DHBW
Duale Hochschule
Baden-Württemberg

Studienarbeit

Erprobung von neuronalen Netzwerken als KI für Computerspiele

Patrick Colbaut, Steffen Czymmeck, Christian Retzbach

16.07.2018

Studiengang, Kurs	Informationstechnik, TIT15
Matrikelnummern	3351060, 3197885, 8783021
Bearbeitungszeitraum	5. und 6. Theoriesemester
Betreuerin	Dipl.-Ing. (BA) Claudia Zinser

Erklärung

gemäß Ziffer 1.1.13 der Anlage 1 zu §§ 3, 4 und 5 der Studien- und Prüfungsordnung für die Bachelorstudiengänge im Studienbereich Technik der Dualen Hochschule Baden-Württemberg vom 29.09.2015.

Wir versichern hiermit, dass wir unsere Studienarbeit mit dem Thema: "Erprobung von neuronalen Netzwerken als KI für Computerspiele" selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt haben. Wir versichern zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Friedrichshafen, 16.07.2018

Patrick Colbaut	Steffen Czymmeck	Christian Retzbach

Freistellungsvermerk

Diese Studienarbeit wird hiermit für die Aufbewahrung über den gesetzlichen Aufbewahrungszeitraum hinaus für studienrelevante Zwecke freigestellt. Dies beinhaltet, ist jedoch nicht begrenzt auf die Einsicht durch Studenten und Dozenten, die Nutzung als Basis für weitere Studienarbeiten und die Aufnahme in den Index der Bibliothek.

Sollte dieser Freistellungsvermerk eventuellen Anforderungen, die nach der Erstellung dieses Dokuments entstehen, nicht genügen, so ist Kontakt mit den Autoren zur Einholung eines gültigen Freistellungsvermerks aufzunehmen.

Inhaltsverzeichnis

Abbildungsverzeichnis

Listingverzeichnis

1 Einleitung

Bereits 1943 entwickelten Walter Pitts und Warren McCulloch ein einfaches mathematisches Modell eines Neurons[1]. Sie zeigten, dass es mit einfachsten neuronalen Netzen möglich ist, simple Funktionen zu berechnen. Auch wenn sie keine praktischen Anwendungen für ihr Modell vorschlugen, legten sie doch den einen der Grundsteine, welche maschinelles Lernen ermöglichten.

Im Jahre 1949 beschrieb Donald Hebb in seinem Buch *The Organization of Behaviour* eine Trainingsregel[1] für den Verband von Neuronen, welche gemeinsame Synapsen haben. Er fand heraus, dass konditionierte Reflexe sich auf die Charakteristika einzelner Neuronen zurückführen lassen.

1958 generalisierte Frank Rosenblatt mit der Entwicklung eines Perzeptrons die Arbeit von Pitts und McCulloch[2]. Er bewies mathematisch, dass sein Modell für beliebige Trainingsdaten, in endlicher Anzahl an Schritten, einen passenden Gewichtungsvektor finden kann. Im darauffolgenden Jahr entwickelte er mit seinem Kollegen Charles Wightman den ersten erfolgreichen Neurocomputer: *Mark I Perceptron*. Mit seinem $20 \cdot 20$ Pixel großen Bildsensor konnte er bereits einfach Ziffern erkennen.

In den 60er Jahren stieß die Forschung im Bereich der neuronalen Netze mehr auf einige schwerwiegende Probleme. Eines dieser Probleme entstand 1969 durch ein Buch von Marvin Minsky und Seymoure Papert, *Perceptrons*. In dieser Arbeit behaupteten sie, dass es einem Perzeptron nicht möglich ist, die simple logische Funktion XOR zu berechnen[3]. Es ist zwar möglich diese Funktion mit einem zweischichtigen Netz mit drei Neuronen darzustellen, allerdings war zu dieser Zeit noch kein Trainingsalgorithmus für diese Netztopologie bekannt.

[1] Hebbsche Lernregel

Erst wieder in 1983 wurde dem Bereich der neuronalen Netze durch DARPA[2], eine Behörde des US-Verteidigungsministeriums, durch die Subventionierung von Projekten, neues Leben eingehaucht. Im Jahre 1986 entwickelten Forscher der PDP Group[3] einen Artikel, in dem sie einen Trainingsalgorithmus für mehrschichtige Netzwerke vorstellten, genannt *Backpropagation*[4]. Mit diesem Algorithmus war es möglich, dass von Minsky und Papert als unlösbar beschriebene Problem zu beseitigen.

Durch Backpropagation war es möglich neuronale Netze für die Lösung einer Vielzahl von Problemen einzusetzen. Eine der ersten bekannten Anwendungen von Backpropagation ist das künstliche neuronale Netz *NETtalk*. Mit diesem war es Mitte der 1980er Jahre möglich, nach 50 Durchläufen eines geringen Trainingsdatensatzes von nur 1024 Wörtern, mit einer Genauigkeit von 78% geschriebene, englische Sprache in eine Codierung der Aussprache, also Grapheme in Phoneme, umzuwandeln[5].

2016 gelang es dem Computerprogramm *AlphaGo* der Firma DeepMind den als weltbesten Profispieler angesehenen Südkoreaner Lee Sedol vier zu eins in dem Brettspiel Go zu schlagen[6]. Dies stellte den ersten Sieg eines Computerprogrammes gegen einen Profispieler ohne Einschränkung dar und ist eine der grundlegenden Motivationen für diese Arbeit.

[2]Defense Advanced Research Project Agency
[3]Parallel Distributed Processing

2 Aufgabenstellung

Ziel der Arbeit ist es, Erfahrungen mit künstlichen Intelligenzen im Bereich von Computerspielen zu sammeln. Der Schwerpunkt liegt auf den Spielen Tic-Tac-Toe und Vier gewinnt und dem Einsatz neuronaler Netze als künstliche Intelligenz. Zunächst soll ein neuronales Netz erstellt werden, welches Tic-Tac-Toe spielen kann. Probleme bei der Entwicklung und dem Training des Netzes sind zu dokumentieren. Anschließend soll untersucht werden, welche Anpassungen am Netz vorzunehmen sind, um das Netz ein ähnliches, aber komplexeres Spiel wie Vier gewinnt erfolgreich spielen zu lassen. Ziel dieser Arbeit ist nicht, eine perfekte künstliche Intelligenz für beide Spiele zu liefern, sonder vielmehr die auftretenden Probleme bei der Entwicklung und die Skalierbarkeit des Endproduktes zu untersuchen und zu dokumentieren.

3 Hintergründe

In diesem Kapitel werden die nötigen Grundlagen für das Verständnis dieser Arbeit allgemein erläutert. Zunächst folgt eine Beschreibung der gewählten Spiele, anschließend folgt eine Kurzeinführung in das Themengebiet neuronaler Netze.

3.1 Spiele

Brettspiele wie Tic-Tac-Toe eignen sich aus vielerlei Gründen für die anfängliche Erprobung eines neuronalen Netzes. Im Falle von Vier gewinnt ist es fast schon erforderlich, da ein Minimax-Algorithmus[4] mit der gewaltigen Anzahl an möglichen Zuständen ohne Optimierungen an seine Grenzen stößt. Ein Vorteil der ausgewählten Spiele, Tic-Tac-Toe und Vier gewinnt, ist, dass sich der Spielzustand nur ändert, wenn die Spieler mit diesem durch vordefinierte Spielzüge interagieren. Die Anzahl an Interaktionsmöglichkeiten ist ebenfalls überschaubar: 9 bei Tic-Tac-Toe und 7 bei Vier gewinnt. Außerdem handelt es sich bei beiden Spielen um Spiele mit perfekter Information, was bedeutet, dass jedem Spieler zum Zeitpunkt einer Entscheidung alle vorangegangenen Entscheidungen der Mitspieler, sowie Zufallsinformationen bekannt sind. Ein Spiel mit imperfekter Information wäre beispielsweise das Kartenspiel Poker. In diesem Unterkapitel wird näher auf die beiden ausgewählten Spiele eingegangen.

3.1.1 Tic-Tac-Toe

Tic-Tac-Toe ist ein Brettspiel, dessen Ziel darin besteht, als erster Spieler ein Gewinnmuster zu erzeugen[7]. Das Spielbrett besteht aus einem 3×3 Gitter, auf dem zwei Spieler abwechselnd jeweils ein Kreuz (X) oder einen Kreis (O) setzen. Der erste Spieler, dem es

[4] Ein Algorithmus, welcher die optimale Spielstrategie für endliche Zwei-Personen-Nullsummenspielen mit perfekter Information findet.

gelingt drei seiner Markierungen vertikal, horizontal oder diagonal zu verbinden, gewinnt das Spiel.

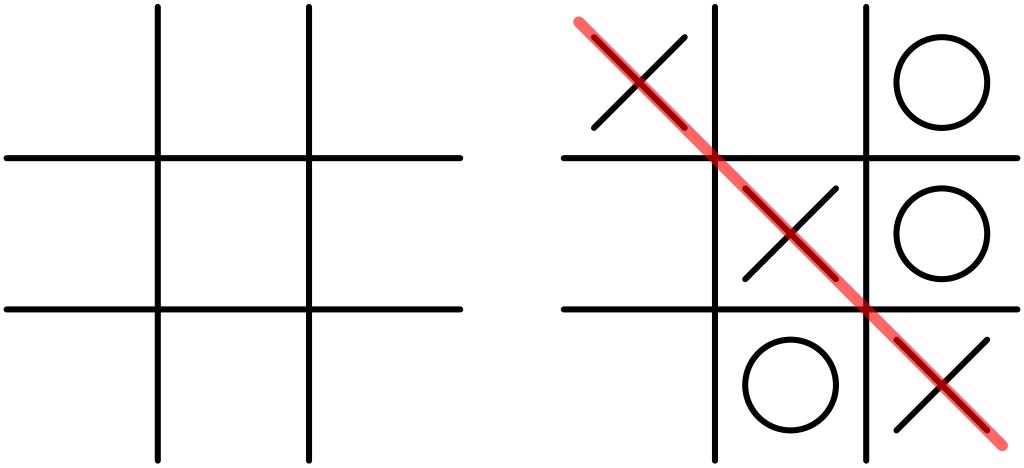

Abbildung 3.1: Auf der linken Seite ist ein leeres Spielbrett zu sehen, rechts gewinnt X

In Abb. 3.1 gewinnt der Spieler mit der Markierung X, da es ihm gelungen ist drei seiner Markierungen diagonal miteinander zu verbinden. Diese Gewinnsituation ist ein mögliches Ergebnis aus 26 830 möglichen Spielen[8]. Ein Tic-Tac-Toe Spiel, das von zwei perfekten Spielern gespielt wird, endet immer in einem Unentschieden, was Tic-Tac-Toe zu einem "Futile Game"[5] macht[9]. Newell und Simon definierten 1972 eine gewichtete Entscheidungsliste, welche es einem Spieler ermöglicht eine perfekte Partie Tic-Tac-Toe zu spielen[10]:

1. **Gewinn**: Wenn der Spieler zwei Markierungen auf dem Spielbrett hat, kann er mit einer dritten diagonal, vertikal oder horizontal gewinnen.

2. **Blocken**: Wenn der Gegenspieler zwei Markierungen auf dem Spielbrett hat und es ihm möglich ist eine dritte zum Sieg zu setzten, so muss der Spieler dieses Feld blockieren.

[5] dt.: zweckloses Spiel

3. **Gabelung**: Der Spieler sollte eine Situation schaffen, in der er zwei Möglichkeiten hat, mit dem nächsten Zug zu gewinnen.

4. **Blockieren einer Gegner-Gabelung**: Ist es dem Gegenspieler möglich, mit seinem nächsten Zug eine Gabelung zu erwirken, so muss der Spieler diese Möglichkeit unterbinden.

5. **Zentrum**: Der Spieler markiert das Zentrum des Brettes.

6. **Gegenüberliegende Ecke**: Hat der Gegner eine Markierung in einer Ecke gesetzt, so muss der Spieler die gegenüberliegende Ecke markieren.

7. **Leere Ecke**: Der Spieler markiert eine leere Ecke.

8. **Leere Seite**: Der Spieler markiert eine Position in der Mitte einer der vier Seiten.

Folgen beide Spieler den Anweisungen dieser Liste, so endet das Spiel immer in einem Unentschieden. Diese Tatsache kann man als Metrik für künstliche Intelligenzen nutzen. Zwei künstliche Intelligenzen, welche das Spiel perfekt beherrschen, werden gegeneinander nie gewinnen oder verlieren.

3.1.2 Vier gewinnt

Vier gewinnt[6] ist, genau wie Tic-Tac-Toe, ein Spiel, in dem es darum geht, durch strategische Züge ein spiel-gewinnendes Muster mit seinen Spielsteinen auf dem Spielbrett zu erzeugen. Das Spielbrett ist in der klassischen Version auf sieben senkrechte Spalten und sechs waagrechte Reihen begrenzt. Bei Vier gewinnt müssen allerdings vier Steine vertikal, horizontal oder diagonal verbunden werden. Im Gegensatz zu Tic-Tac-Toe ist es den Spielern auch nicht möglich auf allen freien Spielfeldern einen ihrer Spielsteine zu setzen. Die Schwerkraft sorgt dafür, dass immer das unterste freie Spielfeld einer Spalte

[6] im englischen Sprachraum unter anderem bekannt als Connect Four

belegt wird. Damit verkleinert sich die Menge der möglichen Aktionen von $6 \cdot 7 = 42$ auf sieben. Die Menge der möglichen Zustände ist allerdings weitaus höher:

- 4 531 985 219 092 Zustände[11, 12]
- 1 904 587 136 600 Gewinn-Zustände[12]
- 713 298 878 Unentschieden-Zustände[12]

Vier gewinnt wurde 1988 von James Dow Allen und davon unabhängig von Victor Allis mathematisch gelöst[13, 14]. 1995 gelang es John Tromp das Spiel mit einer Brute-Force-Methode[7] zu lösen. Das Ergebnis der Untersuchung zeigt, dass die perfekte Vier gewinnt Partie immer vom Spieler gewonnen wird, welcher den ersten Stein setzt[15].

