $6 \cdot 10^{542} \cdot 8Byte = 4,8 \cdot 10^{543} Byte$. Der benötigte Speicherplatz steht in keinem Verhältnis zu einem nur ein paar Kilobyte großem Spiel. Hierbei ist der Zustand von Mario selbst noch nicht einmal mitbedacht, denn dies würde die Anzahl der Zustände nochmals vervierfachen. Die Gesamtzahl der Zustände muss folglich stark verringert werden.

Zustand	**a1**	**a2**	**a3**	**a4**	**a5**	**a6**	**a7**	**a8**	**a9**	**a10**	**a12**	**a12**
0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Tabelle 3.2: Datenbanktabelle

3.3.1 Generalisierung / Zoom-Level

Eine Möglichkeit, die Zustände zu verringern, ist es, die Werte von 'LevelScene' sowie 'Enemies' zu generalisieren. MarioAI nennt dies 'Zoom-Level' und stellt 3 verschiedene zur Verfügung. Je nach 'Zoom-Level' wird von 'LevelScene' nur noch zurückgegeben, ob ein Hindernis vorhanden ist und nicht mehr, um welchen Typ von Hindernis es sich genau handelt. Das gleiche gilt für Gegner. Hier wird nicht mehr unterschieden, um welchen Typ von Gegner es sich genau handelt, sondern nur noch gemeldet, dass ein Feind vorhanden ist. Durch die Abstraktion sollten die möglichen Werte eines Blockes so gering wie möglich sein, ohne dabei Informationen zu verlieren, die für ein erfolgreiches Durchlaufen des Levels wichtig sind.

In der Landschaft von Mario gibt es drei Arten von Blöcken:

- Landschaftsblock
- Ziegelblock
- ?-Block[2]

Landschaftsblöcke bilden den Boden sowie die Hindernisse eines Levels und sind nicht zerstörbar. '?-Blöcke' enthalten 1-n Münzen oder ein so genanntes Powerup (siehe Kapitel 2.2). Um an die Münzen oder das Powerup zu kommen, muss Mario von unten gegen den Block springen. Münzen werden dabei automatisch eingesammelt. Ein Powerup erscheint oberhalb des Blocks und muss von Mario explizit eingesammelt werden, um es zu nutzen. Nachdem der ?-Block entleert wurde, bleibt er trotzdem als Block weiterhin bestehen und kann als Plattform genutzt werden. Bei Ziegelblöcken gibt es zwei Möglichkeiten: Entweder sie werden durch das Gegenspringen von unten komplett zerstört und hinterlassen nichts oder sie verhalten sich wie ?-Blöcke. Da es sehr gut möglich ist, ein Level ohne das Einsammeln von Powerups zu bestehen, da Mario bereits als Feuermario startet und Ziegelblöcke niemals ein Hindernis darstellen, welches nur durch ihre Zerstörung überwindbar wäre, wurde sich im Projekt dafür entschieden, die Art der Blöcke nicht zu unterscheiden und durch das Weglassen von Powerups die Aufgabe für einen Agenten weniger komplex zu gestalten.

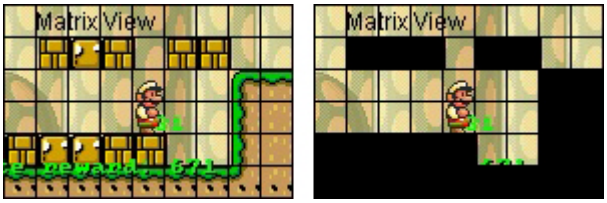

Abbildung 3.3: Links: tatsächliches Level, Rechts: abstrahiertes Level

Abbildung 3.3 zeigt im linken Bild den Ausschnitt des tatsächlichen Levels und rechts, wie es ein Agent sehen würde nach der Abstraktion der Blöcke, hier als schwarze Felder dargestellt. Wie zu sehen ist, gehen dabei keine Informationen über die Bewegungsmöglichkeiten in einem Level verloren.

Sogenannte Brücken stellen eine eigene Art von Feld dar und sind eine wichtige Art, sich in einem Level vertikal zu bewegen und werden deshalb gesondert abgefragt.

[2] ?-Blöcke (Fragezeichen-Blöcke) werden wegen ihres Aussehens so genannt.

In Mario-Spielen gibt es eine Vielzahl von verschiedenen Gegnern. Jeder Feind ist dabei in seinem Verhalten einzigartig und es wäre eine starke Einschränkung, die Gegner zu verallgemeinern. Auf dem Schwierigkeitsgrad des Agenten gibt es jedoch nur drei verschiedene Arten von Gegnern.

- Goomba
- Piranha Pflanze
- Prinzessin

Goombas stellen die einfachsten Gegner im Spiel dar und sind durch Draufspringen zu töten. Piranha Pflanzen sind der Venus Fliegenfalle nachempfunden und ein Hineinspringen verletzt Mario. Sie sind nur durch Feuerbälle von Mario zu töten oder müssen umgangen werden. Die Prinzessin ist kein Gegner im eigentlichen Sinne, sondern sie markiert das Levelende. Um dennoch die Anzahl an Zuständen reduzieren zu können, wurde im Projekt eine Möglichkeit für eine Entfernungsunschärfe eingebaut. Da Mario nur Gegner, die wenige Blöcke weit von ihm entfernt sind, besiegen kann, ist es immer weniger wichtig, um was für einen Gegner es sich handelt, je weiter er entfernt ist. Durch die Unterteilung von 'Enemy'- und 'Detailed Enemy'-Blöcken ist es möglich, in einem Sichtfeld genau diese Entfernungsunschärfe zu erzeugen, wenn es gewollt wird.

Ein Sichtfeld kann deshalb aus den folgenden Blöcken bestehen.

- Block
  - 0 = Kein Hindernis
  - 1 = Hindernis
- Bridge
  - 0 = keine Brücke
  - 1 = Brücke
- Enemy
  - 0 = Kein Gegner
  - 1 = Gegner
  - 2 = Prinzessin
- Detailed Enemy
  - 0 = Kein Gegner
  - 1 = Gegner
  - 2 = Gegner (durch Draufspringen nicht zerstörbar)
  - 3 = Prinzessin

3.3.2 Sichtfeld

Um die Anzahl der möglichen Zustände weiter zu reduzieren, wurde im Projekt das Konzept eines Sichtfelds oder auch Sichtbereichs für Mario mit eingebaut. Die Idee dahinter ist es, nicht alle Blöcke des Gitternetzes abzufragen, sondern nur einen im Verhältnis kleinen Teil. Abbildung 3.4 zeigt ein mögliches Sichtfeld für die 'LevelScene'. Mit 'BL' gekennzeichnete Felder fragen den zugehörigen Block im Gitternetz auf die 'Block' Kategorie ab, 'BR' prüft auf die 'Bridge' Kategorie. Es werden die oben genannten Werte zurückgemeldet. Zudem wurde die Möglichkeit eingebaut, einzelne Blöcke zu einer 'Zone' zusammen zu fassen. Nummer 1 und 5 in Abbildung 3.4 sind solche Zonen. Die gelben Felder repräsentieren die Position von Mario. Die Abbildung 3.5 zeigt dieses Sichtfeld im Spiel angedeutet.

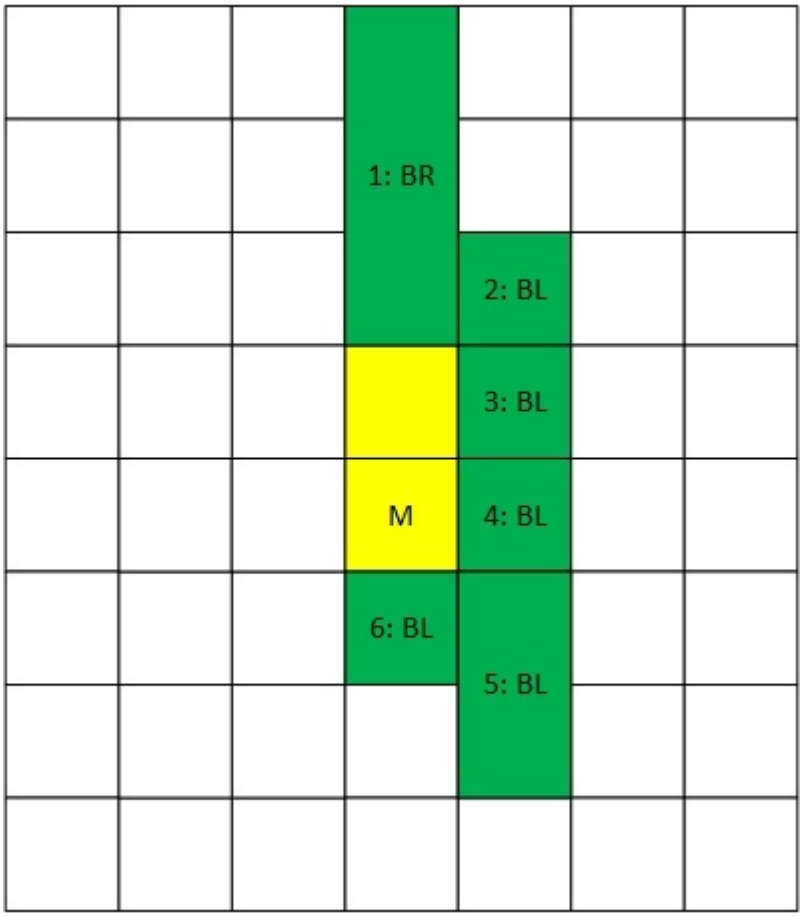

Abbildung 3.4: Mögliches Sichtfeld

Durch die Kombination von Sichtfeldern und der Generalisierung der Rückgabewerte von Blöcken ist es möglich, die Anzahl der Zustände beliebig zu verringern oder zu vergrößern. Der Aufbau eines Zustands ist wie folgt:

Marios Zustand | 'LevelScene' Blöcke/Zonen | 'Enemy' Blöcke/Zonen

Diese Kombination bildet eine Zahl, welche als Zustand verwendet wird.

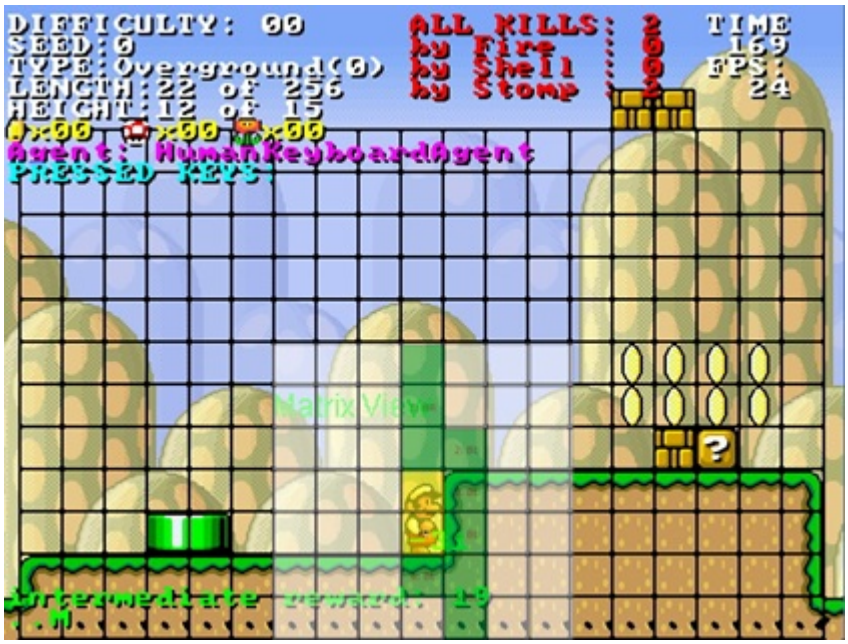

Abbildung 3.5: Sichtfeld im Spiel angedeutet

'Marios Zustand' ist der Zustand von Mario, wie er in Kapitel 2.2 beschrieben wird, auf Zahlen abgebildet.