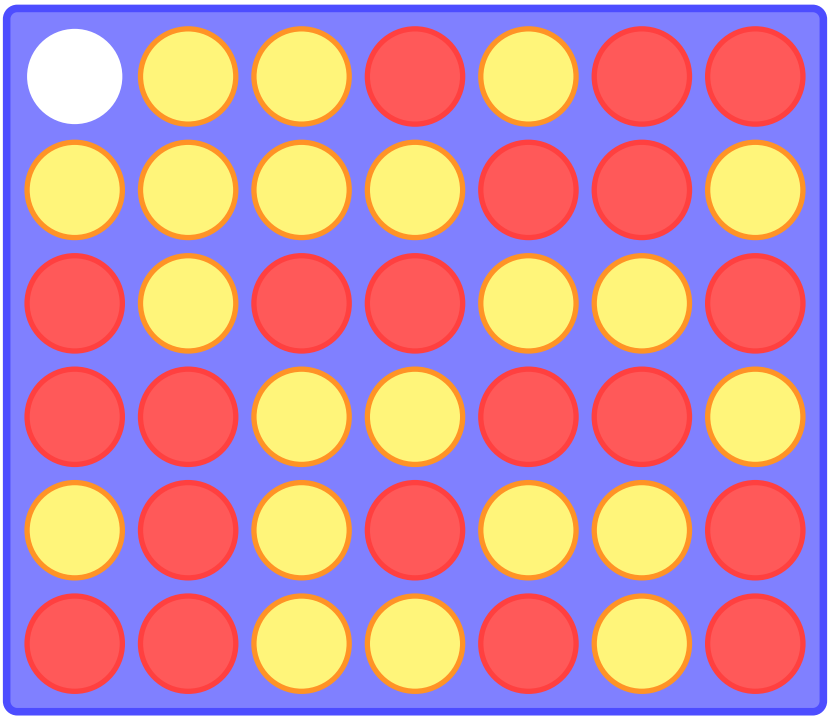

Abbildung 3.2: Endzustand einer perfekt gespielten Vier gewinnt Partie (aus [12])

[7] von englisch ”brute force”, deutsch ”rohe Gewalt”

In Abb. 3.2 ist der letzte Zug einer möglichen perfekt gespielten Partie Vier gewinnt zu sehen. Gelb gewinnt in der linken oberen Ecke mit vier horizontal verbundenen Steinen, nachdem Rot im vorangegangenen Zug in die linke Spalte setzen musste, da alle anderen Spalten bereits gefüllt waren.

3.2 Künstliche Intelligenz in Computerspielen

Ein Computerspiel ist ein Programm, welches zum Ziel hat, fesselnde Erfahrungen für den Benutzer zu generieren. Im Gegensatz zu Filmen, Büchern oder Fernsehen sind Computerspiele interaktiv[16]. Es ist möglich, Computerspiele in verschiedenste Gruppen, Genres und Klassen einzuteilen. Um die Motivation dieser Arbeit nachvollziehen zu können, werden sie in Einzel- und Mehrspieler Spiele eingeteilt: Spiele, die nur einen menschlichen Spieler vorsehen, nennt man Einzelspieler Spiele. Mehrspieler Spiele lassen sich in kompetitive Spiele, in denen zwei oder mehrere menschliche Spieler in Teams oder einzeln gegeneinander konkurrieren, und kooperative Spiele, in denen mehrere Spieler zusammen ”gegen” das Spiel arbeiten, einteilen.

In Spielen aller Art kann es non-player characters[8] geben. Diese NPCs können zum Beispiel in Mehrspieler Spielen als Ersatz für menschliche Spieler dienen. In simplen Spielen, wie zum Beispiel Tic-Tac-Toe, kann ein NPC einfach durch einen Minimax-Algorithmus verwirklicht werden. Doch bereits bei Vier gewinnt stößt eine solche Herangehensweise an ihre Grenzen. Dabei kann Vier gewinnt von der Regelmenge noch als simples Spiel betrachtet werden. Computerspiele übertrafen schnell den Komplexitätswert von Vier gewinnt und so musste nach anderen Lösungen für die Implementierung von NPCs gesucht werden. Den Entwicklern von Computerspielen sind die Regeln der von ihnen entworfenen Spielwelt bewusst, weswegen sie aus diesen ableiten können, was eine gute Strategie ausmacht oder welche Züge in bestimmten Situationen gut sind. Mit dieser

[8]NPC, deutsch ”Nicht-Spieler-Charakter” (NSC)

Methode können spezifische NPCs implementiert werden, deren Umsetzung allerdings sehr aufwändig ist. Zudem sind sie meist auch nicht sehr ”intelligent”, da die definierten Regeln allgemein gehalten werden. Meist gibt man dem vom Computer gesteuerten Spieler im Hintergrund unfaire Vorteile, wie zum Beispiel mehr Rohstoffe, um dessen mangelnde Intelligenz auszugleichen. Durch diese Umsetzung unterscheidet sich eine Partie gegen den Computer meistens drastisch von einer Partie gegen einen menschlichen Spieler. Fehler in den Regeln können durch den menschlichen Spieler ausgenutzt werden, während der Computer nicht aus seinen Fehlern lernt und immer wieder in dieselbe Falle tritt.

Um den Spielern bessere Gegenspieler liefern zu können und damit die Spielqualität zu erhöhen, bietet es sich an, künstliche Intelligenz in Form von neuronalen Netzen einzusetzen. Diese können auch komplexe Spiele wie Schach und Go intelligent spielen und meistern[6].

3.3 Künstliche Neuronale Netze

Künstliche neuronale Netze[9] sind von biologischen neuronalen Netzen, wie zum Beispiel dem menschlichen Gehirn, inspiriert[17]. Genau wie ihre biologische Inspiration bestehen KNNs aus vielen, miteinander vernetzten Neuronen. Im Verbund zusammen können sie Informationen verarbeiten.

Ein Netzwerk ”lernt”, indem es eine Kostenfunktion minimiert. Die Kostenfunktion errechnet sich in einem Trainingsdurchlauf aus der Differenz der vom Netzwerk ausgegebenen Werten und den tatsächlichen Werten. Bei großer Differenz zwischen diesen Werte liefert das Netz ungenaue und ”schlechte” Ergebnisse. Je kleiner die Differenz, desto ”besser” sind die vom Netz gelieferten Werte.

[9]KNN, engl.: artificial neural network, ANN

Es gibt für den Entwickler verschiedene Stellschrauben, um ein Netz verbessern zu können. Dazu gehören unter anderem die Anzahl der Schichten oder die Anzahl der Neuronen pro Schicht. Das Netz selbst ändert zwei Werte, um die ihm gegebene Kostenfunktion zu minimieren: Die Gewichtungen der Verbindungen zwischen den Neuronen und die Schwellwerte der Neuronen.

3.3.1 Neuronen

Ein biologisches Neuron ist die Basis-Recheneinheit des menschlichen Gehirns[18]. Azvedeo und Team kamen 2009 zu dem Ergebnis, dass das Gehirn eines männlichen Erwachsenen durchschnittlich etwa 86 ± 8 Milliarden Nervenzellen enthält[19].

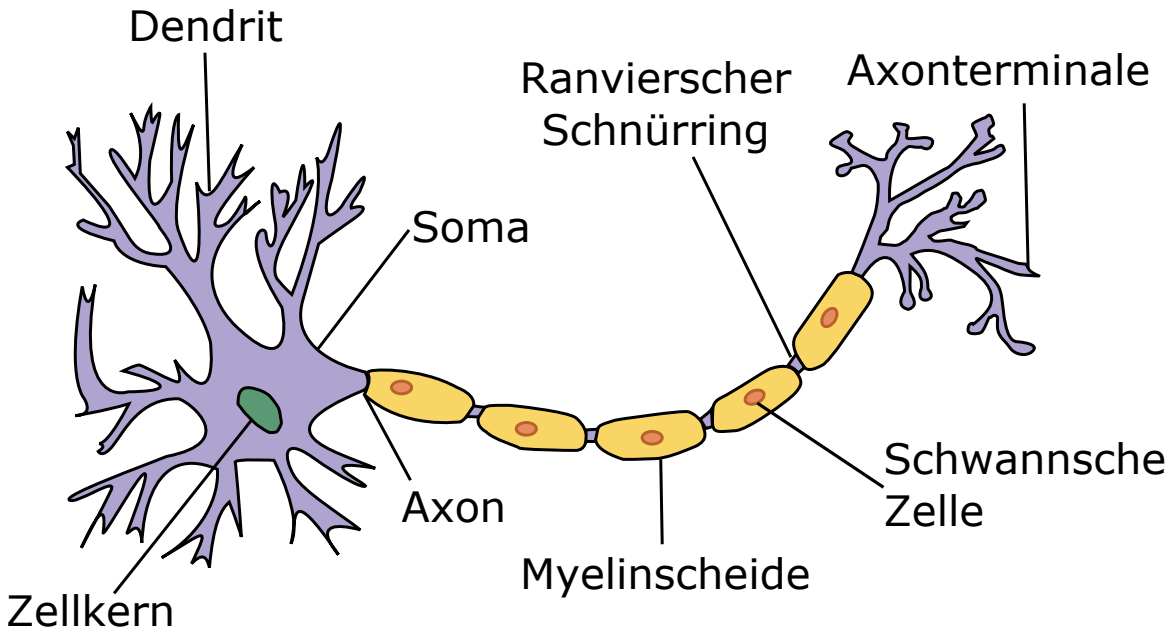

Abbildung 3.3: Aufbau eines biologischen Neurons (aus [20])

Der Aufbau eines biologischen Neurons ist in Abb. 3.3 dargestellt. Das Neuron nimmt Erregungen in Form von elektrischen Signalen an den Dendriten auf. Diese werden über das Soma an das Axon weitergeleitet. Wird dort ein gewisser Schwellwert, das

sogenannte Schwellenpotential, überschritten, so wird ein Aktionspotential entlang des Axons weitergegeben. Dieses Aktionspotential kann zum Beispiel von anderen Neuronen über die Dendriten aufgenommen und weiterverarbeitet werden.

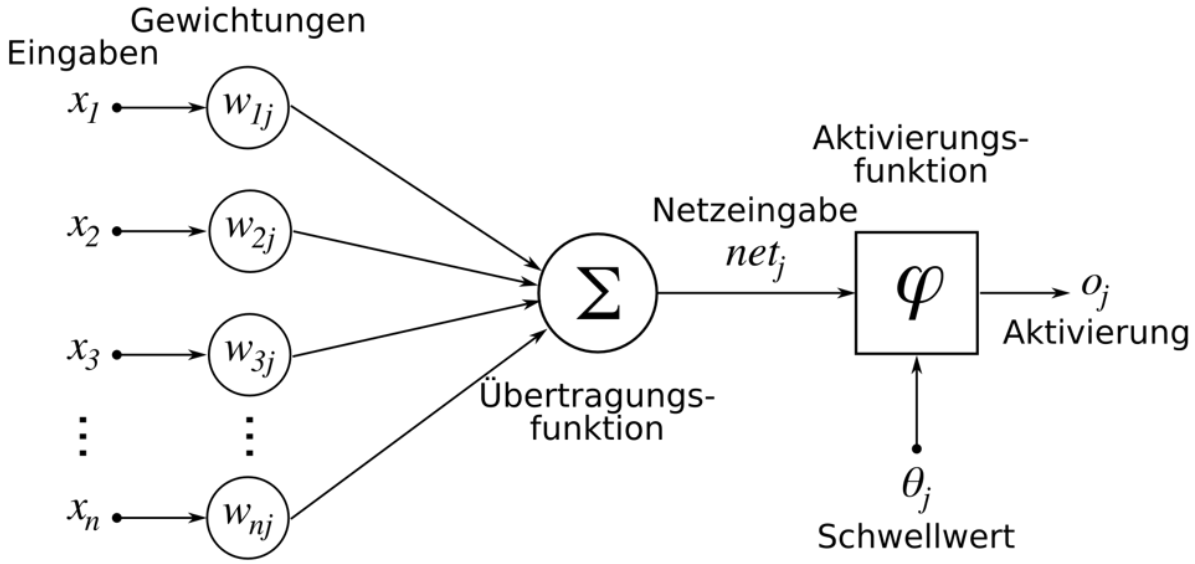

Abbildung 3.4: Aufbau eines künstlichen Neurons (aus [21])

Wie in Abb. 3.4 zu sehen, orientiert sich ein künstliches Neuron im Aufbau und in der Funktion stark an seinem biologischen Konterpart. Die Eingaben x_i werden mit Gewichtungen w_{ij} versehen, welche den Grad des Einflusses bestimmen, den die Eingaben auf das Neuron ausüben. Diese Eingaben werden in einer Übertragungsfunktion zur Netzeingabe net_j aufsummiert. Diese wird in die Aktivierungsfunktion φ des Neurons übergeben und mit einem Schwellwert θ_j verglichen. Ist die Netzeingabe groß genug, erfolgt die Aktivierung, bzw. Ausgabe $o_j = \varphi(net_j - \theta_j)$ des Neurons. Diese Ausgabe kann wiederum von beliebig vielen weiteren Neuronen als Eingabe weiterverarbeitet werden. Die Art der Aktivierungsfunktion des Neurons kann beliebig vom Entwickler gewählt werden, meist werden allerdings Funktionen verwendet, die ein Ergebnis zwischen 0 und 1 ausgeben.

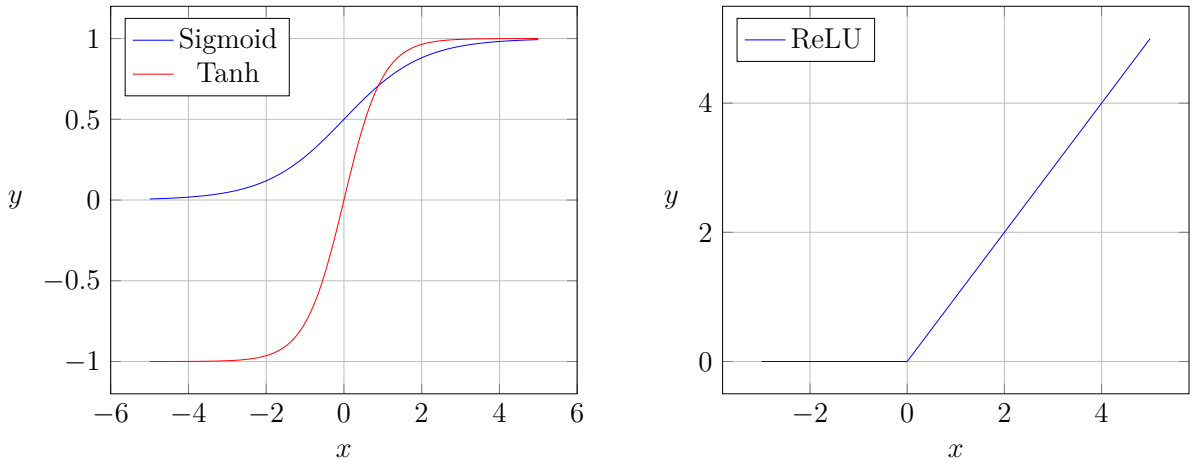

Abbildung 3.5: Unterschiedliche Aktivierungsfunktionen

Abb. 3.5 zeigt die graphischen Darstellungen der Aktivierungsfunktionen Sigmoid, Tanh und ReLU. Sigmoid und Tanh zeichnen sich durch ihren begrenzten S-förmigen Graphen aus, während ReLU bei einem Eingabewert über 0 linear steigt.

3.3.2 Bekannte Netzstrukturen

KNN können als gerichtete, gewichtete Graphen dargestellt werden. Im Graphen repräsentieren die einzelnen Neuronen Knotenpunkte und die gewichteten Verbindungen zwischen den Neuronen die Pfade. Man kann in dieser Darstellung die Netze in zwei Gruppen unterteilen[22]:

- Feedforward-Netze, die keine Schleifen enthalten
- Rekurrente Netze, die Schleifen enthalten

Die bekannteste verwendete Netzwerkarchitektur der Gruppe der Feedforward-Netze ist das mehrlagige Perzeptron [10]. In diesem werden alle Neuronen in einer Schicht mit allen

[10] engl.: multilayer perceptron, MLP

Neuronen der darauffolgenden verknüpft[11]. Zusätzlich können Neuronen Verbindungen zu den übernächsten Schichten besitzen[12].

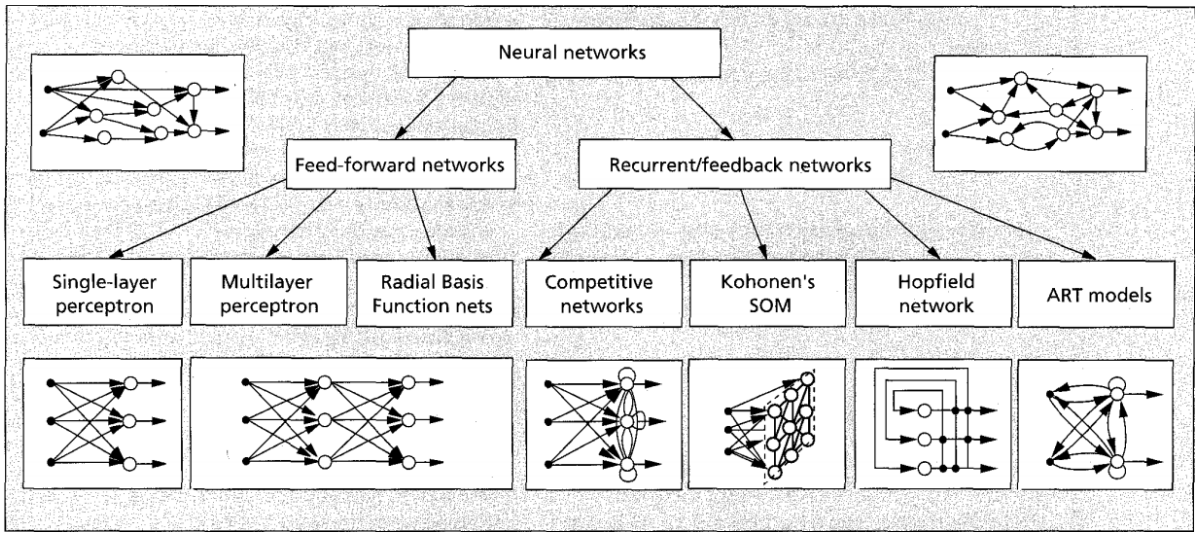

Abbildung 3.6: Unterschiedliche Netzwerkarchitekturen (aus [22])

In Abb. 3.6 zu sehen sind einige bekannte Netzwerkarchitekturen, aufgeteilt in die zwei Gruppen der feedforward- und rekurrenten Netze.

[11] engl.: fully connected
[12] engl. short-cuts

4 Netzwerk Training

In diesem Kapitel werden die verschiedenen Trainingsarten von neuronalen Netzen erläutert. Es wird insbesondere auf das in dieser Arbeit verwendete Reinforcement Learning und den Markov Decision Process[13] eingegangen.

4.1 Supervised Learning

In der Methode des Supervised Learning[14] werden die vom künstlichen neuronalen Netz gelieferten Werte mit vorgegebenen Erwartungswerten verglichen. Die Differenz dieser Werte dient als Metrik für die Qualität des Netzes. Eine kleine Differenz liefert ungefähr die gewünschten Werte und steht somit für ein gewünschtes Ergebnis. Eine große Differenz hingegen sagt aus, dass das Netz keine guten Ergebnisse für diesen Durchlauf geliefert hat. Durch die Minimierung dieser Differenz gelangt das Netz ans Ziel und liefert nach genügend Zeit akzeptable Ergebnisse. Bei Supervised Learning gibt es zu jedem Trainingsinput einen erwarteten Output. Dieser ist dem ”Lehrer” bekannt und wird dem Netz durch ihn mitgeteilt. Es kann eine sofortige Wertung der Ausgabe erfolgen.

4.2 Reinforcement Learning

Die Methode des Reinforcement Learning[15] beschäftigt sich mit der Art, wie ein Software Agent eine bestimmte Aktion in einer Umgebung ausführen sollte, um eine Belohnung zu maximieren. Im Gegensatz zum Supervised Learning bestehen die Trainingsdaten nur aus einem Input. Das Netz lernt die Umgebung und die im jeweiligen Zustand optimalen Aktionen durch eine Mischung aus exploration (Erforschung unbekannter Zustände) und exploitation (Nutzung von aktuellem Wissens)[23]. In Abb. 4.1 ist die Interaktion

[13] Markov Entscheidungsprozess
[14] überwachtes Lernen
[15] bestärkendes Lernen

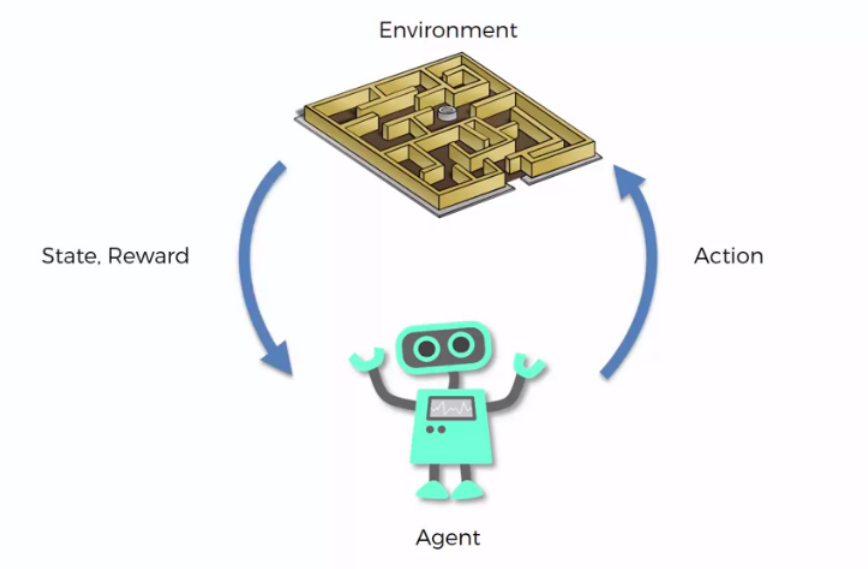

Abbildung 4.1: Darstellung von Reinforcement Learning (aus [24])

eines Reinforcement Learning Prozesses dargestellt. Der Agent führt eine Aktion aus, welche die Umgebung verändert. Die Umgebung gibt dem Agenten den neuen Zustand der Umgebung zurück. Mit dem neuen Zustand können neue Entscheidungen getroffen werden und eine neue Aktion ausgeführt werden.

4.3 Markov Decision Process

Mit dem Markov Decision Process ist es möglich, eine Entscheidungsfällung in einer nicht deterministischen Umgebung mathematisch zu repräsentieren. Bei jedem Zeitschritt befindet sich der Prozess in einem Zustand s und kann eine in diesem Zustand mögliche Aktion a ausführen. Der Prozess gibt dem Entscheider eine Belohnung $R_a(s, s')$, indem er in einen zufälligen Zustand s' übergeht. Die Wahrscheinlichkeit in den Zustand s' zu gehen hängt dabei von der gewählten Aktion ab. Dies erfüllt die Markov-Eigenschaft, welche besagt, dass man für eine optimale Entscheidung nur den aktuellen Zustand benötigt. Die vergangenen Zustände sind für die Entscheidung nicht relevant[25].

5 Q-Learning

In diesem Kapitel wird Q-Learning als Methode für das selbständige Trainieren eines Agenten in einer vordefinierten Umgebung erläutert. Außerdem wird die in dieser Arbeit entwickelte Q-Table Implementierung dargestellt und analysiert.

5.1 Theoretische Hintergründe

Q-Learning ist eine Methode des Unsupervised Learning[16] im Teilbereich des Temporal Difference Learning[17], die es einem Agenten ermöglicht, in einem vorgegebenen Zustand die optimale Aktion auszuführen. Watkins stellte die Idee des Q-Learningns zuerst 1989 vor[26]. Q-Learning liefert als Ergebnis dabei den Nutzen einer Aktion, im Gegensatz zu TD-Learning, welches den Zustand bewertet. Der Vorteil dieser Methode ist die Unabhängigkeit von der Umgebung. Sie benötigt nur den Zustand und die möglichen Aktionen sowie eine Belohnung als Ziel und kann so, ohne weitere Anpassungen, die optimale Strategie finden. Francisco S. Melo hat bewiesen, dass Q-Learning für jeden finiten Markov Decision Process, mit unendlicher Zeit und einer teilweise zufälligen Erforschungsstrategie, die optimalen Aktionen findet, welche zum Ziel führen[27]. Q-Learning lernt durch das Aktualisieren der Qualitätsbewertungen der Aktionen in einem bestimmten Zustand. Zum besseren Verständnis wird das Tutorial von John McCullock hier näher erläutert[28].

In Abb. 5.1 ist die Spielumgebung zu sehen. In einem Haus, welches aus fünf Räumen besteht, ist das Ziel des Agenten das Haus auf kürzestem Wege zu verlassen. ”Draußen” kann man als sechsten Raum sehen, in Abb. 5.1 dargestellt als Raum 5. Der Agent kann durch alle offenen Türen die Räume wechseln, jedoch nicht durch geschlossene Wände.

[16] dt.: unbeaufsichtigtes Lernen
[17] TD-Learning

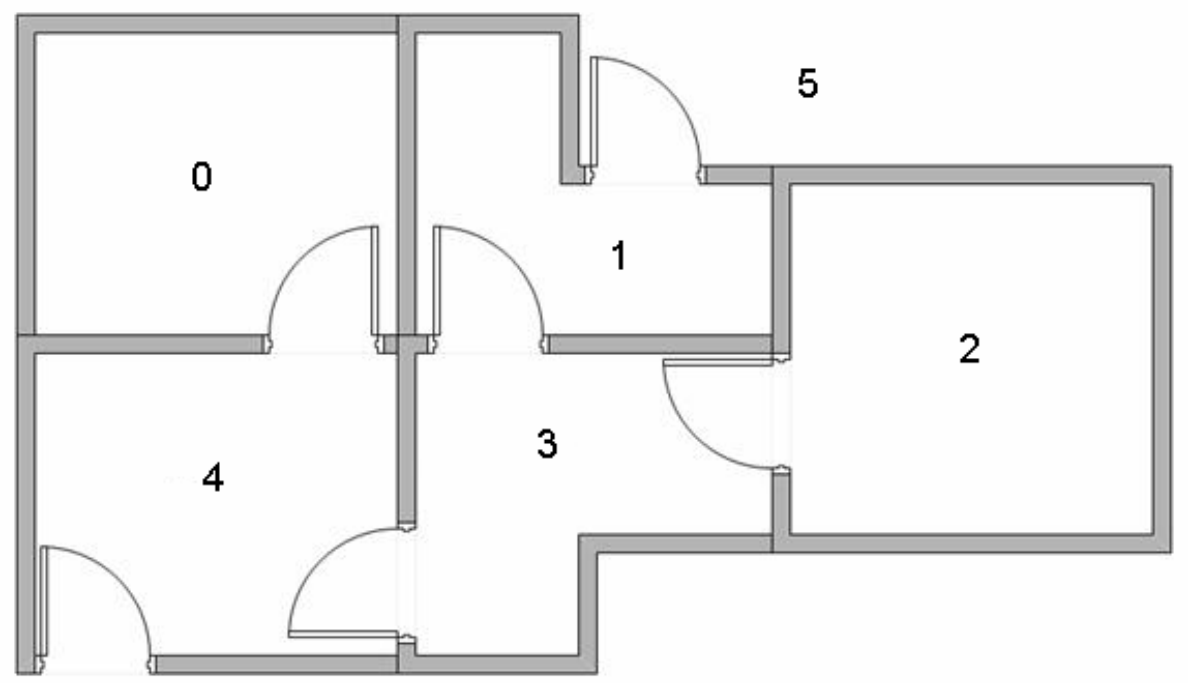

Abbildung 5.1: Gebäudegrundriss als Spielumgebung(aus [28])

Daraus folgt, dass das Ziel Raum 5 zu erreichen nur von Raum 1 oder Raum 4 möglich ist. Eine logische Vermutung wäre also, dass der Agent die Aktionen in den Raum 1 oder 4 zu gehen stärker bewertet als die Bewegung in andere Räume. Diesen Gebäudegrundriss kann man als gerichteten und gewichteten Graphen darstellen, zu sehen in Abb. 5.2. Der Agent kann sich entlang der Pfade in diesem Graphen durch die Räume bewegen, beispielsweise von Raum 2 zu Raum 3 und wieder zurück.