- 1 = Klein
- 2 = Groß
- 3 = Feuermario
- 4 = Unverwundbar

3.3.3 Distanz zum Gegner

Da es in Mario-Spielen nötig ist, relativ genau auf Gegner springen zu können, um diese zu besiegen, wurde während des Projekts eine Möglichkeit eingebaut, die Distanz zum dichtesten Gegner in Pixeln zu erhalten, welche wesentlich genauer als die groben Blöcke des Gitternetzes ist. Hierfür wurde der Bereich um Mario in z.B. 5 verschieden große Sektoren eingeteilt wie Abbildung 3.6 zeigt. Diese Funktion meldet immer, in welchem Sektor sich der dichteste Gegner befindet. Befinden sich mehrere Feinde in verschiedenen Sektoren, wird trotzdem nur der Sektor angegeben, in dem sich der dichteste Gegner befindet und somit die größte Gefahr darstellt. Wie dies im Anwendungsfall aussieht und welchen Vorteil dies mit sich bringt,

verdeutlicht Abbildung 3.7. Die Funktion meldet, dass sich der dichteste Gegner in Sektor 3 befindet. Das rote Feld zeigt, in welchem Block der Feind durch das Gitternetz gemeldet wird. Der neue Aufbau des Zustands:

Gegnerdistanz Sektor | Marios Zustand | 'LevelScene' Blöcke/Zonen | 'Enemy' Blöcke/Zonen

Die Anzahl der Zustände wird hierdurch zwar verfünffacht, die erhöhte Anzahl an Siegen gleicht dies allerdings aus, wie Tests im Projekt zeigten.

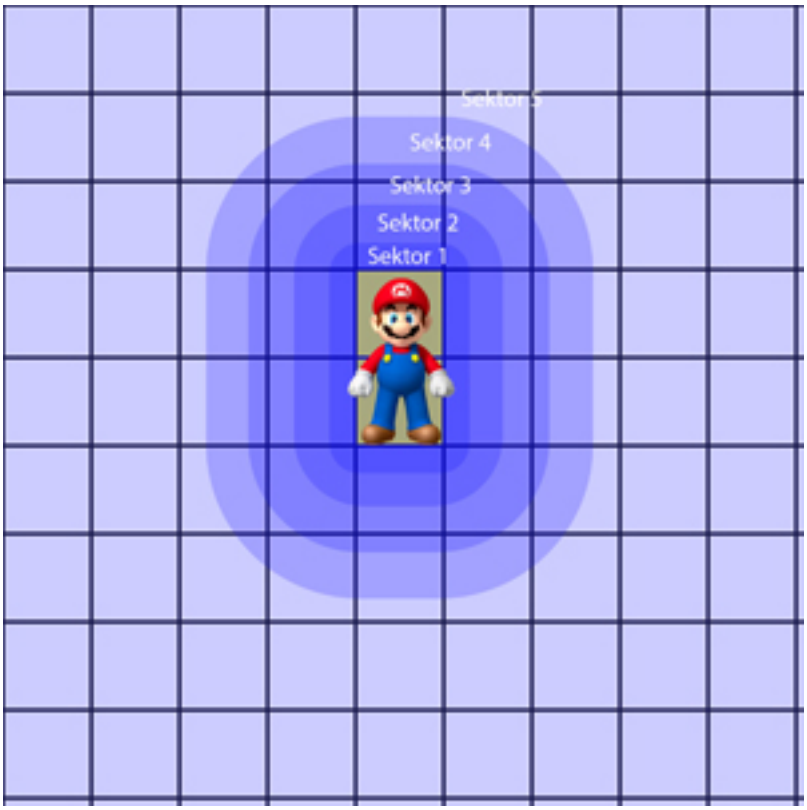

Abbildung 3.6: Gegnerdistanz Sektoren

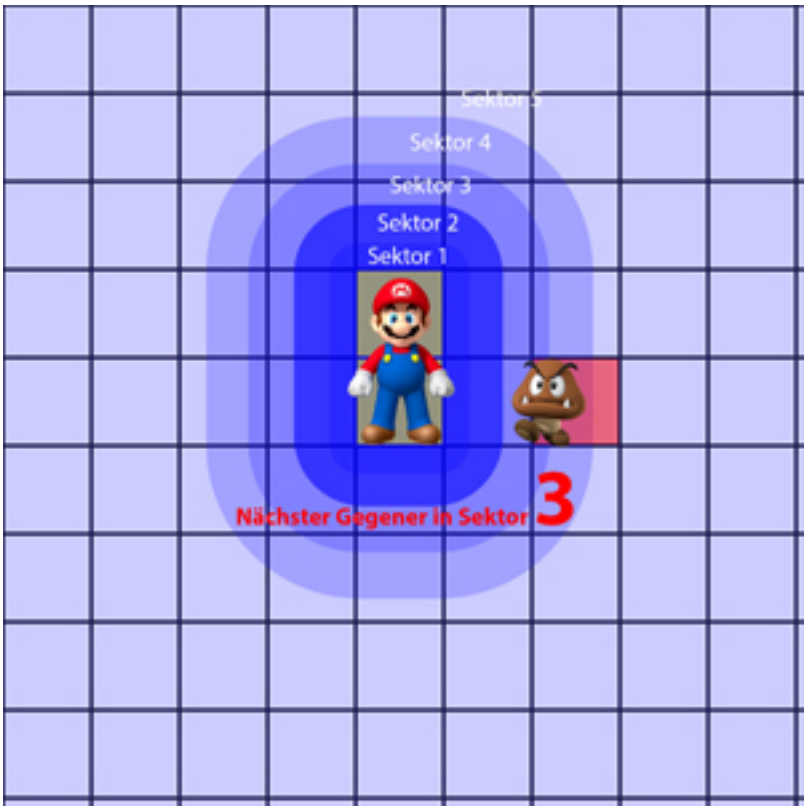

Abbildung 3.7: Gegnerdistanz Sektoren Beispiel mit Gegner

3.4 Sichtfeldvergleich

An dieser Stelle werden die Testergebnisse von drei verschiedenen Sichtfeldern vorgestellt und analysiert. Die Abbildungen 3.8, 3.9 und 3.10 zeigen die drei verwendeten Sichtfelder in der Eingabemaske des Programms.

Abbildung 3.8: Sichtfeld 1

Abbildung 3.9: Sichtfeld 2

Abbildung 3.10: Sichtfeld 3

Gleiche Zahlen in Feldern bedeuten hier, dass diese Felder zu einer Zone zusammengefasst werden. Jeweils 3 Agenten haben mit dem gleichen Sichtfeld in 10000 zufälligen Leveln gelernt zu spielen. Vergebene Belohnungen und Bestrafungen waren in allen Testläufen gleich. Die Ergebnisse wurden zu einem Mittelwert pro Sichtfeld zusammengefasst.

Abbildung 3.11 zeigt die erreichte Siegesrate in den 10000 Episoden der verschieden Sichtfelder, im Vergleich zueinander. In Abbildung 3.12 kann man die durchschnittliche Entwicklung der Siegesrate über die 10000 Episoden sehen.

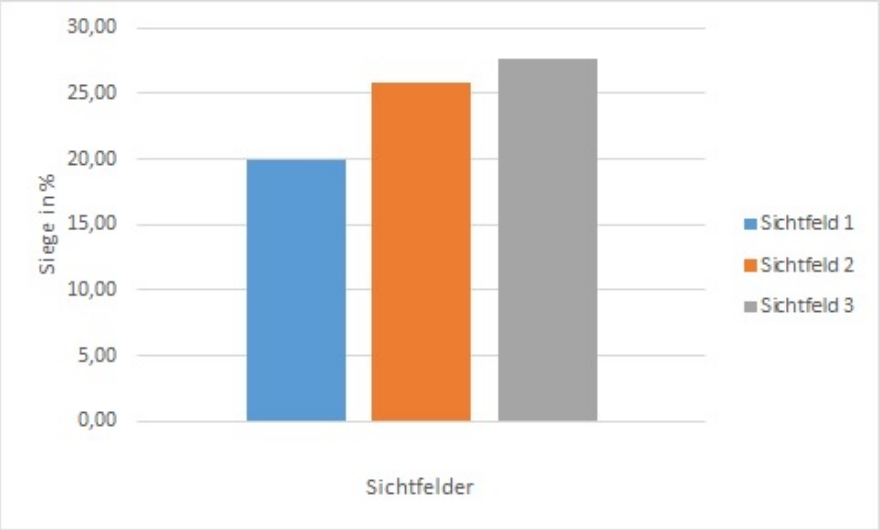

Abbildung 3.11: Durchschnitts-Siegesrate

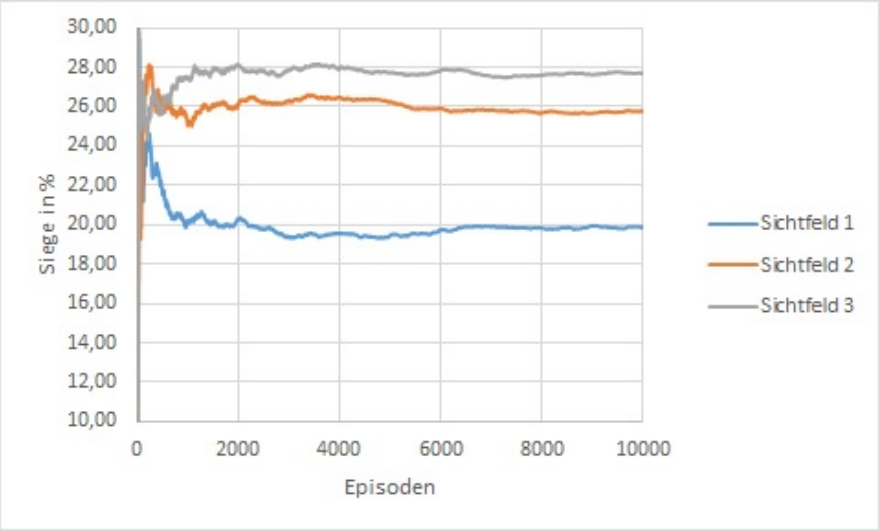

Abbildung 3.12: Entwicklung der Siegesrate

Wie zu sehen ist, werden durch größere Sichtfelder bessere Siegesraten erreicht. Dies ist unter anderem damit zu erklären, dass Situationen genauer erkannt und auch voneinander unterschieden werden können. Mit Sichtfeld 2 und 3 kann z.B. die Höhe eines Hindernisses besser erkannt werden als mit Sichtfeld 1. In Abbildung 3.5 ist so eine Situation gut zu erkennen.

Um noch bessere Ergebnisse zu erhalten, sollte man folglich das Sichtfeld vergrößern, da es auch mit Sichtfeld 3 immer noch Situationen gibt, die nicht erkannt werden können. Abbildung 3.13 zeigt eine Sackgasse, wie sie in einigen Leveln vorkommt. Um so eine Situation erkennen zu können, müsste das Sichtfeld zur Erkennung von Blöcken mindestens drei Reihen nach vorne reichen, wie zu sehen ist. Längere Sackgassen sind durchaus denkbar und ein noch größeres Sichtfeld wäre nötig. Dies führt zu verschiedenen Problemen. Im Projekt wurde der Zustand als 'Long'-Zahl repräsentiert. Sichtfeld 3 stellt die maximale Anzahl an Feldern dar, wie sie als 'Long' darzustellen ist. Mehr Felder im Sichtfeld können somit nicht mehr als 'Long'-Wert dargestellt werden. Mehr Blöcke bedeutet auch eine größere Zahl an Zuständen. Dies führt zu einem erhöhtem Bedarf an Episoden, die ein Agent benötigt, um ausreichend gut spielen zu lernen, sowie zu mehr benötigtem Speicher für die Datenbank, um alle Zustände und zugehörige Q-Werte zu speichern.

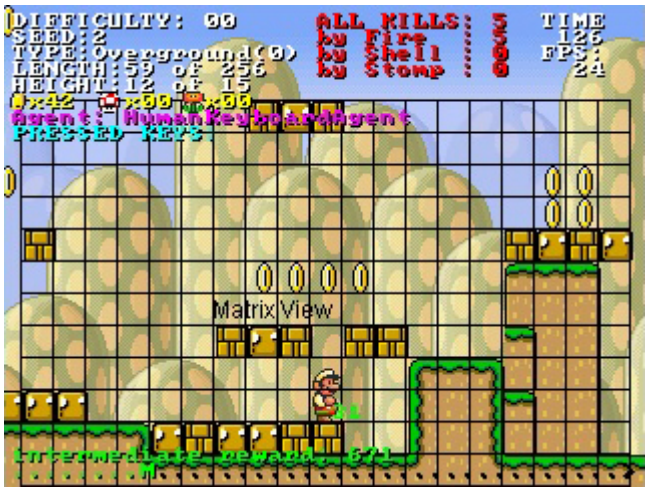

Abbildung 3.13: Sackgasse im Level

3.5 Verbesserungsmöglichkeiten

Im Rahmen des Projekts wurde ein Selbstversuch durchgeführt. Alle 4 Projektteilnehmer haben 100 zufällige Level in MarioAI gespielt. Das Ergebnis war eine Durchschnittssiegesrate von ca. 50%. Es stellt sich die Frage, ob die Siegesrate des Agenten von 28% noch zu steigern ist. Wünschenswert wäre ein Ergebnis näher an 50% oder sogar besser. Veränderungen an folgenden Bereichen führen zu einem verändertem Verhalten des Agenten.

- Aktionen
- Belohnungen/Bestrafungen
- Zustände/Sichtfeld

Verändern der möglichen Aktionen eines Agenten ist in diesem Fall nur möglich, indem Aktionen entfernt werden, da, wie im Kapitel 3.1 beschrieben, schon dem Agenten alle sinnvollen Aktionen zur Verfügung stehen und ein Hinzufügen somit nicht möglich ist. Das beste Resultat wäre zu erwarten durch das Entfernen der Möglichkeit, sich nach links im Level zu bewegen. Dies würde dazu führen, dass der Agent sich nur noch nach rechts in Richtung des Ziels bewegen und nicht mehr zurückgehen kann. Hierdurch wird das Problem einer Sackgasse wie in Abbildung 3.13 jedoch nur verschlimmert, da nun zu dem Problem des Erkennens auch noch jede Möglichkeit fehlt, aus dieser herauszukommen, sofern der Agent einmal in sie hineingegangen ist. Das Reduzieren von Aktionen kann Vorteile haben wie bereits beschrieben, diese sind allerdings immer mit Nachteilen verbunden, durch die der Agent teilweise stark eingeschränkt wird.Das Reduzieren ist somit keine gute Option, den Agenten zu verbessern.

Das Verändern der Belohnungen des Agenten stellt allerdings eine schnelle und einfache Möglichkeit dar, das Verhalten des Agenten zu verändern. Es sind durch das Projekt jedoch schon alle sinnvollen Belohungs- und Bestrafungsmöglichkeiten berücksichtigt worden und so bleibt nur noch das Verändern der Werte als Möglichkeit übrig. Wichtig bei Veränderungen der Belohnungen ist, dass nicht die tatsächliche Höhe, sondern das Verhältnis der Belohnungen und Bestrafungen zueinander ausschlaggebend ist für eine Veränderung im Verhalten des Agenten. Eine Änderung um z.B. Faktor 10 aller Belohnungen und Bestrafungen hat keinerlei Auswirkung auf das Verhalten des Agenten. Das perfekte Verhältnis der Belohnungen zu finden ist jedoch bei den hier vorhanden 9 Möglichkeiten sehr aufwendig. Die derzeit verwendeten Werte lieferten während des Projekts die besten Ergebnisse. Durch die Belohnungen und Bestrafungen erkennt der Agent zwar, was gute und was schlechte Aktionen sind, sie helfen ihm jedoch nicht dabei, sich besser durch ein Level zu bewegen. Eine Bestrafung für das Berühren

eines Gegners nützt dem Agenten nichts, wenn er den Gegner nicht erkennen kann. Eine große Verbesserung der Siegesrate ist mit dem Verändern von Belohnungen alleine nicht möglich.

Somit bleibt nur noch die Veränderung der Zustände bzw. das Verändern des Sichtfelds als Möglichkeit übrig. Es ist sinnvoll, das Sichtfeld des Agenten so zu vergrößern, dass der komplette Bereich, den auch ein menschlicher Spieler sehen kann, abgedeckt wird, um dem Agenten möglichst wenig Information vorzuenthalten, die auch ein Mensch hätte. Hierfür ist es nötig, das komplette Gitternetz zu nutzen. In der Breite bedarf dies keiner weiteren Erklärung, da das Gitternetz genauso breit ist wie der sichtbare Bereich für einen Menschen. Das komplette Gitternetz nach oben und unten zu verwenden sieht auf den ersten Blick aus, als ob dem Agenten mehr Information zur Verfügung gestellt wird als einem Spieler, welches unfair für einen Vergleich der Ergebnisse wäre. Abbildung 3.14 zeigt, wie sich das Gitternetz und der sichtbare

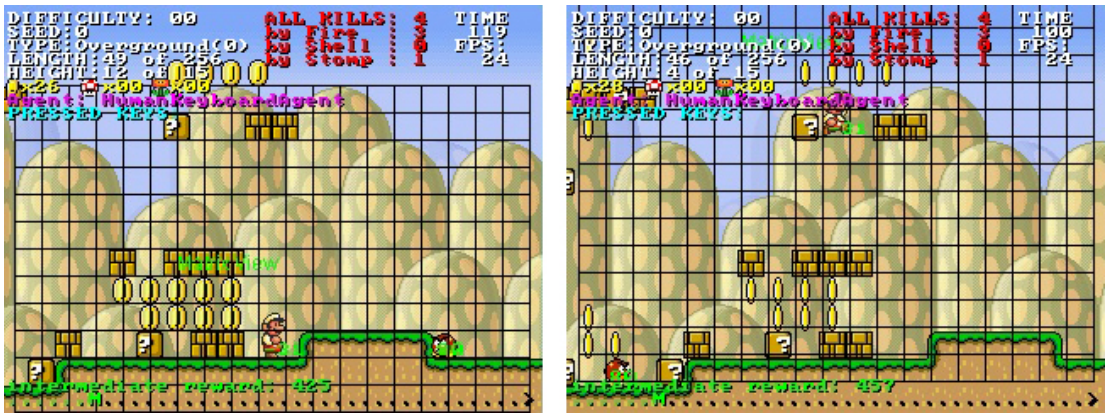

Abbildung 3.14: Gitternetz in verschiedenen Positionen von Mario

Bereich des Levels in verschieden Positionen von Mario verhalten. Wie man sehen kann, ist der sichtbare Bereich des Levels unabhängig von der Höhe, in der sich Mario befindet. Die Position von Mario im Gitternetz ist immer genau in der Mitte. Damit unabhängig von Marios Höhe immer der komplette sichtbare Bereich von Sichtfeld abgedeckt wird, muss das Gitternetz sowohl über als auch unter Mario verwendet werden. In diesem Fall ist das Sichtfeld größer als der für einen Spieler sichtbare Bereich. Theoretisch hat der Agent somit mehr Information als ein Spieler, in der Praxis ist dies jedoch zu vernachlässigen, da außerhalb des sichtbaren Bereichs nichts ist und sich somit keinerlei Vorteil gewinnen lässt. Blöcke vom Gitternetz, die außerhalb des Sichtbereichs liegen, liefern als Wert 0 zurück. Somit ist es möglich, das komplette Gitternetz für das Sichtfeld zu verwenden, ohne dass hierdurch ein Agent einen Vorteil gegenüber

einem menschlichen Spieler hat. Die genaue Umsetzung wird in Kapitel 4 beschrieben. Die Vergrößerung führt jedoch zu Problemen, wie schon in Kapitel 3.4 beschrieben. Das größte Problem hierbei ist, dass die Menge an möglichen Zuständen so groß wird, dass ein Agent viel länger braucht, bis ein zufriedenstellendes Lernergebnis vorliegt. Da der Agent in jedem Zustand 12 Aktionen zur Verfügung hat, muss dieser mindestens 12 mal durchlaufen werden, um alle Aktionen einmal probiert zu haben und die Beste zu finden. Vorhandenes Wissen wird dabei nicht auf neue Zustände angewandt. Bei jedem neuen Zustand beginnt der Agent somit mit dem willkürlichen Ausprobieren einer Aktion. Eine Vergrößerung des Sichtfelds führt dazu, dass schon eine kleine Veränderung im Level einen neuen Zustand darstellt. Durchläuft ein Agent mit einem großem Sichtfeld zufällige Level, ist zu erwarten, dass der Agent viele Zustände antrifft, allerdings die meisten nicht viel mehr als einmal und somit auch nur eine Aktion ausprobiert werden kann. Die Wahrscheinlichkeit, einen bisher unbekannten Zustand anzutreffen, ist viel größer, als auf einen schon bekannten zu treffen. Ein gutes Lernergebnis ist hier nicht zu erwarten und der Agent wird entsprechend erfolglos handeln. Um dieses Problem zu lösen, muss das bereits vorhandene Wissen auf neu entdeckte Zustände übertragen werden. Die Lösung hierfür ist die Verwendung eines Näherungsverfahrens (vgl. Sutton und Barto (1998), S.193).

3.6 Auswahl eines geeigneten Näherungsverfahrens

Alle Näherungsverfahren, die im Bereich des überwachten Lernens eingesetzt werden, können prinzipiell in Kombination mit Reinforcement Learning verwendet werden, wie z.B. künstliche neuronale Netze, Entscheidungsbäume und Multivariate Regression (vgl. Sutton und Barto (1998), S.195).

> „Gradient-decent methods are among the most widely used of all function approximation methods and are particulary well suited to reinforcement learning“(Sutton und Barto (1998), S.197).

Es stellt sich als nächstes die Frage, ob es sich bei der Zielfunktion, die angenähert werden soll, um eine lineare oder nichtlineare Funktion handelt. Damit eine lineare Funktion angenähert werden kann, darf es keine Abhängigkeiten zwischen einzelnen "Features"[3] eines Zustands geben. Dies bedeutet, dass die Bewertung, ob der Wert eines Features gut oder schlecht ist, nicht von dem Wert eines anderen Features abhängig sein darf. Im Falle von MarioAI ist dies nicht gegeben und es gibt viele Situationen, bei denen die Bewertung der Features voneinander

[3] Features sind die Einzelteile, aus denen sich ein Zustand zusammensetzt.

abhängig ist. Als Beispiel bietet sich die Sackgasse an wie in Abbildung 3.13. Ein Block, der sich über Mario befindet, ist kein Problem, da er einfach darunter durchlaufen kann. Ein Stapel Blöcke vor ihm sind auch kein Problem, da er hochspringen kann, um diese zu überwinden. Ist jedoch beides gegeben, bilden die Blöcke eine Sackgasse und stellen ein Problem dar. Um trotzdem eine lineare Funktion annäheren zu können, müssen diese Abhängigkeiten als zusätzliche Features modelliert werden (vgl. Sutton und Barto (1998), S.202). Da es sich in diesem Fall nicht nur um einen großen Zustandsraum handelt, sondern sich der Zustand selber aus sehr vielen Features zusammensetzt und dadurch selbst sehr groß ist, ist das weitere Vergrößern des Zustands keine gute Idee, zumal es sehr viele Möglichkeiten von Abhängigkeiten in MarioAI gibt. Alleine die Blöcke, die für eine Sackgasse verantwortlich sind, können an etlichen Positionen auftauchen. Eine Methode für die Annäherung an eine nichtlineare Funktion erspart hier Arbeit und benötigt keine weitere Modifikation des Zustands. Eine gute Methode hierfür stellen künstliche neuronale Netze dar.

> „Backpropagation methods for multilayer neural networks are methods for nonlinear gradient-descent function approximation“(Sutton und Barto (1998), S.222).