Startpunkt des folgenden Beispiels ist der Raum 2. Die Aktion in Raum 5 zu gelangen ist mit einer Belohnung von 100 versehen, um dem Q-Learning Agenten das Ziel vorzugeben. Aus Raum 5 ist es möglich wieder in den Raum 5 zu gelangen, in Abb. 5.2 wird dies durch die Schleife am Knoten 5 gezeigt. Die Q-Learning Methode versucht den Zustand mit der größten Belohnung zu erreichen. Es ist also davon auszugehen, dass der Agent ewig in Raum 5 bleiben wird, sobald er ihn erreicht hat. Der Agent entscheidet sich bei jeder

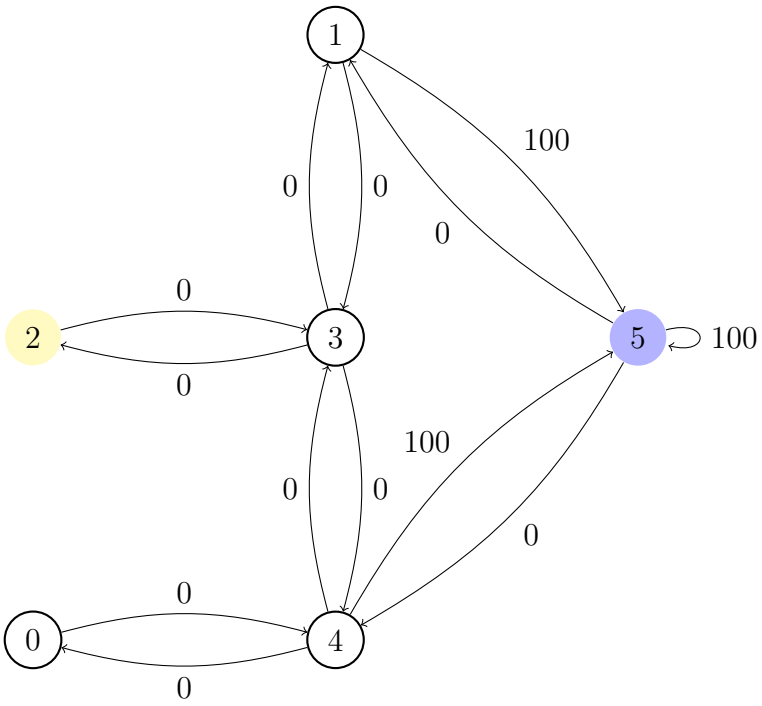

Abbildung 5.2: Darstellung der Spielumgebung als Graph (in Anlehnung an [28])

Aktion anhand des berechneten Q-Wertes. Zu Beginn des Trainings wird eine Matrix R mit allen Zuständen und deren möglichen auszuführenden Aktionen mit 0 initialisiert.

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{5.1}$$

In Gleichung (5.1) ist die initiale Matrix Q zu sehen. Die Zustände, in Form der Räume 0-5, sind zeilenweise aufgelistet, die Aktionen, in einen Raum zu wechseln, spaltenweise. Es ist dem Agenten noch nicht bekannt welche Aktion optimal in einem gewissen Zustand ist. Damit ist die Q-Werte Matrix initialisiert und der Agent kann mit dem Lernen beginnen.

$$Q^{new}(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{alter Wert}} + \underbrace{\alpha}_{\text{Lernrate}} \cdot \overbrace{\Big(\underbrace{r_t}_{\text{Belohnung}} + \underbrace{\gamma}_{\text{Gewichtung}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\substack{\text{Schätzung des} \\ \text{optimalen} \\ \text{Zukunftwertes}}} \Big)}^{\text{gelernter Wert}} \tag{5.2}$$

In Gleichung (5.2) ist die Formel, welche zur Berechnung der Qualität Q einer Aktion a, die im Zustand s ausgeführt wird, zu sehen. Der Agent lernt, indem er die Matrix in Gleichung (5.1) mit den neu berechneten Q-Werten aktualisiert. Dabei ist $Q(s_t, a_t)$ der bereits vorhandene Wert der Matrix, α eine vorgegebene Lernrate, γ ein vorgegebener Gewichtungsfaktor, r die Belohnung des aktuellen Zustandes s und $\max_a Q(s_{t+1}, a)$, der geschätzte optimale Zukunftswert. Die Lernrate α, ein Wert zwischen $0 \leq \alpha \leq 1$ gibt an, wie der Agent ältere Informationen bei der Entscheidung in Betracht zieht. Bei einem Wert von $\alpha = 0$ lernt der Agent nicht aus neuen Erfahrungen und nutzt nur das bereits gelernte Wissen als Entscheidungsgrundlage. Bei einem Wert von $\alpha = 1$ wird nur das neueste Wissen genutzt. Mit dem Gewichtungsfaktor γ kann der Einfluss von zukünftigen Belohnungen bestimmt werden. Ein Wert nahe 0 fokussiert den Agenten auf aktuelle Belohnungen, ein Wert nahe 1 legt den Schwerpunkt auf langfristige Belohnungen, die in der Zukunft liegen. Der Agent beginnt nun mit dem Erkunden der Zustände. So lange er keinen Zustand erreicht, welcher eine Belohnung enthält, ändert sich anfangs nichts an der Q-Werte Matrix. Startet er in Raum 2 und bewegt sich in Raum 3, da dies die einzige Möglichkeit ist, muss nun die Entscheidung zwischen Raum 1 und 4 getroffen werden.

In der Q-Werte Matrix finden sich zu diesem Zeitpunkt noch keine Werte wieder, also entscheidet sich der Agent zufällig, in diesem Beispiel für den Raum 1. Von diesem aus kann er in Raum 5 oder Raum 3 gelangen. Abermals sind keine Werte in der Matrix zu finden. Um diese Erläuterung kurz zu halten, gehen wir davon aus, dass der Agent sich für Raum 5 entscheidet. Dadurch erhält er eine Belohnung von 100. Diese wird gemäß der Formel in Gleichung (5.2) in die Matrix in Gleichung (5.1) eingetragen, zu sehen in Gleichung (5.3).

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{5.3}$$

Als Lernrate wird hier zur Vereinfachung ein Wert von $\alpha = 1$ und ein Gewichtungsfaktor von $\gamma = 0.8$ verwendet. Sollte der Agent nun erneut in den Raum 1 gelangen, so wird der zugehörige Q-Wert in der Matrix anhand der vorherigen erhaltenen Belohnung aktualisiert. Die erhaltene Belohnung r im Raum 1 entspricht 0, die erwartete Bewertung um von Raum 1 in Raum 5 zu gelangen wird mit einem Gewichtungsfaktor $\gamma = 0.8$

multipliziert, was zu einem neuen Ergebnis für den Q-Wert zu Raum 1 führt:

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 100 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 80 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \end{array} \tag{5.4}$$

Der Agent lernt anhand dieser Methode die optimale Aktion in einem gegebenen Zustand zu treffen, ohne von einem Trainer beaufsichtigt zu werden. Durch einen Zufallsfaktor wird garantiert, dass alle Zustände erforscht werden. Laut McCullock konvergiert die Matrix aus diesem Beispiel nach genügend Spielen wie in Gleichung (5.5)[28].

$$Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array} \begin{array}{c} \begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\ \begin{pmatrix} 0 & 0 & 0 & 0 & 80 & 0 \\ 0 & 0 & 0 & 64 & 0 & 100 \\ 0 & 0 & 0 & 64 & 0 & 0 \\ 0 & 80 & 0 & 0 & 80 & 0 \\ 64 & 0 & 51 & 64 & 0 & 100 \\ 0 & 80 & 0 & 0 & 80 & 100 \end{pmatrix} \end{array} \tag{5.5}$$

Es ist deutlich zu erkennen, dass die Aktionen, um in Raum 1 und Raum 4 zu gelangen, höher bewertet sind, als um in Raum 3 zu gelangen, da nur von diesen beiden Räumen Raum 5 als Ziel erreicht werden kann. Im Gegensatz dazu ist die Aktion um von Raum 1 in Raum 3 zu gelangen nur mit 64 bewertet.

5.2 Tic-Tac-Toe: Q-Table Implementierung

Nachdem Q-Learning an einem Beispiel mit einer sehr überschaubaren Anzahl an Zuständen erläutert wurde, folgt nun die Überführung auf ein größeres Beispiel mit der Umsetzung des Spiels Tic-Tac-Toe. Die Implementierung erfolgt dabei in der Programmiersprache Python. Dazu werden in Listing 5.1 zunächst die grundlegenden Variablen

```
Q = np.zeros([3 ** 9, 9])
learningrate = 0.001
gamma = 0.9
epsilon = 0.1
episodes = 1250000
```

Listing 5.1: Initialisierung grundlegender Variablen für Q-Learning

wie Lernrate, Gewichtungsfaktor, der Faktor für zufällige Erkundung, die Anzahl zu trainierender Episoden und die initiale Q-Tabelle festgelegt. Bei der Größe der Q-Werte Matrix werden dabei keine Optimierungen hinsichtlich unmöglicher Zustände vorgenommen. Entsprechend wird sie auf das in Gleichung (5.6) gezeigte Produkt aus möglichen Zuständen pro Feld potenziert mit der Anzahl Felder und der Anzahl maximal verfügbarer Aktionen initialisiert.

$$\text{Maximale Einträge der Matrix} = 3^{3 \cdot 3} \cdot 9 = 177\,147 \tag{5.6}$$

Ein weiterer wichtiger Aspekt ist das Festlegen von Belohnungen, die der Agent in entsprechenden Situationen erhält. Listing 5.2 zeigt dabei die gewählten Belohnungswerte für illegale, gewinnbringende, ein Unentschieden erzielende und verlierende Züge. Illegale und verlierende Züge werden negativ bewertet, um sie dem Agenten abzutrainieren. Gewinnbringende Züge und solche, die zu einem Unentschieden führen, werden positiv bewertet.

```
illegal_move_reward = -100
win_reward = 1
tie_reward = 0.5
loss_reward = -1
```

Listing 5.2: Festlegung von Belohnungen für Q-Learning

Um während des Ablaufs effizient auf einen Zustand in der Matrix zugreifen zu können, wird die Darstellung eines Spielzustands, wie in Listing 5.3 zu sehen, durch Multiplikation mit aufeinanderfolgenden Dreierpotenzen zu einer eindeutigen Zahl umgewandelt. Der in

```
index_conversion = np.array([3 ** 0, 3 ** 1, 3 ** 2, 3 ** 3, 3 ** 4, 3
↪  ** 5, 3 ** 6, 3 ** 7, 3 ** 8]).T


def stateindex(state):
  return np.dot(state, index_conversion)
```

Listing 5.3: Transformation eines Zustands in einen Arrayindex

Abb. 5.3 gezeigte Spielzustand wird dabei in die Zahl $1 \cdot 3^4 + 2 \cdot 3^5 + 1 \cdot 3^7 + 2 \cdot 3^8 = 15\,876$ umgewandelt.

Zur Durchführung des Trainings wird, wie in Listing 5.4 auszugsweise zu sehen, die festgelegte Anzahl Trainingsepisoden absolviert. Dazu wird in jeder Episode zunächst ein leeres Spielfeld angelegt. Anschließend wird für jeden Zug des Agenten bis zum Spielende entweder zufällig[18] ein verfügbarer Zug oder mit Hilfe der Q-Werte Matrix der aktuell am besten bewertete Zug ausgewählt. Ein Aufruf der `make_move(state, action)` Methode erzeugt automatisch eine gegnerische Aktion, sofern erforderlich, und gibt den neuen Spielzustand, eine eventuell erhaltene Belohnung und den Spielstatus zurück. Basierend auf diesen Informationen wird anschließend unter der Beachtung von Rotationen, Spiege-

[18] zur Steigerung der Erkundung verschiedener Spielrichtungen

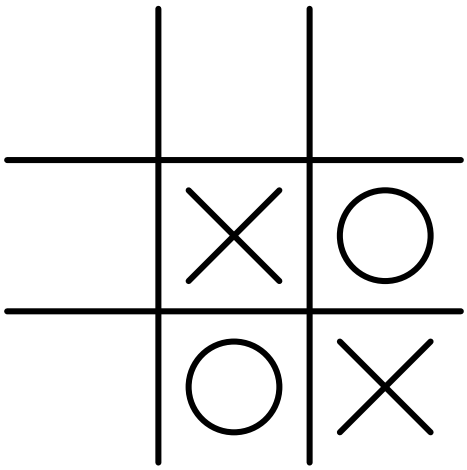

Abbildung 5.3: Ein Tic-Tac-Toe Spielzustand mit Index Repräsentation 15 876

lungen und der Vertauschung der Spieler das Training wie in Abschnitt 5.1 beschrieben vorgenommen.

Nachdem das Training abgeschlossen ist, startet das Programm ein interaktives Tic-Tac-Toe Spiel, dessen Züge auf der trainierten Q-Werte Matrix aufsetzen. Abb. 5.4 zeigt den Ablauf eines solchen Spiels, in dem der Computergegner mit O gegen den menschlichen Spieler mit X spielt und ihn nach einer Fehlentscheidung unter Nutzung einer Falle schlägt.

```
for i in range(1, episodes + 1):
  state = new_board()

  done = False
  while not done:
    action = np.random.choice(np.where(state == 0)[0] if
    ↪  np.random.rand() < epsilon else np.where(Q[stateindex(state), :]
    ↪  == np.max(Q[stateindex(state), :]))[0], 1)
    new_state, reward, done = make_move(state, action)
    max_q = np.max(Q[stateindex(new_state), :])
    Q[stateindex(state), action] = (1 - learningrate) *
    ↪  Q[stateindex(state), action] + learningrate * (reward + gamma *
    ↪  max_q)
    r, c = action // 3, action % 3
    state_90 = np.reshape(np.rot90(np.reshape(state, [3, 3])), [9])
    rot_90_action = 6 - 3 * c + r
    Q[stateindex(state_90), rot_90_action] = (1 - learningrate) *
    ↪  Q[stateindex(state_90), rot_90_action] + learningrate * (reward
    ↪  + gamma * max_q)
```

Listing 5.4: Q-Learning: Durchführung des Trainings

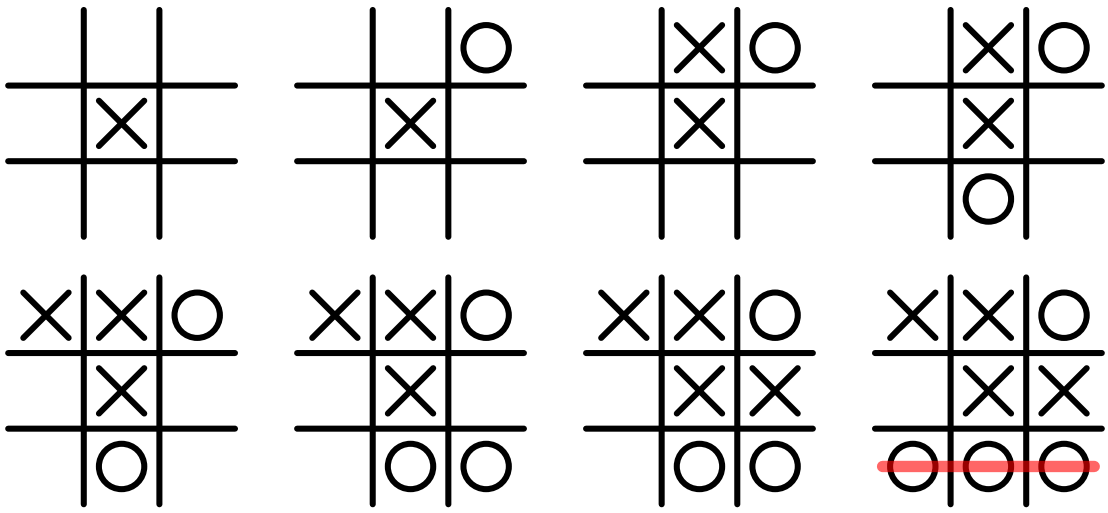

Abbildung 5.4: Tic-Tac-Toe Spiel gegen Q-Werte Matrix gesteuerten Computergegner

6 Kombination von Q-Learning und neuronalen Netzen

In diesem Kapitel werden die bereits vorgestellten Methoden und Technologien des Q-Learning mit neuronalen Netzen verbunden. Es wird insbesondere auf die Art des Lernens von neuronalen Netzen bei der Verwendung von Q-Learning eingegangen. Außerdem wird das Grundprinzip des ersten Ansatzes des im Zuge dieser Arbeit entwickelten Deep Q Network für das Spiel Vier gewinnt vorgestellt.

6.1 Q-Learning in neuronalen Netzen: Backpropagation

Während bei einem Spiel wie Tic-Tac-Toe Q-Learning noch ohne weiteres mit klassischen Matrizen zur Speicherung der Q-Werte eines Zustands gearbeitet werden kann, gerät diese Vorgehensweise bei Vier gewinnt bereits an ihre Grenzen.

Ohne Beachtung von unerreichbaren Zuständen, frühzeitig beendeten Spielen oder Variationen des selben Zustands durch Spiegelungen und Drehungen benötigt man für beide Spiele maximal folgende Tabellengröße:

- TicTacToe: $3^{3 \cdot 3} \cdot 9 = 177\,147$ Einträge
- Vier gewinnt: $3^{6 \cdot 7} \cdot 7 = 765\,932\,923\,920\,586\,514\,463$ Einträge

Selbst wenn man diese Menge auf die tatsächliche Anzahl erreichbarer Zustände reduziert, so bleiben für Vier gewinnt dennoch $4\,531\,985\,219\,092$ legale Spielzustände[29]. Zieht man weiterhin klare Gewinn- und Unentschieden-Zustände[12] ab, so bleiben immer noch $2\,626\,684\,783\,614$ Zustände übrig. Es wären $18\,386\,793\,485\,298$ Matrixeinträge nötig, um die entsprechenden Q-Werte zu speichern.

Statt diesem simplen matrixbasierten Ansatz kombiniert man nun die Methodik von Q-Learning mit der Eigenschaft neuronaler Netze, durch Training Funktionen zumindest annähernd abbilden zu können. Dazu erstellt man ein neuronales Netz, dessen

Eingabedaten durch eine Abbildung des aktuellen Spielzustands gebildet werden. Wie Abb. 6.1 zeigt, entsteht durch die Verrechnung mittels internen Gewichtungen der Verknüpfungen zwischen einzelnen Ebenen des neuronalen Netzes in der Ausgabeschicht eine Reihe von Werten, welche den Q-Werten für die in diesem Zustand durchführbaren Züge entsprechen[30].

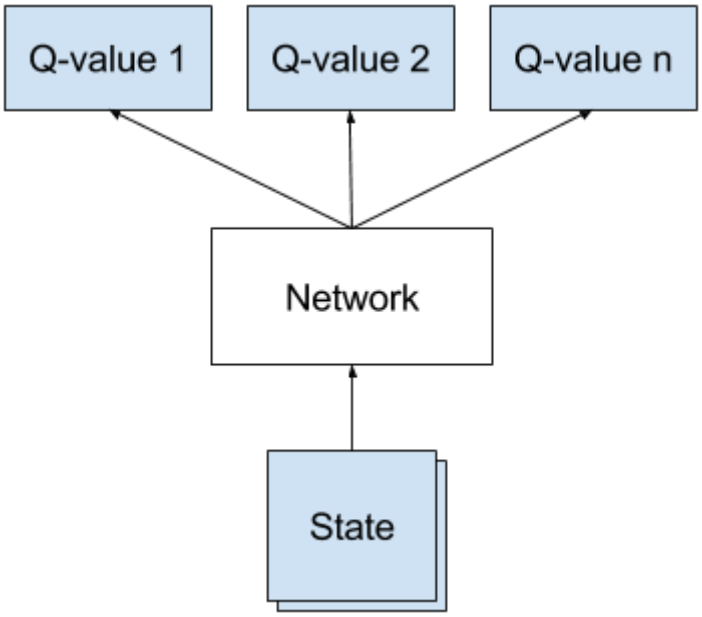

Abbildung 6.1: Optimierte Architektur eines Deep Q-Netzwerks (aus [24])

Um dieses Netz zu trainieren, wird die Methode der Backpropagation, manchmal auch detaillierter Backpropagation of Error[19] genannt, verwendet. Bei dieser Methode handelt es sich um ein Verfahren, die Gewichtungen innerhalb eines neuronalen Netzes aufgrund des am Ausgang auftretenden Fehlers zu optimieren. Ziel der Backpropagation ist es, einem mehrschichtigen neuronalen Netz die nötigen internen Gewichtungen anzutrainieren, sodass eine im Grunde beliebige Abbildung von Eingabe- auf Ausgabewerte möglich

[19] dt.: Fehlerrückführung

ist[4]. Im Folgenden wird dieses Verfahren auf Basis der in [4] gezeigten Formeln und Erläuterungen kurz dargelegt, um ein Grundlegendes Verständnis zu schaffen.

Das Trainingsziel wird über die Minimierung einer sogenannten Fehlerfunktion erreicht. Als Fehlerfunktion eignet sich beispielsweise der quadratische Fehler zwischen erzielter und erwarteter Ausgabe. Die Fehlerfunktion, manchmal im Englischen auch als *Cost Function* oder *Loss Function* bezeichnet, ist hier dabei wie folgt definiert:

$$E = \frac{1}{2} \sum_{c} \sum_{j} (o_{j,c} - d_{j,c})^2 \tag{6.1}$$

Dabei handelt sich bei E um den resultierenden quadratischen Gesamtfehler, bei d um den erwünschten und bei o um den tatsächlichen Ausgabewert. Der Index c identifiziert dabei einzelne Eingabe-Ausgabe Paare, der Index j einzelne Neuronen des Netzwerks. Der Faktor $\frac{1}{2}$ dient zur einfacheren Ableitung in folgenden Schritten. Da später eine Multiplikation mit einem frei wählbaren konstanten Faktor, der Lernrate, vorgesehen ist, hat der hier hinzukommende Faktor keinen Einfluss auf das letztendliche Ergebnis.

Mittels partieller Ableitung und unter Anwendung der Kettenregel lassen sich die Auswirkungen einzelner Gewichtungen auf die Berechnung der Ausgabewerte errechnen. Die partielle Ableitung von Gleichung (6.1) nach o unter der Betrachtung eines spezifischen Eingabe-Ausgabe Paares lautet:

$$\frac{\delta E}{\delta o_j} = o_j - d_j \tag{6.2}$$

Betrachtet man nun den Aufbau eines Neurons wie in Abb. 3.4, erlaubt die wiederholte Anwendung der Kettenregel auf Gleichung (6.2) folgende letztendliche Schlussfolgerung:

$$\frac{\delta E}{\delta w_{ij}} = \delta_j o_i \tag{6.3}$$

mit

$$\delta_j = \frac{\delta E}{\delta o_j}\frac{\delta o_j}{\delta net_j} = \begin{cases} \varphi'(net_j)(o_j - d_j) & \text{wenn } j \text{ ein Ausgabeschicht-Neuron identifiziert,} \\ \varphi'(net_j)\sum_k \delta_k w_{jk} & \text{wenn } j \text{ ein inneres Neuron identifiziert.} \end{cases} \tag{6.4}$$

Dabei identifizierten i, j und k in Gleichungen (6.3) und (6.4) Neuronen des Netzes in aufeinanderfolgenden Schichten.

Unter Verwendung der Lernrate α erfolgt die letztendliche Berechnung der Gewichtungsänderung folgendermaßen:

$$\Delta w_{ij} = -\alpha \delta_j o_i$$

Für Neuronen, die direkt an den Eingängen des neuronalen Netzes liegen, ist o_i entsprechend ein Eingabewert.

6.2 Training durch Selfplay

Die Nutzung von Backpropagation fordert einen erwarteten Referenzwert, der beim überwachten Lernen durch einen Algorithmus wie Minimax oder Negamax während der Laufzeit oder auch durch vorgegebene Trainingsbeispiele erfolgt. Dies erfordert jedoch zusätzliche, bei potenziell bis zum Ende untersuchten längeren Spielen, teils enorme Rechenzeit während des Trainingsvorgangs oder eine intelligente, meist zu einem gewissen Grad von Hand erfolgte Zusammenstellung von Trainingsmaterial.

Das Training eines neuronalen Netzes durch Selfplay[20], am Beispiel dieser Arbeit das wechselseitige Spielen von TicTacToe oder Vier gewinnt, erfordert hingegen deutlich weniger Aufwand im Voraus, bringt jedoch eine verlängerte Trainingszeit mit sich. Die

[20] Selbständiges Training ohne äußere Einflüsse

Erzeugung der Referenzwerte für das Training erfolgt nach folgendem Algorithmus[24] zur Laufzeit:

Gegeben sei eine Spieltransition der Form (s, a, r, s').

1. Durchführung eines vorwärts propagierenden Durchlaufs des Netzes mit aktuellem Zustand s zur Q-Wert Vorhersage

2. Durchführung eines vorwärts propagierenden Durchlaufs des Netzes mit Folgezustand s' und Auswahl des größten vorausgesagten Q-Wertes $\max_{a'} Q(s', a')$ für die Transition (s', a', r', s'')

3. Setzen des Referenz-Q-Wertes für Aktion a auf $r+\gamma \max_{a'} Q(s', a')$. Für alle anderen Aktionen wird der vorhergesagte Q-Wert aus Schritt 1 verwendet, um einen Fehler von 0 für diese Ausgänge zu erhalten

4. Aktualisierung der Gewichtungen mittels Backpropagation im Netz

Ein auf diese Weise gegen selbst spielendes Netz wurde bereits 1995 erfolgreich eingesetzt[31]. Auch in kürzerer Vergangenheit zeigte sich 2015 im DeepMind Paper in Verbindung mit weiteren Verbesserungsmaßnahmen, wie in Abschnitt 7 beschrieben, große Wirkung[30] bei der Verwendung dieses Algorithmus.

6.3 Vier gewinnt: Deep Q Network Implementierung

Da Vier gewinnt im Vergleich zu Tic-Tac-Toe einen deutlich größeren Zustandsraum besitzt, wie in Abschnitt 6.1 gezeigt, reicht ein einfacher Q-Learning Ansatz mit einer Zustandsbewertungsmatrix nicht mehr aus. Stattdessen wird eine Implementierung auf Basis von neuronalen Netzen nach dem Vorbild der sogenannten Deep Q Networks [30] vorgenommen. Im Gegensatz zur referenzierten Implementierung in [30] setzt dieser

Ansatz zunächst auf die einfache Verarbeitung der verhältnismäßig überschaubaren Eingabevektoren.