Eine Kombination von TD-learning und einem neuronalen Netz wurde schon beim bekannten 'TD-Gammon'[4] sehr erfolgreich eingesetzt. Deshalb fällt die Wahl des Näherungsverfahrens für diese Anwendung auch auf ein neuronales Netz.

[4] TD-Gammon ist ein Computerprogramm, welches auf Weltmeisterniveau das Brettspiel Backgammon spielt.

4 Entwicklung

Dieses Kapitel beschreibt die Umsetzung der benötigten Veränderungen zur Vergrößerung des Sichtfelds sowie die Implementierung eines neuronalen Netzes in Verbindung mit einem RL-Agenten.

4.1 Vergrößerung des Sichtfelds

Wie in Kapitel 3.5 beschrieben soll nun das Sichtfeld des Agenten auf das gesamte Gitternetz vergrößert werden, um dem Agenten möglichst wenig Information vorzuenthalten gegenüber einem menschlichen Spieler. Nach einer Anpassung der GUI[1] ist es möglich, das komplette Gitternetz für das Sichtfeld zu verwenden. Abbildung 4.1 zeigt die veränderte GUI sowie das neue Sichtfeld für den Agenten. Wie zu sehen ist, werden die Felder aus dem Bereich 'Bridge' und 'Enemy' nicht verwendet.

Der Bereich 'Enemy' dient dazu, eine Entfernungsunschärfe zu erzeugen, wie in Kapitel 3.3.1 beschrieben. Dies wird jetzt nicht mehr benötigt, da sowieso der ganze Bereich abgeprüft wird und für einen menschlichen Spieler keine Entfernungsunschärfe existiert.

Auf Brücken muss nach wie vor geprüft werden. Es ist jedoch mathematisch sinnvoller, Felder aus dem Bereich 'Block' auch auf Brücken prüfen zu lassen und keine weiteren Felder aus dem Bereich 'Bridge' zu verwenden, als jedes Feld im 'Block' sowie 'Bridge' Bereich zu überprüfen. Dies würde sowohl die Anzahl als auch die Länge von Zuständen unnötig erhöhen. Die folgende Rechnung zeigt den Unterschied in der Anzahl der Zustände ohne den 'DetailedEnemy' Bereich.

[1] GUI: graphical user interface / grafische Benutzeroberfläche

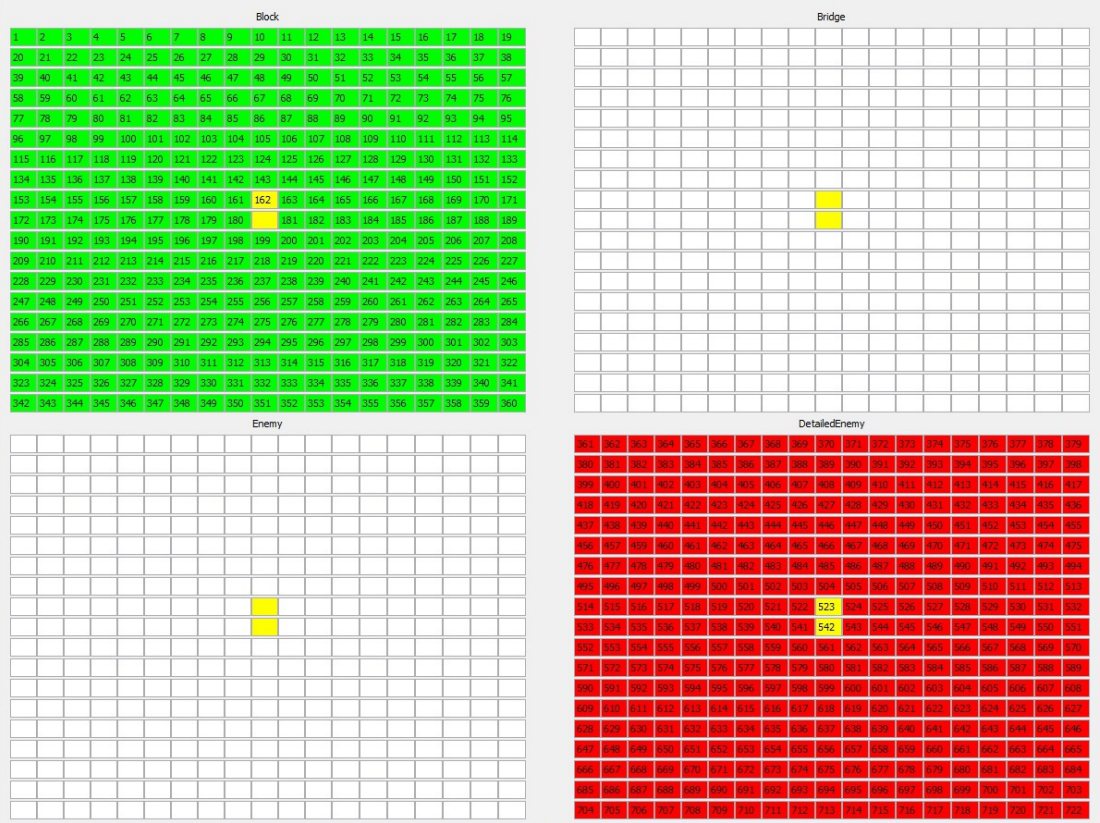

Abbildung 4.1: Maximales Sichtfeld

$$Block^{360} \cdot Bridge^{360} > BlockNeu^{360}$$
$$2^{360} \cdot 2^{360} > 3^{360}$$
$$5,515652 \cdot 10^{216} > 5,802988 \cdot 10^{171}$$

Felder des ’Block’ Bereichs haben hierdurch folgende Rückgabewerte:

- 0 = Nichts
- 1 = Block
- 2 = Bridge

Das mittlere Feld im ’Block’ Bereich wird nicht überprüft, da dieses Feld immer von Mario selber belegt wird und an dieser Stelle sich kein Block befinden kann. Mittels einer weiteren Veränderung der Rückgabewerte des ’Block’ Bereichs können die ’DetailedEnemy’ Felder

ebenso integriert werden, um einen Zustand trotz maximalem Sichtfeld so klein wie möglich zu halten. Somit ergeben sich 361 Felder im 'Block' Bereich, die überprüft werden müssen, um einen Zustand zu erzeugen. Die neuen Rückgabewerte sind folgende:

- 0 = Nichts
- 1 = Block
- 2 = Bridge
- 3 = Gegner
- 4 = Gegner (durch Daraufspringen nicht zerstörbar)
- 5 = Prinzessin

Es ergibt sich folgende Gesamtanzahl an Zuständen:

$$Gegnerdistanz \cdot MarioZustand \cdot Block^{361} = 9 \cdot 4 \cdot 6^{361} = 2,94377 \cdot 10^{282}$$

4.2 Veränderung der Zustandsimplementierung

Bisher wurden Zustände als Long-Wert dargestellt. Aufgrund der nun hohen Anzahl an Ziffern, die benötigt werden um den Zustand darzustellen, ist eine reine Darstellung als Zahl nicht mehr möglich.

$$DistanzZumGegner + MariosZustand + BlockFelder = 1 + 1 + 361 = 363$$

Ein Zustand besteht nun, wie zu sehen ist, aus 363 Ziffern. Um die Anzahl verwalten zu können, wird der Zustand intern als 'ArrayList<Integer>' repräsentiert. Dies hat den Vorteil, in der Länge flexibel zu sein und sich einem anderen Sichtfeld anpassen zu können, sowie bei Bedarf auf einzelne Werte direkt zugreifen zu können.

4.3 Veränderung der Datenbank

Um die Arrays in der Datenbank speichern zu können, wird die Klasse 'ASConverter' implementiert, um einen Array in einen String mit ';' als Trennungszeichen zwischen den Werten umzuwandeln. Die Klasse bietet zudem die Möglichkeit, einen String in einen Array umzuwandeln, wenn dieser z.B. aus der Datenbank gelesen wird.

Die erzeugten Strings haben eine Zeichenlänge von:

$$Ziffern + Trennzeichen = 363 + 362 = 725 Zeichen$$

Strings haben zudem den Vorteil, unabhängig von der verwendeten DB zu sein. Durch die große Menge an Zuständen können diese nicht mehr wie bisher im Zwischenspeicher gehalten werden, sondern müssten am einfachsten einzeln aus der Datenbank abgerufen sowie gespeichert werden. Dies hat zur Folge, dass die Laufzeit pro Episode erheblich ansteigt.

Da lediglich die Zustände auf die Q-Werte abgebildet werden, reichen die Funktionalitäten einer Key-Value-Datenbank vollkommen aus. Aufgrund schon vorhandener Erfahrung, Einfachheit in der Implementierung sowie schnellerer Verarbeitung von Anfragen (vgl. DBBenchmarks) wird Redis als neue Datenbank für Zustände verwendet. Mit Hilfe der 'ASConverter' Klasse können auch die Q-Werte in einen String umgewandelt werden, sodass in Redis lediglich Strings auf Strings abgebildet werden müssen. Durch die Implementierung von Redis können nun die langen Strings der Zustände problemlos verwaltet werden.

4.4 Wahl eines Frameworks für neuronale Netze

Bei der Wahl eines Frameworks für neuronale Netze gibt es einige Auswahlmöglichkeiten wie z.B. 'Joone', 'Neuroph' und 'Encog'. Alle Frameworks haben ihre Vor- und Nachteile. In der Bachelorarbeit von Frau Ott wurden diese drei miteinander verglichen. Das Ergebnis ist, dass das Encog Framework empfohlen wird, deshalb fällt auch hier die Wahl auf Encog (vgl. Ott (2013) S.65).

4.5 Erzeugung trainierter neuronaler Netze

Um von den Daten in der Datenbank zu einem trainierten neuronalem Netzwerk zu gelangen, sind einige Zwischenschritte nötig, um dies zu erreichen. Die Abbildung 4.2 zeigt die Schritte, die benötigt werden, um ein trainiertes Netz zu erzeugen. In den folgenden Abschnitten werden die einzelnen Schritte näher betrachtet und erläutert.

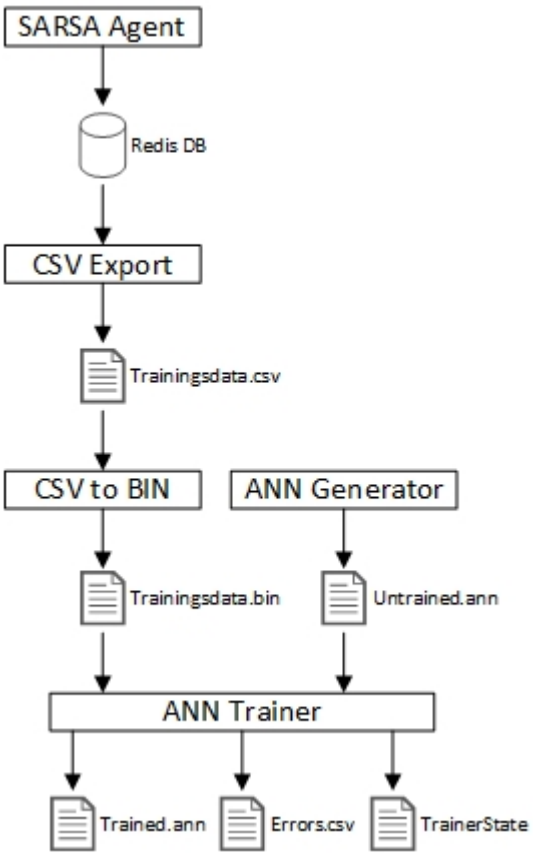

Abbildung 4.2: Benötigte Schritte zum trainierten neuronalem Netz

4.5.1 Erzeugung von CSV Dateien

Um die Datensätze aus der Datenbank auszulesen und als CSV Datei abzuspeichern wird die Klasse 'CSVExport' implementiert. Diese kann mittels der implementierten 'CSVGenerator' Klasse die Redis Datenbank mit dem 'SCAN'-Befehl iterativ auslesen. Die ausgelesenen Werte werden in einer erzeugten CSV Datei abgespeichert. Die Dateien können später als Trainings- und Validierungsdaten für die neuronalen Netze verwendet werden. Zusätzlich wird eine 'CSVSplitter' Klasse implementiert, welche eine CSV Datei in gleich große Teile aufteilen und abspeichern kann. Die Größe, bei der aufgeteilt werden soll, ist anpassbar und kann angegeben werden.