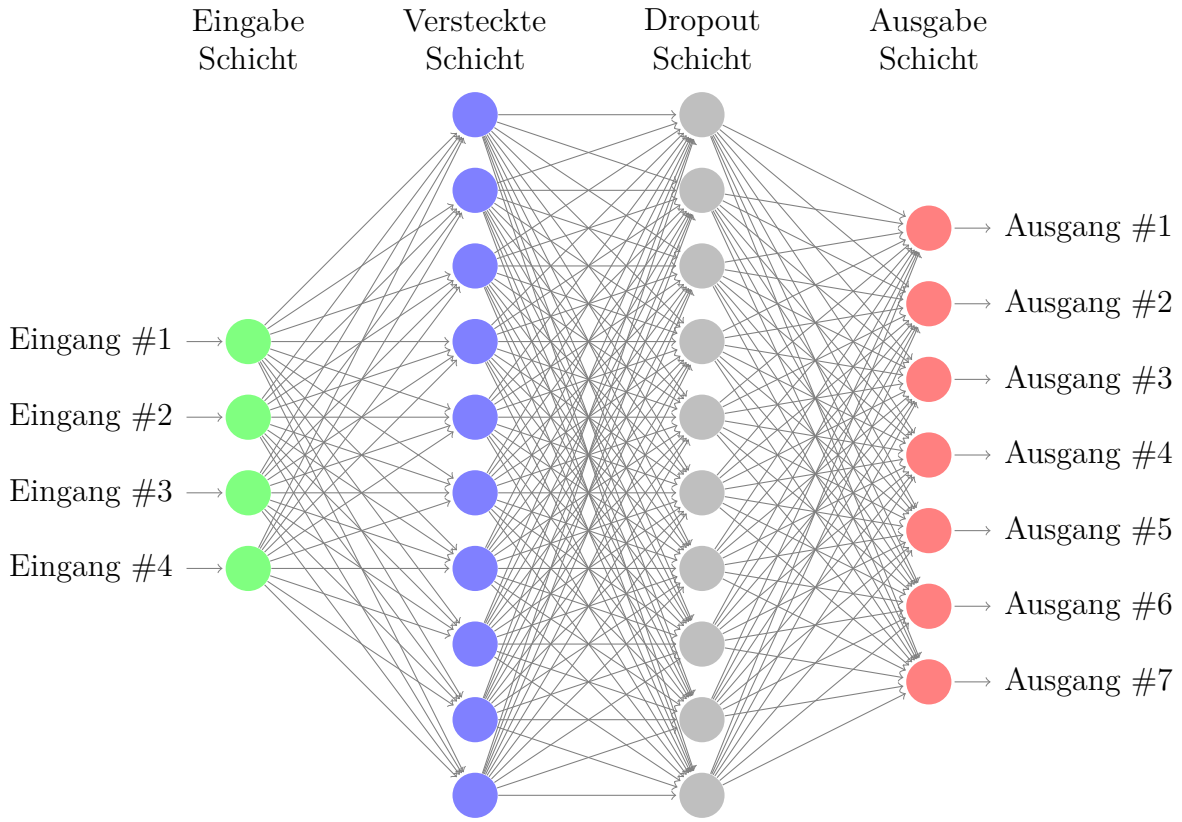

Abbildung 6.2: Beispielhafter Aufbau eines neuronalen Netzes mit mehreren Schichten

Das Netzwerk wird dabei grundlegend wie in Abb. 6.2 aufgebaut. In der Eingangsschicht wird der aktuelle Spielzustand eingespeist. Im Inneren des Netzwerks befinden sich ein oder mehrere Paare bestehend aus je einer herkömmlichen, voll verknüpften, versteckten Schicht und einer Dropout Schicht. Letztere trägt ihren Namen durch die Eigenschaft, dass während des normalen Durchlaufs Neuronen in dieser Art von Schicht mit einer frei wählbaren Wahrscheinlichkeit für diesen Durchlauf nicht aktiv werden. Dies ist eine der Möglichkeiten das sogenannte Overfitting[21] von neuronalen Netzen zu verhindern[32]. In

[21] dt.: Überanpassung

der Ausgangsschicht werden letztendlich die resultierenden Ausgabewerte abgegriffen. Im Falle unseres Netzes handelt es sich dabei um die approximierten Q-Werte der Aktionen für den eingespeisten Spielzustand.

Ähnlich der einfach gehaltenen Matrix Implementierung für Tic-Tac-Toe in Abschnitt 5.2 sind auch hier grundlegende Einstellungen zu treffen, wie in Listing 6.1 zu sehen. Ebenfalls erkennbar ist, dass die hier auszugsweise gezeigte Implementierung in der Lage ist, verschiedene Gitter-basierte Spiele wie beispielsweise Tic-Tac-Toe, Vier gewinnt und Fünf gewinnt zu verwenden.

```
# TicTacToeGame, ConnectFourGame or FiveInARowGame
game = ConnectFourGame()
# size of hidden layers
network_hidden_layers = [1024, 512, 256, 128]
# number of learning episodes
num_episodes = 10e6
# Q learning gamma value
gamma = 0.9
# learning rate of neural network
learning_rate = 0.0001
```

Listing 6.1: Grundlegende Konfigurationsparameter des neuronalen Netzes

Zur Implementierung des neuronalen Netzes werden die Programmiersprache Python und das Framework Tensorflow verwendet. Ein neuronales Netz, wie in Abb. 6.2 beispielhaft dargestellt, lässt sich dadurch mit recht geringem Aufwand über die in Listing 6.2 gezeigten Aufrufe aufbauen.

Sind das neuronale Netz angelegt und die nötige Implementierung des Gitter-basierten Spiels vorhanden, so kann der Trainingsvorgang beginnen. Dazu wird in der auszugsweise in Listing 6.3 gezeigten Spielschleife für jeden Zug eine Aktion durch einen Aktionsselektor ausgewählt und im Anschluss der Lernprozess mit den resultierenden Werten angestoßen.

Die Auswahl eines Zuges durch einen Aktionsselektor ermöglicht, verschiedene Auswahlverhalten sauber getrennt von der Spiellogik zu behandeln. Zur Verfügung stehen in der hier verwendeten Implementierungen drei Aktionsselektoren:

- ein greedy Aktionsselektor
- ein ϵ-greedy Aktionsselektor
- ein Boltzmann-Wahrscheinlichkeit Aktionsselektor

Diese drei Aktionsslektoren handeln nach dem Prinzip von ”exploration”, zu Deutsch Erkundung, und ”exploitation”, zu Deutsch Ausnutzung. Mit Blick auf die Auswahl von Zügen versteht sich darunter ein Abwiegen zwischen der meist zufälligen Erkundung und der auf gesammeltem Wissen basierenden Ausnutzung.

Das Auswahlverhalten des greedy, zu Deutsch gierigen, Aktionsselektors resultiert immer in der Auswahl des aktuell am besten eingeschätzten Zuges und handelt demnach komplett nach dem Prinzip der Ausnutzung. Der ϵ-greedy Aktionsselektor, dessen Implementierung in Listing 6.4 zu sehen ist, wählt anhand eines konfigurierbaren Prozentsatzes einen zufälligen zulässigen Zug. Dies fördert Erkundung von ansonsten potenziell nicht beachteten Spielrichtungen, die dem Agenten zu neuem Wissen verhelfen können. Wird kein zufälliger Zug gewählt fällt die Auswahl zurück auf das Verhalten des gierigen Aktionsselektors. Der Boltzmann-Wahrscheinlichkeit Aktionsselektor stellt eine intelligente Verknüpfung aus Erkundung und Ausnutzung dar. Dabei werden Aktionen aufgrund gewichteter Wahrscheinlichkeiten gewählt. Der größte Vorteil gegenüber des ϵ-greedy Aktionsselektors ist, dass auch Informationen über Aktionen, die weniger optimal eingeschätzt werden, teil der Entscheidungsfindung sind.

Nachdem die festgelegte Anzahl Trainingsepisoden absolviert ist, wird einem Spieler eine interaktive Möglichkeit gegen das Netz zu spielen geboten, wie bereits bei der auf einer Q-Werte Matrix basierenden Implementierung von Tic-Tac-Toe in Abschnitt 5.2.

```
def __init__(self, scope, game: GridBasedGame, hidden_layers) -> None:
  # state input, shape: [batch, players, rows, columns]
  self.state_in = tf.placeholder(tf.float32, [None, 2, game.rows,
  ↪  game.columns], "state_in")
  # flatten input
  self.neural_network = tf.reshape(self.state_in, [-1, 2 * game.rows *
  ↪  game.columns])
  with tf.variable_scope(scope):
  self.scope = tf.get_variable_scope().name

  self.keep_prob = tf.placeholder(tf.float32, ())
  # create hidden layers
  for layer_size in hidden_layers:
    self.neural_network = layers.fully_connected(self.neural_network,
    ↪  layer_size, tf.nn.relu)
    self.neural_network = layers.dropout(self.neural_network,
    ↪  self.keep_prob)

  # q values output
  self.q_out = layers.dense(inputs=self.neural_network,
  ↪  units=game.num_actions)

  # input for action vectors
  self.actions_in = tf.placeholder(tf.float32, [None, game.num_actions],
  ↪  "actions_in")
  # input for expected q values of the selected action
  self.target_q_in = tf.placeholder(tf.float32, [None], "target_q_in")
  # get the current q values of the selected action
  self.current_action_q_values = tf.reduce_sum(tf.multiply(self.q_out,
  ↪  self.actions_in), [1])
  # calculated mean square loss
  self.loss = tf.reduce_mean(tf.square(self.target_q_in -
  ↪  self.current_action_q_values))
  self.learning_rate = tf.placeholder(tf.float32, ())
  self.optimizer = tf.train.AdamOptimizer(self.learning_rate)
  self.updater = self.optimizer.minimize(self.loss)
```

Listing 6.2: Erstellung eines neuronalen Netzes unter Verwendung von Tensorflow

```
while training_episode < num_episodes:
  training_episode += 1
state = game.new_game()
  while not game.done:
    action = exploration_action_selector.select_action(state,
    ↪  game.get_valid_actions(), main_network, learn_dropout_keep_prob,
    ↪  random_factor)
    state_after_player_action, reward, done = game.step(action)

    learn(state, action, reward, done)
```

Listing 6.3: Spielschleife des Trainingsvorgangs mit automatisierter Zugauswahl

```
class EpsilonGreedyActionSelector(ActionSelector):
  """randomly selects either a random or the best action according to
  ↪  the network"""
  def __init__(self) -> None:
    self.greedy_action_selector = GreedyActionSelector()

  def select_action(self, state, valid_action_indices, network,
  ↪  dropout_keep_prob, random_factor):
    if np.random.random() < random_factor:
      return valid_action_indices[np.random.randint(len(
      ↪  valid_action_indices))]
    else:
      return self.greedy_action_selector.select_action(state,
      ↪  valid_action_indices, network, dropout_keep_prob,
      ↪  random_factor)
```

Listing 6.4: Implementierung eines ϵ-greedy Aktionsselektors

7 Einsatz von Verbesserungsmaßnahmen

In diesem Kapitel wird auf die im Zuge dieser Arbeit angewendeten Verbesserungsmaßnahmen des neuronalen Netzes eingegangen. Der Schwerpunkt der Analyse liegt auf den Themen Experience Replay, Double Deep Q Network und Dueling Deep Q Network.

7.1 Experience Replay

Um bei komplett unbeaufsichtigtem Training der neuronalen Netze gute Fortschritte zu erzielen, sind Verbesserungsmaßnahmen nötig.

Bei der ersten umgesetzten Verbesserungsmaßnahme handelt es sich um das sogenannte Experience Replay[22]. Während ohne diese Technik das Training bereits im Gange der gespielten Episoden durchgeführt wird, verzögert Experience Replay diesen Schritt. Ein Erfahrungstupel nimmt dabei folgende Form an:

$$(s, a, s', r)$$

Es besteht aus dem aktuellen Zustand s, der vom Agenten gewählten Aktion a, dem darauf folgend Zustand s' und der Belohnung r, welche aus der gewählten Aktion resultiert. Der Folge-Zustand s' kann dabei bereits eine Reaktion eines eventuell vorhandenen Spielgegners beinhalten, je nach Simulationsumgebung[33]. Im Falle eines Nullsummenspiels wie Tic-Tac-Toe oder Vier gewinnt ist dies der Fall, da der Agent nur eine sinnvolle Reaktion auf einen erfolgten gegnerischen Zug erlernen kann.

Während des Spiels werden erzeugte Erfahrungstupel in einem begrenzten Puffer gespeichert. Neuere Erfahrungen verdrängen dabei mit der Zeit ältere. Dies verhindert sowohl das wiederholte Lernen derselben Erfahrungen, als auch mögliche Rückschritte

[22] dt.: Erfahrungswiedergabe

durch das Lernen von älteren gesammelten Erfahrungen, die auf Aktionen basieren, welche von einem weniger trainierten Stand des Netzes gewählt wurden.

Bei einfachem Experience Replay erfolgt die Auswahl der zu verarbeitenden Erfahrungen zufällig aus dem Puffer. Durch die Wahl verteilter, unabhängiger Erfahrungen wird ein Overfitting auf Spielsituationen von direkt zusammenhängenden Aktionen verhindert. Je größer der Zustandsraum eines Spiels jedoch ist, desto unwahrscheinlicher ist es auch, dass Erfahrungen mit belohnungsbringenden Aktionen antrainiert werden. Eine Erweiterung des Experience Replays besteht daher in der gezielten, gewichteten Auswahl von gesammelten Erfahrungen mit einer erhaltenen Belohnung, um schneller profitable und zu vermeidende Spielzweige anzutrainieren[34].

Die in dieser Arbeit verwendete Implementierung stellt eine Vereinfachung der gezeigten Erweiterung dar, da ein Priorisieren der Erfahrungen mittels zweier Puffer und der Trennung auf Basis einer vorhandenen Belohnung umgesetzt wurde.

7.2 Double Deep Q Network

Eine weitere Verbesserungsmaßnahme ist die Implementierung eines sogenannten Double Deep Q Networks. Der Grundgedanke hinter dieser Maßnahme ist die Trennung der Auswahl einer durchzuführenden Aktion von ihrer Bewertung. Bei einem üblichen, einfachen Deep Q Network finden sowohl Auswahl als auch Bewertung der durchzuführenden Aktion in einem Zustand durch das selbe Netzwerk statt. Dies kann jedoch zu Überbewertungen von Aktionen führen. Kombiniert mit unglücklicher Verbreitung dieser zu optimistischen Bewertung im Netz resultiert diese anfängliche Überbewertung in einer allgemein schlechteren Einschätzung[35]. Zur Trennung von Auswahl und Bewertung wird daher ein zweites neuronales Netz, das sogenannte Target Network, zur Bewertung ausgewählter Aktionen verwendet.

Bei einem Target Network wird das Hauptnetz strukturell kopiert, die Gewichtungen innerhalb des Netzes jedoch nicht direkt beim Training angepasst. Je nach genauer Implementierung werden die Gewichtungen innerhalb des Target Network entweder periodisch durch die des Hauptnetzes ersetzt oder mit der Zeit in kleinen Schritten an die Gewichtungen im Hauptnetz angenähert. In dieser Arbeit wurde letzteres Verfahren verwendet.

Durch diese Änderung wird die Bewertung von gewählten Aktionen resistent gegen schnelle Schwankungen der Gewichtungen im Hauptnetz, reduziert generell die Überbewertung von Aktionen und führt insgesamt zu einer besseren Leistung des Netzes.

7.3 Dueling Deep Q Network

Neben der Trennung von Auswahl und Bewertung einer Aktion in einem Zustand durch die Nutzung von Double Deep Q Networks gibt es einen weiteren Punkt, an dem eine Aufspaltung zu Verbesserungen führen kann. Die sogenannten Dueling Deep Q Networks setzen dabei im Vergleich zu Double Deep Q Networks innerhalb des Netzes an. Der Ansatzpunkt liegt dabei, wie in Abb. 7.1 zu sehen, direkt vor der Berechnung der Q-Werte für die ausführbaren Aktionen in einem Zustand.