4.5.2 Vorbereiten von CSV Dateien

Da die erzeugten CSV Dateien sehr groß sein können, können diese nicht komplett als Trainingsdatensatz von Encog in den Speicher geladen werden. Encog bietet hierfür ein sogenanntes 'BufferedMLDataSet' an, welches aus einer BIN Datei erzeugt werden kann. Zusätzlich wird eine Methode angeboten, um CSV Datein in das benötigte Format zu konvertieren. Hierfür wird die Klasse 'CSVtoBIN' implementiert. Diese konvertiert mit Hilfe von Encog eine gegebene CSV Datei in eine BIN Datei. Außer dem Dateinamen muss noch die Anzahl an Eingangs- sowie Ausgangsneuronen angegeben werden. Die BIN Datei ermöglicht es Encog, auch mit großen Datenmengen zu arbeiten und nur gerade benötigte Daten in den Speicher zu laden.

4.5.3 Erstellen von neuronalen Netzen

Zur Erzeugung von neuronalen Netzen wird die Klasse 'ANNGenerator' implementiert. Diese kann ein sogenanntes 'BasicNetwork' mittels 'Encog' erstellen und als Datei abspeichern für einen späteren Gebrauch. Es kann angegeben werden, wie viele Eingangs-, Ausgangs- und Hiddenneuronen verwendet werden sollen, sowie die Aktivierungsfunktion dieser und Anzahl an Schichten von Hiddenneuronen. Nach der Erstellung wird das Netz mit zufälligen Verbindungsgewichten initialisiert und anschließend abgespeichert. Für spätere Tests können so einfach und schnell verschiedene Netze erstellt werden.

4.5.4 Trainig von neuronalen Netzen

Zum Trainieren der erzeugten neuronalen Netze wird die Klasse 'ANNTrainer' implementiert. Diese benötigt zwei Dateien:

- ANN Datei
- Trainingsdatensatz

Die ANN Datei ist das neuronale Netz, welches trainiert werden soll. Hier wird ein Netz angegeben, das mittels der 'ANNGenerator' Klasse erstellt worden ist. Beim Trainingsdatensatz handelt es sich, wie der Name vermuten lässt, um die Datei, in der die Datensätze für das Training gespeichert sind. In diesem Fall handelt es sich hierbei um die erzeugte BIN Datei, die mittels der 'CSVtoBIN' Klasse erstellt worden ist. Der aktuelle Zustand des Trainers am Ende vom Training wird in einer zusätzlichen Datei gespeichert, um dieses bei Bedarf fortsetzen zu können (TrainerState). Für eine spätere Auswertung werden die ermittelten Trainingsfehler in der Error Datei im CSV Format gespeichert. Nach dem Laden des Netzes sowie der Trainingsdaten aus den Dateien muss ein Trainer- Objekt erzeugt werden. Dieser dient dazu, das neuronale

Netz mit den Trainingsdaten zu trainieren. Als Trainingsalgorithmus wird 'Resilient Propagation' verwendet, dieser wird unter anderem vom Chefentwickler des Encog- Frameworks als effizientester Algorithmus empfohlen (vgl. Heaton (2011) S.74). Als Alternative kann der 'Backpropagation- Algorithmus' verwendet werden. Dieser benötigt zusätzlich die Angaben von Lernrate und sogenanntem Momentum. Werden diese Werte schlecht gewählt, kann nicht garantiert werden, dass das Netz optimal trainiert wird. Es müssten für jeden Anwendungsfall neue optimale Werte gefunden werden. Das aber bedeutet einen hohen Aufwand und ist deshalb nicht zu empfehlen (vgl. Ott (2013) S.77). Der Trainer kann nun Trainingsiterationen durchführen. Bei einer Iteration wird der gesamte Trainingsdatensatz einmal durchlaufen und anschließend das Netz zur Fehlerreduzierung optimiert. Es gibt zwei Möglichkeiten, wie häufig eine Iteration durchgeführt werden soll. Entweder wird eine festgelegte Anzahl an Iterationen durchlaufen oder es wird so lange iteriert, bis der Trainingsfehler unter einen angegebenen Wert fällt. In diesem Fall könnte es zu einer Endlosschleife kommen, wenn das Netz nicht unterhalb des geforderten Fehlers trainiert werden kann. Deshalb wird hier die erste Möglichkeit verwendet. Zudem bietet sie bessere Vergleichsmöglichkeiten zwischen verschiedenen Netzen, da alle Netze so gleich lange trainiert werden. Nach dem das Training beendet ist, werden das trainierte Netz, der aktuelle Zustand des Trainings sowie die Fehlerdaten abgespeichert.

4.6 Verwendung eines neuronalen Netzwerks mit einem RL-Agenten

Da das Ziel bei der Verwendung eines neuronalen Netzes unter anderem ist, die Datenbank ganz oder teilweise abzulösen, ist es wünschenswert, dass das Netzwerk ähnlich wie die Datenbank funktioniert. Um dies zu erreichen, muss das neuronale Netz als Eingabe einen Zustand annehmen und als Ausgabe die Q- Werte hierzu liefern. Hierfür benötigt das Netz genauso viele Eingangsneuronen, wie ein Zustand sogenannte Features hat. Ein Feature ist z.B. ein Wert von einem Block im Sichtfeld oder auch der Zustand von Mario. Das bedeutet, ein Netzwerk benötigt 363 Eingangsneuronen. Da es 12 Aktionen gibt, werden auch 12 Q- Werte benötigt. Somit ergibt sich, dass 12 Ausgangsneuronen benötigt werden. So kann die Funktion der Datenbank vom neuronalen Netz übernommen werden. Es gibt nun drei Möglichkeiten, ein Netz mit dem RL- Agenten zu verwenden.

Die einfachste wäre, die Datenbank komplett durch das Netz zu ersetzen. Hierbei werden alle Anfragen, die sonst an die Datenbank gehen, an das neuronale Netzwerk gestellt. Updates werden nicht durchgeführt. Dies bedeutet, dass der Agent kein neues Wissen mehr lernt und

die Strategie des Agenten sich nicht mehr verändert. Der Nachteil dieser Lösung ist sehr offensichtlich. Der Vorteil dieser Variante ist, dass keine Datenbank verwendet wird und somit der benötigte Speicher sich nicht vergrößert, da das Netz seine Größe beibehält. Zusätzlich wird die Generalisierung für neue unbekannte Zustände vom Netz sich zunutze gemacht. Dieser Ansatz verspricht wenig Erfolg und ist nur sehr begrenzt zu empfehlen, wenn z.B. nur beschränkter Speicherplatz vorhanden ist oder die gelernte Strategie des Agenten aus den Trainingsdaten ausreicht und nur eine gute Generalisierung benötigt wird.

Eine wünschenswertere Lösung ist, die Datenbank komplett zu ersetzen wie in der ersten Variante, jedoch trotzdem noch dem Agenten die Möglichkeit zu geben, etwas dazulernen zu können. Hierfür muss eine Trainingsiteration für das Netzwerk durchgeführt werden mit den neuen Q- Werten für einen Zustand. Dies klingt nach der optimalen Lösung, da nur ein fester Speicherplatz benötigt wird und keine an Größe zunehmende Datenbank verwendet wird. Außerdem wird die Generalisierung des Netzwerks genutzt mit der Möglichkeit, dass der Agent noch Wissen dazulernen kann und die Strategie des Agenten sich anpassen kann. Bei Tests mit dieser Variante zeigte sich, dass das neuronale Netz durch jede weitere Trainingsiteration jedoch schlechter wurde und dem Agenten immer unbrauchbarere Werte lieferte. Hierbei kommt es zu einer sogenannten Interferenz im neuronalen Netz. Diese entsteht durch die häufige Anpassung der Verbindungsgewichte im Netzwerk (vgl. Shiraga u. a. S.1). Somit ist diese Lösung nicht verwendbar mit den hier zum Einsatz kommenden neuronalen Netzen.

Als letzte Möglichkeit bietet sich eine hybride Lösung an. Hierfür wird ein neuronales Netz in Kombination mit einer Datenbank verwendet. Anfragen für Q- Werte werden an die Datenbank gestellt, wenn diese keinen Eintrag zu dem fraglichen Zustand beinhaltet wird das Netzwerk befragt. Neues Wissen wird in der Datenbank gespeichert und steht für eine erneute Anfrage zur Verfügung. Der Vorteil ist hier, dass zusätzlich zur Generalisierung des neuronalen Netzes auch die Möglichkeit besteht, dass neu gelerntes Wissen des Agenten gespeichert und genutzt werden kann. Die Nachteile sind, dass durch die Verwendung einer Datenbank der benötigte Speicherplatz mit der Zeit ansteigt und die Strategie, die das Netz generalisiert, nicht verändert wird, da kein weiteres Training stattfindet. Da diese Lösung bessere Ergebnisse verspricht gegenüber der ersten Möglichkeit, wird diese implementiert und für die Testläufe im folgenden Kapitel verwendet.

5 Auswertung

In diesem Kapitel werden die durchgeführten Versuche beschrieben und deren Ergebnisse ausgewertet.

5.1 Generierung von Trainingsdaten

Zur Erzeugung von Trainingsdaten hat ein SARSA- Agent 10000 mal das erste Level von MarioAI durchlaufen. Das Durchlaufen von 10000 zufälligen Leveln scheint nicht sinnvoll, da wie schon im Kapitel 3.5 beschrieben, hierbei nur sehr wenige Aktionen pro Zustand erkundet werden und die entstehenden Daten keine Strategie erkennen lassen würden, von der ein neuronales Netz sinnvoll generalisieren könnte. Abbildung 5.1 zeigt die Siegesrate des Agenten bei den jeweiligen 10000 Durchläufen.

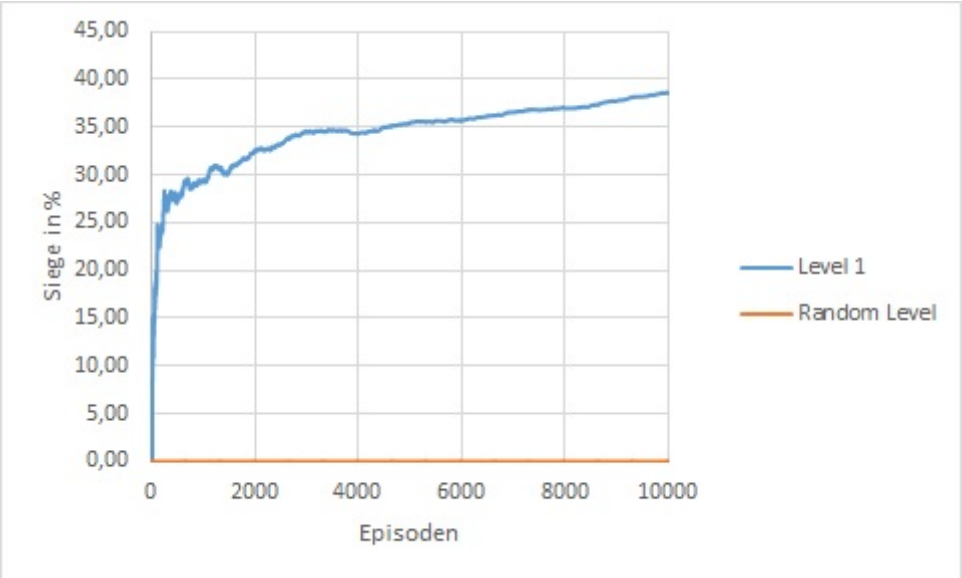

Abbildung 5.1: Siegesrate bei neuem Sichtfeld

Bisherige Annahmen werden hierdurch nochmals bestätigt. Eine Vergrößerung des Sichtfeldes führt zu einer höheren Siegesrate, die Anzahl an Zuständen ist jedoch so hoch, dass in zufälligen Leveln keine Siege vom Agenten erzielt werden konnten. Zudem zeigt die Kurve der Durchläufe in Level 1 einen deutlichen Trend nach oben, welches bedeutet, dass der Agent auch nach 10000 Durchläufen durch das selbe Level immer noch dazulernt und seine Strategie verbessert.

5.2 Aufbau des neuronalen Netzes

Wie viele Eingangs- und Ausgangsneuronen ein neuronales Netz haben muss, steht fest auf Grund der benötigten Daten, die das Netz verarbeiten soll (363 Inputneuronen und 12 Outputneuronen). Ebenso ist die Aktivierungsfunktion dieser Neuronen schnell gewählt, ohne ein große Auswahl zu haben. Sowohl für die Eingangs- als auch für die Ausgangsneuronen werden Werte benötigt, die außerhalb des Intervalls von [-1;1] liegen. Deshalb wird für beide Schichten eine lineare Aktivierungsfunktion gewählt. Da der Wertebereich der einzelnen Features von einem Zustand überwiegend identisch sind, wird von einer Normalisierung der Eingangswerte abgesehen. Da es für die Anzahl von zuverwendenen Hiddenneuronen und Schichten von Hiddenneuronen keine Formel gibt, mit der man eine optimale Anzahl berechnen könnte, muss sich auf vorhandene Faustformeln verlassen werden. Für mögliche neuronale Netze werden folgende Formeln verwendet, wobei n die Anzahl an Inputneuronen darstellt und m die Ausgangsneuronen.

- $\sqrt[2]{n \cdot m}$ (im folgenden SQRT genannt)
- $n \cdot 0,75$ (im folgenden 75n genannt)
- $n \cdot 2$ (im folgenden 2n genannt)

Diese Formeln wurden ebenfalls in der Bachelorabreit von Frau Ott verwendet, um ein geeignetes Netzwerk zu finden. Ebenso wird die Empfehlung berücksichtigt, so wenig Hiddenschichten wie möglich einzusetzen (vgl. Ott (2013) S.76).

5.3 Trainieren der neuronalen Netze

Für das Trainieren und Auswerten von neuronalen Netzen wurde immer der gleiche Datensatz aus den 10000 Durchläufen des ersten Levels verwendet. 70% der Daten wurden als Trainingsdaten verwendet und 30% wurden den Netzen vorenthalten, um sie zu validieren und hierdurch die Generalisierung der Netze zu überprüfen.

5.3.1 Erste Netz- Iteration

Es wurden drei Netze für den Anfang erstellt, jeweils eins für eine der drei Faustformeln. Es wurde eine Schicht mit Hiddenneuronen Verwendet. Als Aktivierungsfunktion für die Hiddenneuronen wurde die Tangens Hyperbolicus Funktion verwendet, da es sich hierbei um eine nicht lineare Funktion handelt die einen positiven als auch negativen Wertebereich besitzt. Diese erscheint sinnvoll, da in Kapitel 3.6 festgestellt wurde, dass die finale Funktion nicht linear sein wird und die Q- Werte sowohl positiv als auch negativ sein können. Alle Netze haben 200 Trainingsiterationen durchlaufen. Nach jeder Iteration wurde zudem eine Validierung des Netzes durchgeführt.

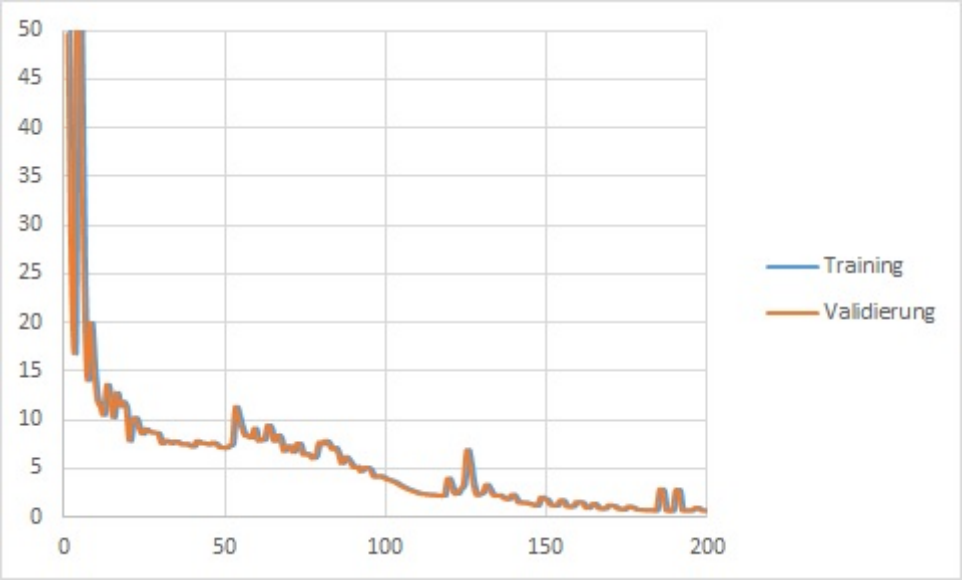

Abbildung 5.2: SQRT mit Tangens Hyperbolicus Aktivierungsfunktion

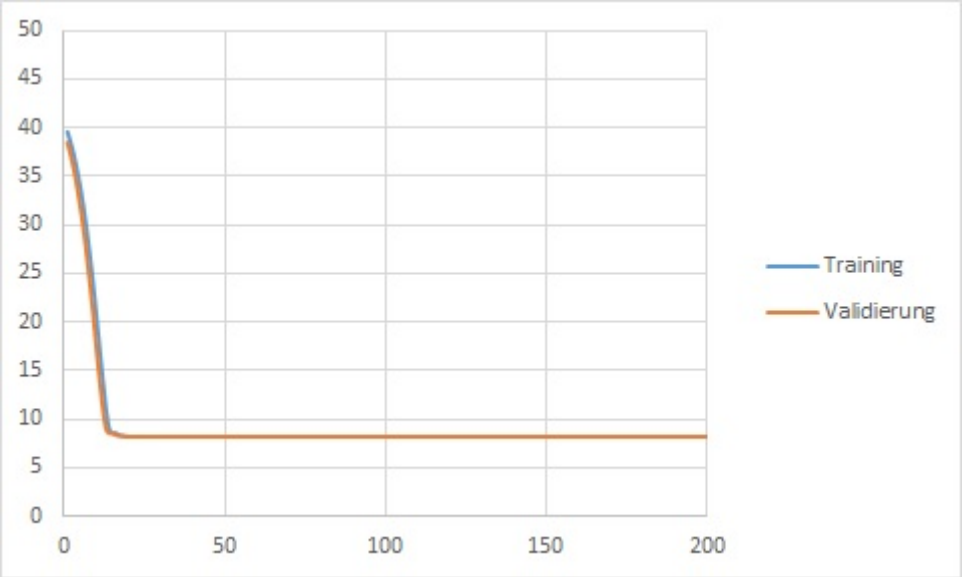

Abbildung 5.3: 75n mit Tangens Hyperbolicus Aktivierungsfunktion

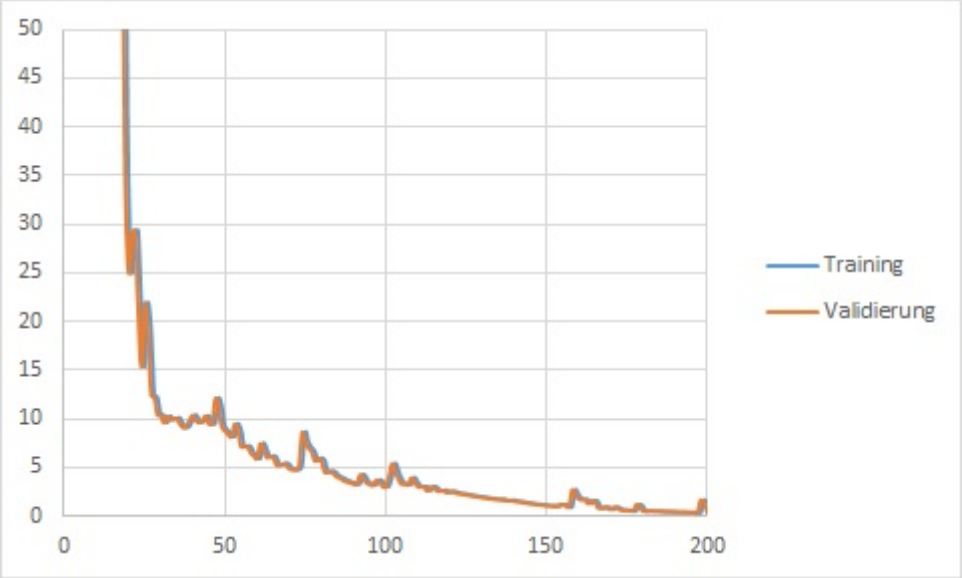

Abbildung 5.4: 2n mit Tangens Hyperbolicus Aktivierungsfunktion

Die Abbildungen 5.2, 5.3 und 5.4 zeigen den Verlauf des Trainings- und Validierungsfehlers der drei Netze. Wie zu erkennen ist, konnte das Netz mit den ’75n’ Hiddenneuronen keinen zufriedenstellenden Trainingsfehler erreichen. Die beiden anderen Netze wurden daraufhin

verwendet um jeweils 10000 zufällige Level zu durchlaufen. Der Agent konnte jedoch bei beiden Netzen keine Siege erzielen. Die Tabelle 5.1 Zeigt die erreichten Trainings- sowie Validierungsfehler der Netze.

Netz	Trainingsfehler	Validierungsfehler
SQRT	0,691	0,687
75n	8,169	8,162
2n	1,445	0,408

Tabelle 5.1: Fehler 1. Netz- Iteration

5.3.2 Zweite Netz- Iteration

Da mit den ersten drei Netzen keine Erfolge erzielt werden konnten, wurden zum Testen der Auswirkungen in jedem Netzwerk eine zweite Schicht mit Hiddenneuronen hinzugefügt. Beide Schichten haben die gleiche Anzahl an Neuronen.