Der Hintergedanke dieser Verbesserungsmaßnahme ist die Aufspaltung der Bewertung von Zustand und getätigter Aktion in zwei zu verrechnende Werte: eine Bewertung des aktuellen Zustands und eine Bewertung des Vorteils, der durch die Ausführung einer Aktion entsteht. Am Beispiel des Atari Spiels Enduro lässt sich das Prinzip verdeutlichen. Wie in Abb. 7.2 zu sehen, fokussiert sich die Bewertung des Zustands stark auf den Horizont, an dem neue vorausfahrende Autos erscheinen, auf direkt vorausfahrende Fahrzeuge und den aktuellen Punktestand. Die Vorteilsbewertung hingegen fokussiert sich praktisch ausschließlich auf direkt vorausfahrende Autos, da gewählte Aktionen quasi nur in Situationen mit einem direkt vorausfahrenden Auto relevant sind. Formuliert man

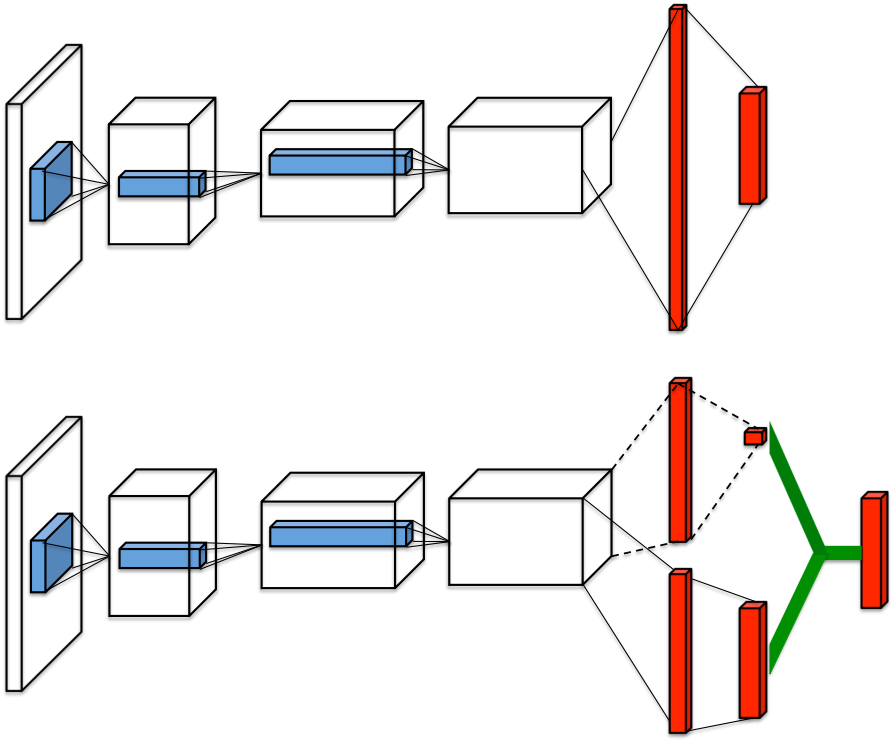

Abbildung 7.1: Darstellung eines einströmigen und eines Dueling Netzwerkes

Ein einfaches einströmiges Q-Netz (oben) und ein Dueling Deep Q Network (unten). Grün Hervorgehoben ist die Umsetzung von Gleichung (7.2) (aus [36])

diese Vorstellung eines zusammengesetzten Q-Values direkt als Gleichung, erhält man:

$$Q(s,a) = V(s) + A(s,a) \tag{7.1}$$

Die entstehenden Q-Values $Q(s,a)$ bilden dabei die Summe aus einer skalaren Wertungsfunktion des Zustands $V(s)$ und einer Vorteilsbewertung einzelner Aktionen in diesem Zustand $A(s,a)$.

Gleichung (7.1) erscheint eventuell auf den ersten Blick als sinnvoll, bringt jedoch noch nicht den gewünschten Effekt, da kein eindeutiges Rückschließen auf den zugrunde liegenden Zustand möglich ist. Aus einer nötigen Kompensationen der Vorteilsbewertung

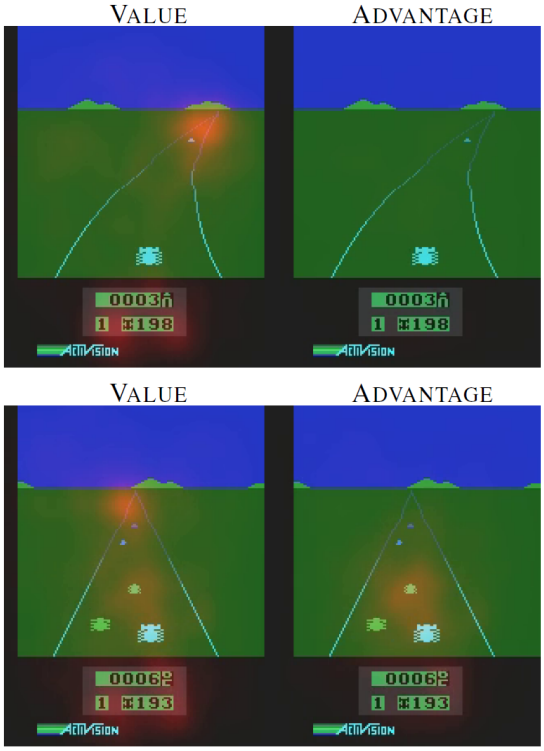

Abbildung 7.2: Visualisierung von Zustands- und Vorteilsbewertung

Visualisierte Wichtigkeit von Pixelregionen (rötliche Hervorhebung) für Zustandsbewertung (links) und Vorteilsbewertung (rechts) für ein trainiertes Dueling Deep Q Network in zwei verschiedenen Spielzuständen (aus [36])

und einer Anpassung zur Stabilisierung der Optimierung des Netzes[36] resultiert die folgende angepasste Formel:

$$Q(s,a) = V(s) + (A(s,a) - \frac{1}{|A|}\sum_{a'} A(s,a')) \tag{7.2}$$

Da diese Änderungen Teil des Netzwerkes sind, ist keinerlei Anpassung des restlichen Vorgehens nötig. Wie bei einem üblichen einfachen Q-Learning Netz kann weiterhin Training des Netzes mittels Backpropagation durchgeführt werden.

7.4 Kontrollierte Netzentwicklung durch periodischen Vergleich

Eine weitere Maßnahme zur Verbesserung der erzielten Ergebnisse ist die Verwaltung von zwei oder mehr parallel geführten neuronalen Netzen. Dabei ist ein Netz mit seinen Gewichtungen das bisher beste Netz, während andere aktiv trainiert werden. Nach einer festgelegten Anzahl Episoden tritt ein trainiertes Netz gegen das bisher beste bekannte Netz an. Schlägt das trainierte Netz sein Gegenüber, so werden die Gewichtungen des trainierten Netzes als neue Bestwerte übernommen. Ist keine ausschlaggebende Verbesserung des trainierten Netzes zu sehen, so werden die Gewichtungen verworfen und erneut von der besten bekannten Basis trainiert. Dies sorgt dafür, dass stets auf Basis des bisher besten erreichten Ergebnis gearbeitet wird. Die in dieser Arbeit verwendete Implementierung stellt eine starke Vereinfachung des Evaluations-Schritts des AlphaGo Zero Trainings[37] dar.

Im späteren Verlauf der Arbeit wurde das direkte Spielen gegeneinander durch die Überprüfung der Reaktion auf vorgegebene Spielzustände ergänzt. Dadurch lässt sich messen, wie gut den Netzen das korrekte Spielen in Fällen eines bevorstehenden Gewinns oder Verlustes gelingt.

8 Ergebnis

Der Einstieg in Neuronale Netzwerke zum Spielen von Computerspielen war dank der vorhandenen Ressourcen und den in den letzten Jahren gewonnenen Erkenntnissen schnell gelungen. Bereits in einer frühen Version des Netzwerkes, welche von keiner der Verbesserungsmaßnahmen aus Abschnitt 7 profitierte, konnten gute Ergebnisse in Tic-Tac-Toe verzeichnet werden. Selbst bei perfekter Spielweise eines menschlichen Spielers nach [10] konnte die selbstlernende künstliche Intelligenz nicht besiegt werden.

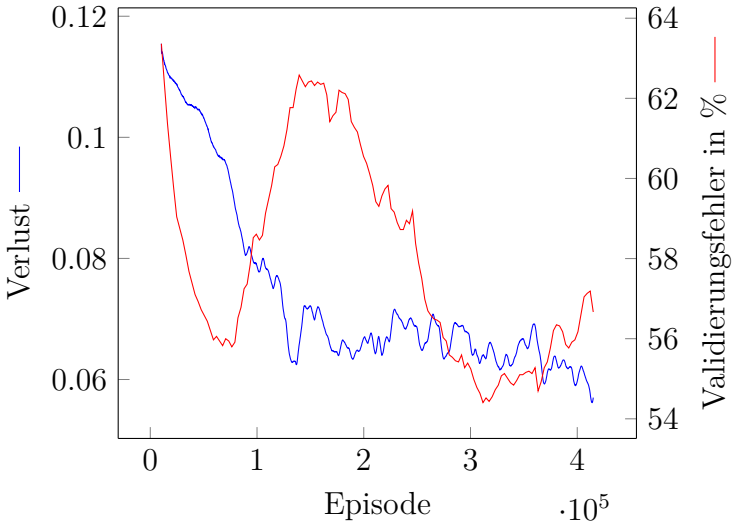

Abbildung 8.1: Verlust und Validierungsfehler in Vier gewinnt

Der Umstieg auf das deutlich komplexere Spiel Vier gewinnt zeigte jedoch, dass der weitaus größere Zustandsraum nicht ohne weiteres zu bewältigen ist. Dank der beschriebenen Verbesserungen stellt das antrainierte Netzwerk einen starken Gegner dar, welcher je nach Situation Gewinnmöglichkeiten ausnutzt und die des Gegners blockiert. Es stellte sich jedoch heraus, dass es äußert schwierig ist, diese subjektive Bewertung des Netzwerkes in objektiven Zahlen zu fassen. In Abb. 8.1 sind der Verlust und der Fehler auf ein eigens erstelltes Validierungsset über einen Trainingslauf zu sehen. Während der

Verlust tendenziell über die Zeit sinkt, fluktuiert der Validierungsfehler erheblich, was es schwierig macht, die tatsächliche Qualität des Netzwerkes zu bewerten. Des Weiteren ist zu sehen, dass die subjektive Einschätzung nicht mit der Validierung übereinstimmt.

Auch bei Tic-Tac-Toe konnte ein ähnliches Verhalten festgestellt werden. Zur Validierung konnte dank des stark begrenzten Zustandsraums auf ein größeres Validierungsset[23] gesetzt werden, was eine bessere Bewertung des Netzwerkes versprach. Obwohl unterschiedliche Einstellungen, wie die Methode der Zugauswahl (siehe Abb. 8.2) oder Größe des Netzwerkes (siehe Abb. 8.3), unterschiedliche Ergebnisse liefern, ist deutlich zu sehen, dass kein Zusammenhang zwischen Validierungsfehler und Verlust hergestellt werden kann. Auch hier stimmt die subjektive Einschätzung nicht mit der des Validierungssets überein; trotz der scheinbar perfekten Spielweise der künstlichen Intelligenz bei Tic-Tac-Toe weist es einen Fehler auf, weshalb sich diese Methode der Validierung als ungeeignet herausstellte.

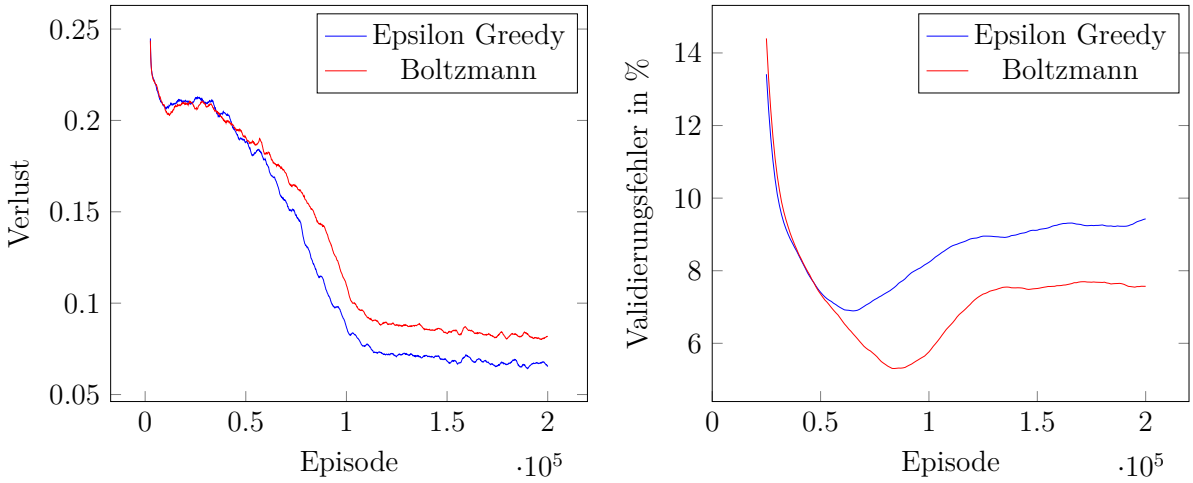

Abbildung 8.2: Vergleich zweier Zugauswahlstrategien in Tic-Tac-Toe

[23] http://www.connellybarnes.com/work/class/2016/deep_learning_graphics/proj1/

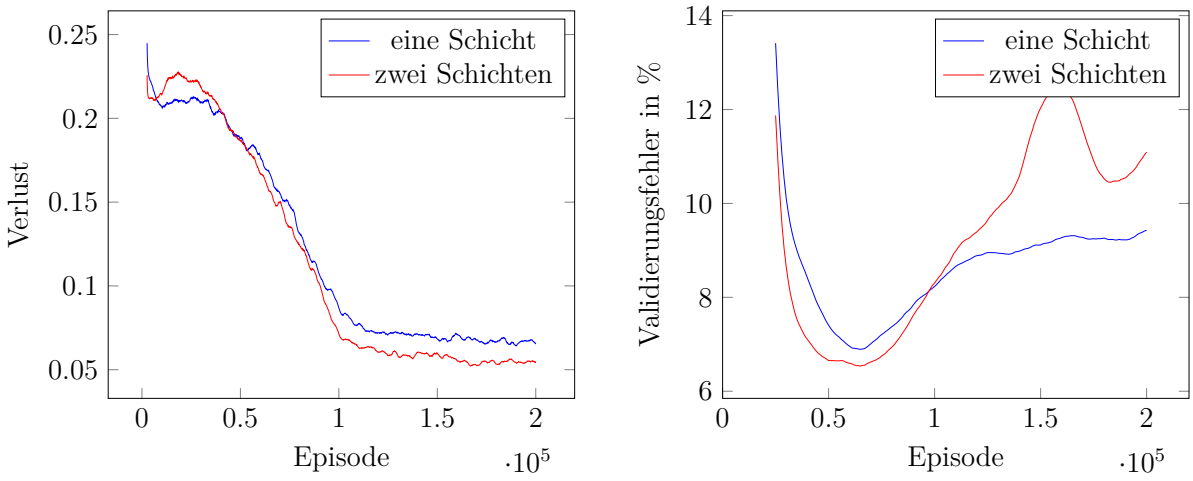

Abbildung 8.3: Vergleich zwischen ein und zwei versteckten Schichten in Tic-Tac-Toe