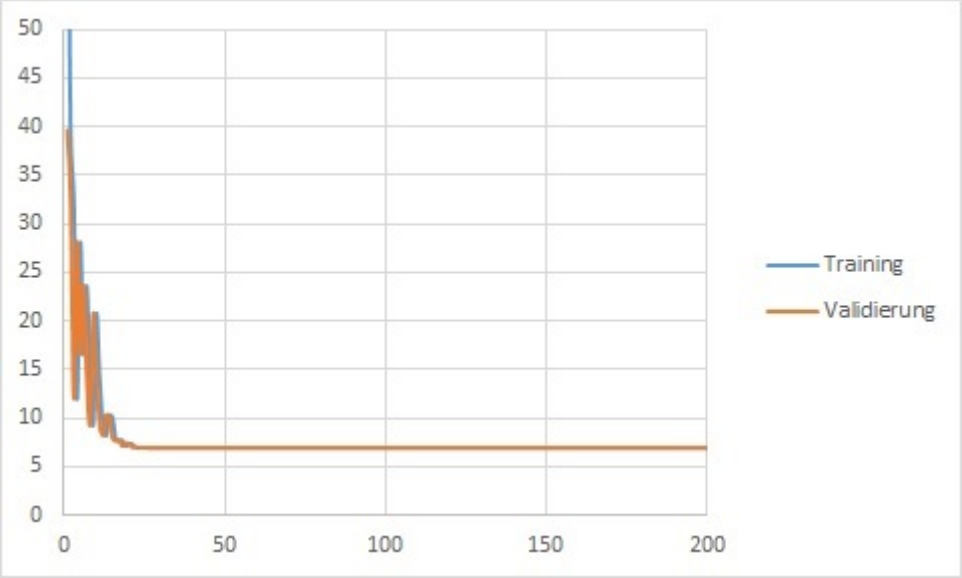

Abbildung 5.5: Zwei mal SQRT mit Tangens Hyperbolicus Aktivierungsfunktion

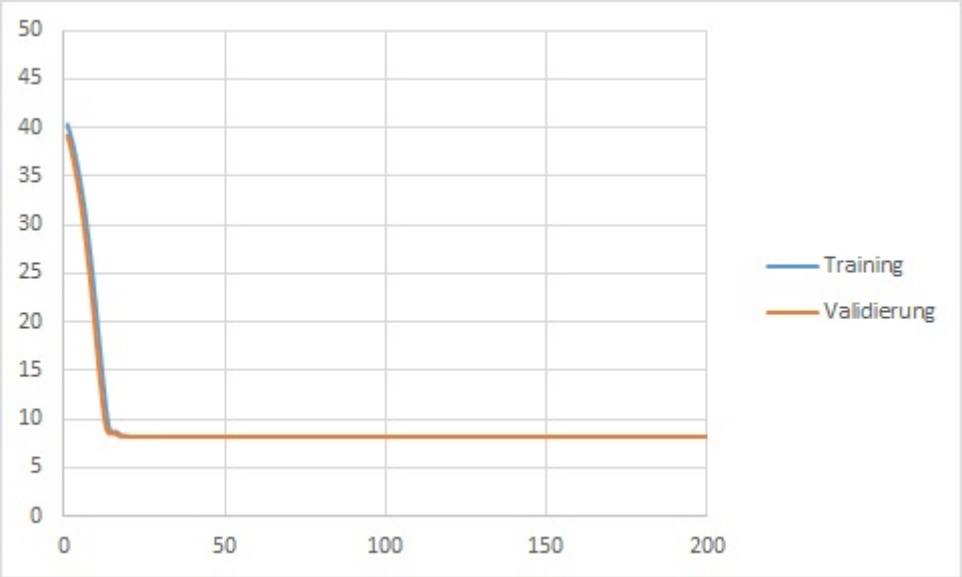

Abbildung 5.6: Zwei mal 75n mit Tangens Hyperbolicus Aktivierungsfunktion

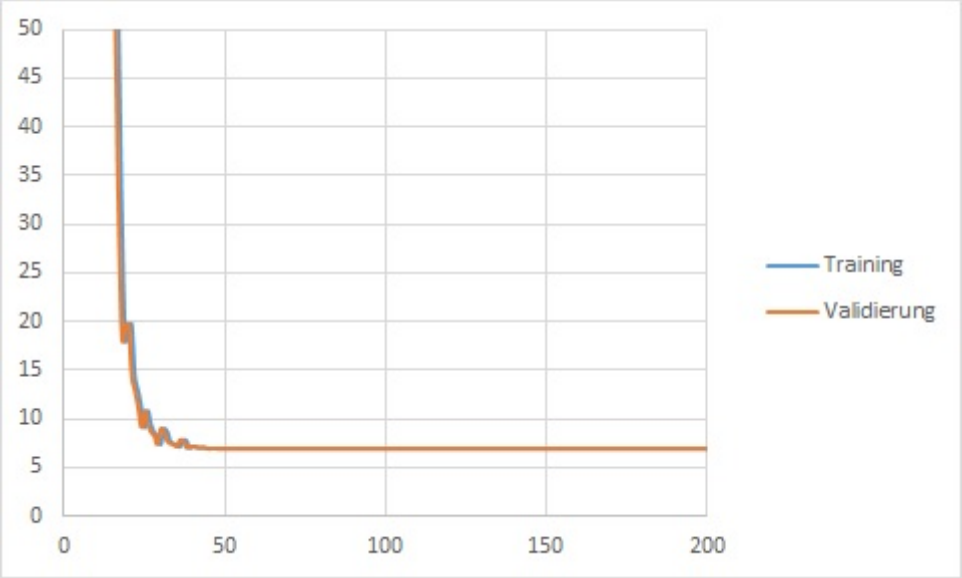

Abbildung 5.7: Zwei mal 2n mit Tangens Hyperbolicus Aktivierungsfunktion

Wie die Abbildungen 5.5, 5.6 und 5.7 zeigen konnte kein Netzwerk einen ausreichend kleinen Trainingsfehler erreichen und sie sind somit alle unbrauchbar für einen tatsächlichen Einsatz mit einem Agenten. Die Tabelle 5.2 zeigt die erreichten Fehlerwerte.

Netz	Trainingsfehler	Validierungsfehler
SQRT	6,902	6,93
75n	8,169	8,162
2n	6,902	6,93

Tabelle 5.2: Fehler 2. Netz- Iteration

5.3.3 Dritte Netz- Iteration

Eine zweite Schicht Hiddenneuronen hat die Ergebnisse verschlechtert, somit hat sich gezeigt, das hier eine Schicht die bessere Lösung darstellt. Als weitere Veränderungsmöglichkeit bietet sich die Aktivierungsfunktion der Hiddenneuronen an. Im den folgenden Netzen wurde eine lineare Aktivierungsfunktion für die Hiddenneuronen benutzt, wie sie auch für die Eingangs- und Ausgangsneuronen verwendet wird.

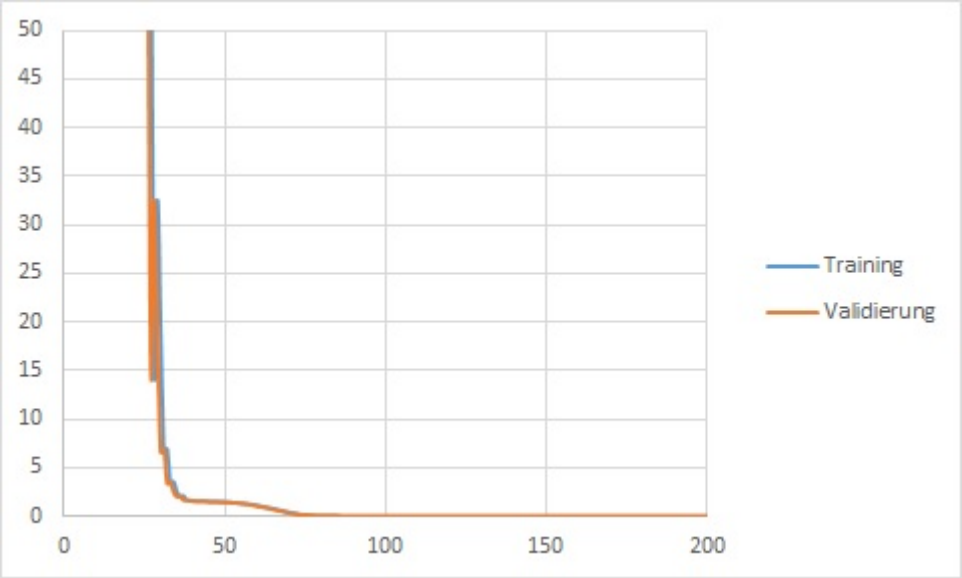

Abbildung 5.8: SQRT mit linearer Aktivierungsfunktion

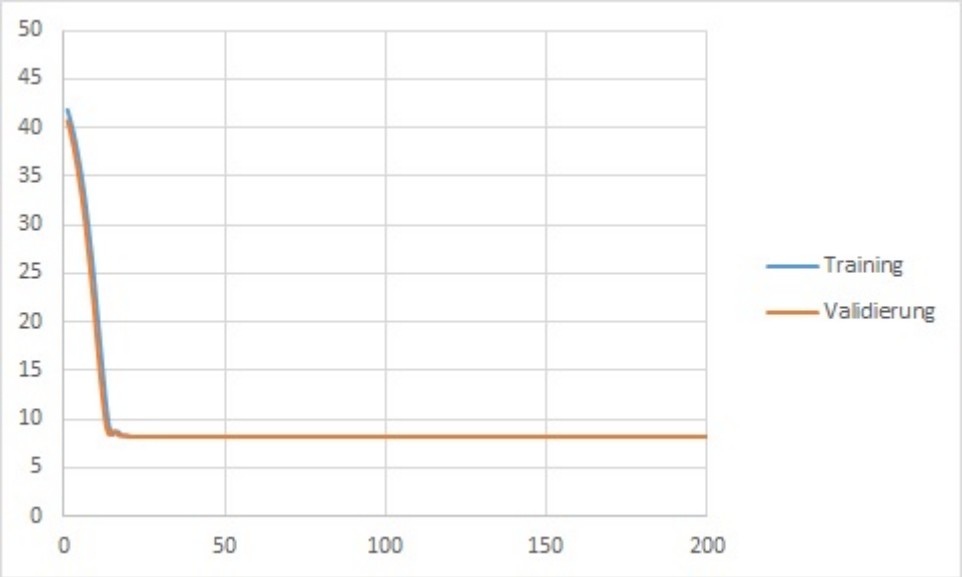

Abbildung 5.9: 75n mit linearer Aktivierungsfunktion

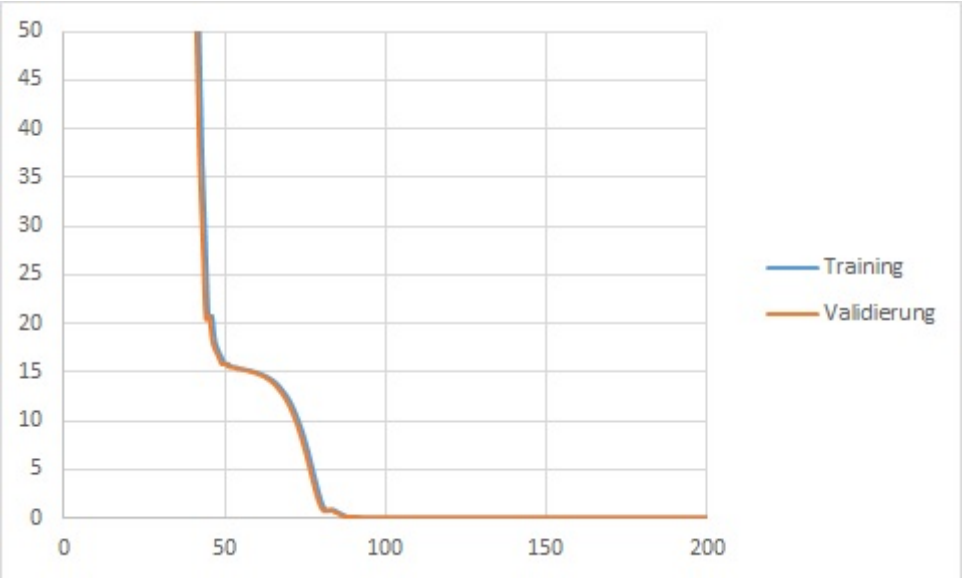

Abbildung 5.10: 2n mit linearer Aktivierungsfunktion

Die Abbildungen 5.8, 5.9 und 5.10 zeigen den Verlauf des Trainings- und Validierungsfehlers. Tabelle 5.3 zeigt die erreichten Fehlerwerte.

Netz	Trainingsfehler	Validierungsfehler
SQRT	<0,001	<0,001
75n	8,169	8,162
2n	0,043	0,062

Tabelle 5.3: Fehler 3. Netz- Iteration

Das Netz mit 75n Hiddenneuronen erreicht wie bei der ersten Iteration keinen angemessenen Trainingsfehler und wird nicht weiter getestet. Das Netzwerk mit SQRT Hiddenneuronen erreichte einen Trainingsfehler von unter 0,001 und wird zusammen mit einem Agenten in 10000 zufälligen Leveln getestet ebenso wie das Netzwerk mit 2n Hiddenneuronen. Nur das SQRT Netz konnte Siege erzielen. Abbildung 5.11 zeigt die Siegesrate des Agenten in den drei durchgeführten Tests mit dem Netzwerk.

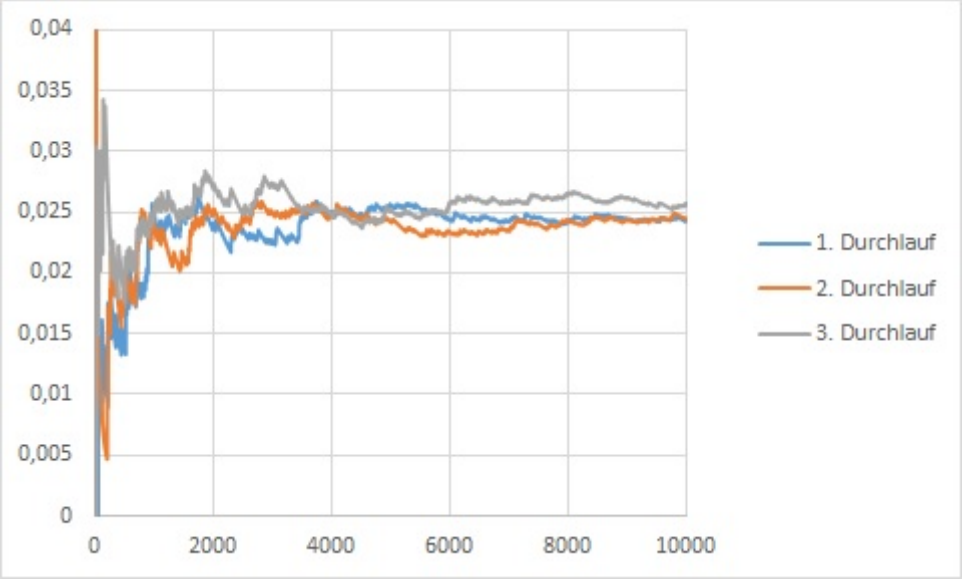

Abbildung 5.11: Siegesrate von Agent mit SQRT Netz 3. Iteration

Von neun erstellten und angelernten Netzwerken konnte nur eins dem Agenten zu Siegen verhelfen. Um den Grund hierfür zu finden, wurde die Generalisierung des Netzes mit SQRT und dem mit 2n Hiddenneuronen genauer überprüft. Hierfür ist ein SARSA- Agent jeweils 10000 mal das zweite, dritte und vierte Level durchlaufen, um Daten von diesen Leveln zu generieren, wie sie das Netzwerk berechnen sollte. Anschließend wurden die beiden Netze mit diesen Daten validiert. Abbildung 5.12 und 5.13 zeigen die Ergebnisse der Validierung.

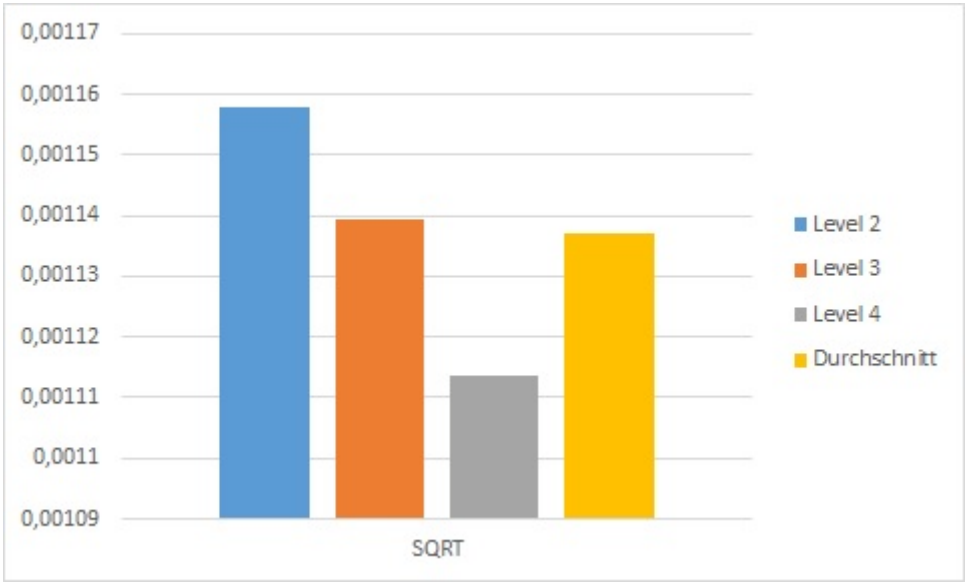

Abbildung 5.12: Validierung des SQRT Netzes mit linearer Aktivierungsfunktion

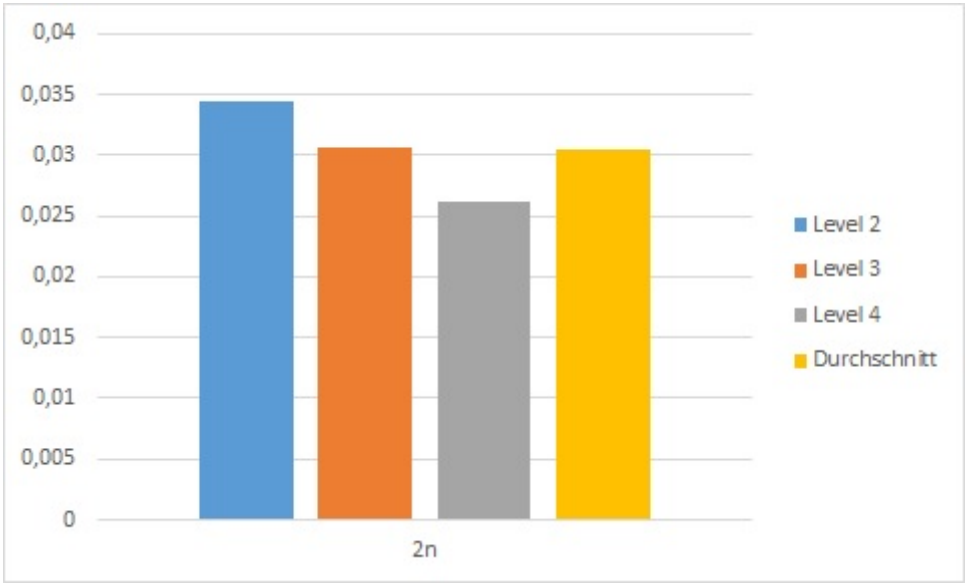

Abbildung 5.13: Validierung des 2n Netzes mit linearer Aktivierungsfunktion

Wie zu sehen ist, sind die Validierungsfehler für vollkommen unbekannte Level ebenso gering wie die Trainingsfehler der Netzwerke. Dies bedeutet, dass die Generalisierung gut funktioniert und es einen anderen Grund für die geringen Siege geben muss. Da die erzeugten

Werte von den neuronalen Netzen den Werten eines SARSA- Agenten entsprechen, mit einer nur sehr geringen Abweichung, kann der Grund für die nicht erfolgten Siege nur in der erlernten Strategie des Agenten liegen. In der Abbildung 5.1 ist zu erkennen, dass der Agent nach 10000 Episoden in einem Level immer noch einen Trend zur Verbesserung zeigt, welches wiederum bedeutet, dass es sich bei der zu dem Zeitpunkt erlernte Strategie noch nicht um eine finale und optimale für das Level handelt. Da das neuronale Netz nach dem Training nicht weiter verändert wird, wäre es nötig, eine möglichst gute anfängliche Strategie erlernt zu haben, um mehr Siege erzielen zu können. Dies wiederum bedeutet, dass viel mehr Zeit in das anfängliche Training des Agenten investiert werden müsste, welche zu der Zeit, die für das Training des Netzwerks benötigt wird, dazukommt.

6 Fazit

Abbildung 5.1 zeigt deutlich, dass die Vergrößerung des Sichtfeldes durchaus die Siegesrate stark erhöhen kann, hierbei jedoch der Zustandsraum so groß wird, dass ein RL- Agent an seine Grenzen stößt und keine Siege mehr erzielen konnte, sofern zufällige Level durchlaufen wurden. Eine verwendete Datenbank benötigt viel mehr Speicher und wächst sehr schnell an durch die großen Zustände und deren Anzahl. Die Verwendung eines neuronalen Netzes anstelle einer Datenbank ist vielversprechend, da so ein Netz eine feste Größe besitzt und zudem gelerntes Wissen generalisieren kann. Leider stellte sich heraus, dass die Verwendung eines neuronalen Netzes alleine keine Möglichkeit darstellt, sofern der Agent weiter dazulernen soll auf Grund der entstehenden Interferenzen im Netzwerk. Die in dieser Arbeit verwendete hybride Lösung mit einer zum Netz zusätzlichen Datenbank ermöglichte es dem Agenten, trotz der Verwendung eines neuronalen Netzes weiterhin dazulernen zu können. Bei neuen Zuständen können hierdurch einem Agenten die generalisierten Q- Werte vom Netzwerk angeboten werden anstelle der mit Nullen initialisierten Werte einer Datenbank. Dies führte dazu, dass ein Agent Siege in zufälligen Leveln erzielen konnte wie in Abbildung 5.11 zu sehen ist. Jedoch ist die Anzahl an Siegen sehr gering, wenn man sie mit der vorherigen Anzahl aus Abbildung 3.12 vergleicht.

Da keine allgemeingültige Aussage getroffen werden kann, wie genau ein neuronales Netz aufgebaut sein muss, damit bei einem speziellem Problem optimale Ergebnisse geliefert werden, müssen zwangsläufig mehrere verschiedene Netze erzeugt und trainiert werden, um ein passendes Netzwerk zu finden. Während die Erstellung von Netzwerken schnell funktioniert, kann das Training eines solchen Netzes viel Zeit beanspruchen, abhängig von der Größe des Netzwerks, Anzahl an Trainingsdaten und Iterationen die durchgeführt werden. In diesem Fall dauerte das Training eines Netzes mit $2 \cdot n$ Hiddenneuronen ca. 48 Stunden.

Die benötigte Zeit für die Durchführung der Testläufe mit einem Agenten ist ebenfalls stark angestiegen, hauptsächlich auf Grund der erhöhten Anzahl an benötigten Datenbankanfragen, da es sich um zu viele Zustände handelt, um diese im Speicher während der Tests vorzuhalten und nur am Ende einmal abzuspeichern.

Somit lässt sich abschließend sagen, dass ein neuronales Netz durchaus einem RL- Agenten dabei helfen kann, seine Aufgabe zu erledigen, jedoch in diesem Fall der benötigte Mehraufwand nicht angemessen ist, um diese Methode empfehlen zu können. Ein kleines Sichtfeld benötigt wenig Speicher und lieferte bei zufälligen Leveln schneller und bessere Siegesraten. Das Sprichwort 'weniger ist mehr' erscheint hier das passende Fazit zu sein.

Literaturverzeichnis

[KNN] *Neuronale Netze Eine Einführung.* – URL http://www.neuronalesnetz.de/downloads/neuronalesnetz_de.pdf. – Abruf: 23-10-2015

[DBBenchmarks] *NoSQL Performance Benchmarks.* – URL http://www.planetcassandra.org/nosql-performance-benchmarks/. – Abruf: 23-10-2015

[Heaton 2011] Heaton, Jeff: *Programming Neural Networks with Encog 3 in Java.* Heaton Research, Inc., 2011. – ISBN 978-1-60439-022-3

[Ott 2013] Ott, Bianca: *Erstellung von Bedarfsprognosen durch Künstliche Neuronale Netze am Beispiel von Backmengenempfehlungen im Einzelhandel*, Hochschule für Angewandte Wissenschaften Hamburg, Bachelorarbeit, 2013

[Shiraga u. a.] Shiraga, Naoto ; Ozawa, Seiichi ; Abe, Shigeo: *A Reinforcement Learning Algorithm for Neural Networks with incremental learning ability.* Graduate School of Science and Technology, Kobe University, Kobe Japan. – URL http://www2.kobe-u.ac.jp/~ozawasei/pub/iconip02a.pdf. – Abruf: 23-10-2015

[Sutton und Barto 1998] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning An Introduction.* Cambridge, Massachusetts : The MIT Press, 1998. – ISBN 0-262-19398-1

Anhang

Inhalt der CD:

- Bachelorarbeit als PDF: Bachelorarbeit Wagener
- Source-Code im Ordner 'MarioAI'
- Excel- Tabellen mit Diagrammen im Ordner 'Auswertungen'

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, 23. Oktober 2015 Benjamin Wagener