Obwohl die Qualität der Netzwerke nicht bewiesen wurde, konnte eine solide Basis für aufbauende Arbeiten dieser Art erstellt werden. Die erfolgreichsten wissenschaftlichen Erkenntnisse aus dem Bereich des Spielens von Computerspielen mit Neuronalen Netzwerken sind in den verwendeten und beigelegten Quelltext eingeflossen. Änderungen an den Lernparametern und die Wahl einer geeigneteren Validierungsstrategie sind daher erfolgversprechend. Dank des flexiblen Aufbaus können beliebige weitere Spiele ohne große Aufwände implementiert und angelernt werden, was Aussichten auf vielversprechende aufbauende Arbeiten erlaubt.

9 Ausblick

In diesem Kapitel wird auf die Zukunft und Fortführung dieser Studienarbeit eingegangen. Es werden diverse Vorschläge zur Verbesserung des entwickelten neuronalen Netzes vorgestellt und einige potentiell relevante Technologien kurz erläutert.

9.1 Verbesserte Zugauswahl: Monte Carlo Tree Search

Nachdem in dieser Arbeit bereits einige Verbesserungsmaßnahmen umgesetzt wurden, wird in diesem Abschnitt eine weitere mögliche Verbesserungsmaßnahme etwas genauer begutachtet. Bei dieser Verbesserung handelt es sich in der ersten Variante um den Einsatz eines Monte Carlo Suchalgorithmus[38] zur verbesserten Auswahl von zufälligen Zügen bei einer ϵ-gierigen Zugauswahl.

Eine übliche ϵ-greedy Auswahl wählt komplett zufällig einen der möglichen Züge aus. Bei der Verwendung einer Monte Carlo Search wird die Wahl des Zuges durch zeitlich begrenztes Weiterspielen mittels der letzten beiden Schritte einer Monte Carlo Tree Search, ”playout” und ”backup”, umgesetzt[39]. Der Zug mit dem besten nachfolgenden Spiel wird letztendlich ausgewählt.

Die zweite Variante ist von Q-Learning auf eine Kombination eines Reinforcement Learning Netzes mit einer Monte Carlo Tree Search umzusteigen. Dabei wird die Zugauswahl komplett über eine Monte Carlo Tree Search, unterstützt durch das neuronale Netz, getroffen. In Kombination mit vielen anderen Verbesserungsmaßnahmen und einer deutlich komplexeren Netzstruktur als in dieser Arbeit möglich, war diese Zugauswahl ein Mittel zum Erfolg des AlphaGo Zero Projektes[37]. Eine Umsetzung in reduzierter Form könnte jedoch auch in kleinerem Rahmen bereits Verbesserungen zeigen.

9.2 Weitere Verbesserungsmöglichkeiten selbst trainierender Netze

Zur Verbesserung des Trainings der selbst trainierenden Netze stehen weitere, potenziell recht greifbare Verbesserungsmaßnahmen zur Verfügung, die in dieser Arbeit jedoch aus Zeit- und Ressourcengründen nicht umgesetzt wurden.

Die erste potenzielle Verbesserung ist das parallele Trainieren mehrerer Netze, potenziell sogar auf mehrere Maschinen verteilt, um das in Abschnitt 7.4 beschriebene Verfahren zu optimieren. Je mehr parallele Erkundung erfolgt, d. h. je mehr zufällige Züge ausgewählt werden, desto leichter sollte sich ein neuer Satz Gewichtungen finden lassen, der das bisher beste Netz schlägt.

Die zweite potenzielle Verbesserung ist der Einsatz von Population Based Training[40], bei dem parallel Netze mit variierten Hyperparametern wie beispielsweise der Lernrate, der Anzahl der versteckten Schichten und der Art der Aktivierungsfunktionen betrieben werden. Mit der Zeit entwickeln sich Hyperparameter, die eine gesteigerte Netzleistung erbringen.

Die dritte potenzielle Verbesserung ist der Einsatz eines genetischen Algorithmus zur Bestimmung optimaler Hyperparameter. Genetische Algorithmen wurden bereits erfolgreich auf einem mit Deep Q Networks konkurrierenden Niveau eingesetzt[41]. Vor allem die Kombination eines genetischen Algorithmus mit Q-Learning birgt möglicherweise hohes Potenzial.

Die vierte potenzielle Verbesserung ist neben der bereits durchgeführten Iteration verschiedener Netzarchitekturen weitere Arten der Dateneingabe und -verarbeitung zu testen. Dazu gehört unter anderem die Eingabe und Verarbeitung von Daten über Convolutional Layer[24], die vor allem bei der Arbeit mit Bildmaterial und Videospielen erfolgreiche Verwendung zeigen konnten[42].

[24] dt.: faltende Schichten

9.3 Themenvorschlag für eine aufbauende Studienarbeit

In dieser Studienarbeit wurden hauptsächlich Grundlagen für eine Arbeit mit durch Self Play trainierenden Netzen geschaffen. Es zeigt sich, dass ohne äußere Steuerung bereits bei augenscheinlich geringer Steigerung der Komplexität, wie beim Schritt von Tic-Tac-Toe zu Vier gewinnt, ein mitunter stark gesteigerter Bedarf an Trainingszeit einhergeht.

Daher könnte sich eine Untersuchung der Entwicklung von selbst trainierenden Netzen nach einer anfänglichen geführten Trainingsphase, in welcher zumindest Teile der für einen Menschen offensichtlichen Spielregeln antrainiert wurden, für eine weiterführende Arbeit eignen. Dazu gehören beispielsweise, am Beispiel von Vier gewinnt, das Ergreifen von direkten Gewinnmöglichkeiten, das verhindern von Gewinnen des Gegners in eindeutigen Situationen und das optimale Starten eines Spiels in der mittleren Spalte.

Literatur

[1] Warren S. McCulloch und Walter Pitts. „A logical calculus of the ideas immanent in nervous activity“. In: *The bulletin of mathematical biophysics* 5.4 (Dez. 1943), S. 115–133. DOI: 10.1007/BF02478259.

[2] F. Rosenblatt. „The Perceptron: A Probabilistic Model for Information Storage and Organization in The Brain“. In: *Psychological Review* 65 (Dez. 1958), S. 386–408. DOI: 10.1037/H0042519.

[3] Marvin Minsky und Seymour Papert. *Perceptrons. An Introduction to Computational Geometry*. MIT Press, Jan. 1969. ISBN: 9780262130431.

[4] David E. Rumelhart, Geoffrey E. Hinton und Ronald J. Williams. „Learning representations by back-propagating errors“. In: *Nature* 323 (Okt. 1986), S. 533–536. DOI: 10.1038/323533A0.

[5] Terrence J. Sejnowski und Charles R. Rosenberg. „Parallel networks that learn to pronounce English text“. In: *Complex systems* 1.1 (Feb. 1987), S. 145–168.

[6] Jon Russel. *Google's AlphaGo AI wins three-match series against the world's best Go player*. Mai 2017. URL: http://tcrn.ch/2rXmQup (besucht am 10. 07. 2018).

[7] Bela Bollobas. *Contemporary Combinatorics*. Bolyai Society Mathematical Studies. Springer Berlin, 2002. ISBN: 978-3-642-07660-2.

[8] Steve Schaefer. *MathRec Solutions(Tic Tac Toe)*. Jan. 2002. URL: http://www.mathrec.org/old/2002jan/solutions.html (besucht am 10. 07. 2018).

[9] Eric W. Weisstein. *Tic Tac Toe*. URL: http://mathworld.wolfram.com/Tic-Tac-Toe.html (besucht am 10. 07. 2018).

[10] Kevin Crowley und Robert S. Siegler. „Flexible Strategy Use in Young Children's Tic-Tac-Toe". In: *Cognitive Science* 17.4 (Okt. 1993), S. 531–561. DOI: 10.1207/S15516709cog1704_3.

[11] Stefan Edelkamp und Peter Kissmann. „Symbolic classification of general two-player games". In: *KI 2008: Advances in Artificial Intelligence*. Springer Berlin, 2008, S. 185–192. DOI: 10.1007/978-3-540-85845-4_23.

[12] Brady Haran. *Connect Four - Numberphile*. youtube. Dez. 2013. URL: https://www.youtube.com/watch?v=yDWPi1pZ0Po (besucht am 10. 07. 2018).

[13] Louis Victor Allis. „A knowledge-based approach of connect-four. The Game is Solved: White Wins". Masterarbeit. Amsterdam, The Netherlands: Vrije Universiteit, Okt. 1988.

[14] James D Allen. *Expert Play in Connect-Four*. 1990. URL: http://www.pomakis.com/c4/expert_play.html (besucht am 11. 07. 2018).

[15] John Tromp. *John's connect four playground*. URL: https://tromp.github.io/c4/c4.html (besucht am 11. 07. 2018).

[16] Robin Hunicke und Vernell Chapman. „AI for Dynamic Difficulty Adjustment in Games". In: *Challenges in Game Artificial Intelligence* (2005), S. 91–96.

[17] Marcel van Gerven und Sander Bohte. *Artificial Neural Networks as Models of Neural Information Processing*. Frontiers Research Topics. Frontiers Media, Feb. 2018. ISBN: 978-2-88945-401-3.

[18] Radek Mackowiak. *KNN: Natur als Vorbild – Biologische Neuronen*. 2015. URL: https://data-science-blog.com/blog/2015/09/29/knn-natur-als-vorbild-biologische-neuronen/ (besucht am 11. 07. 2018).

[19] Frederico A.C. Azevedo u. a. „Equal numbers of neuronal and nonneuronal cells make the human brain an isometrically scaled-up primate brain“. In: *Journal of Comparative Neurology* 513.5 (Apr. 2009), S. 532–541. DOI: 10.1002/cne.21974.

[20] *Schema eines Neurons*. Dez. 2006. URL: https://de.wiktionary.org/wiki/Datei:Neuron_(deutsch)-1.svg (besucht am 11. 07. 2018).

[21] Chrislb. *Darstellung eines künstlichen Neurons mit seinen Elementen*. Juli 2005. URL: https://de.wikipedia.org/wiki/Datei:ArtificialNeuronModel_deutsch.png (besucht am 11. 07. 2018).

[22] A. K. Jain, Jianchang Mao und K. M. Mohiuddin. „Artificial neural networks: a tutorial“. In: *Computer* 29.3 (März 1996), S. 31–44. DOI: 10.1109/2.485891.

[23] Leslie Pack Kaelbing, Michael L. Littman und Andrew W. Moore. „Reinforcement Learning: A Survey“. In: *Journal of Artificial Intelligence Research* 4 (1996), S. 237–285. DOI: 10.1613/jair.301.

[24] Tambet Matiisen. *Demystifying Deep Reinforcement Learning*. Dez. 2015. URL: http://neuro.cs.ut.ee/demystifying-deep-reinforcement-learning/ (besucht am 11. 07. 2018).

[25] Leonardo Araujo dos Santos. *Artificial Inteligence*. 2018. URL: https://legacy.gitbook.com/book/leonardoaraujosantos/artificial-inteligence (besucht am 10. 07. 2018).

[26] Christopher John Cornish Hellaby Watkins. „Learning from Delayed Rewards“. Diss. Cambridge, UK: King's College, Mai 1989.

[27] Francisco S Melo. *Convergence of Q-learning: A simple proof*. Tech. rep. Institute Of Systems und Robotics, 2001.

[28] John McCullock. *A painless q-learning tutorial*. URL: http://mnemstudio.org/path-finding-q-learning-tutorial.htm (besucht am 12. 07. 2018).

[29] Peter Kissmann und Stefan Edelkamp. „Symbolic Classification of General Multi-Player Games“. In: *European Conference on Artificial Intelligence (ECAI)*. 2008, S. 905–906. DOI: 10.3233/978-1-58603-891-5-905.

[30] Volodymyr Mnih u. a. „Human-level control through deep reinforcement learning“. In: *Nature* 518.7540 (Feb. 2015), S. 529–533. DOI: 10.1038/nature14236.

[31] Gerald Tesauro. „Temporal Difference Learning and TD-Gammon“. In: *Communications of the ACM* 38.3 (März 1995), S. 58–68. DOI: 10.1145/203330.203343.

[32] Nitish Srivastava u. a. „Dropout: A Simple Way to Prevent Neural Networks from Overfitting“. In: *Journal of Machine Learning Research* 15 (Jan. 2014), S. 1929–1958. ISSN: 1532-4435.

[33] Long-Ji Lin. „Self-improving reactive agents based on reinforcement learning, planning and teaching“. In: *Machine Learning* 8.3 (Mai 1992), S. 293–321. DOI: 10.1007/BF00992699.

[34] Tom Schaul u. a. „Prioritized Experience Replay“. In: *CoRR* abs/1511.05952 (2015). URL: https://arxiv.org/abs/1511.05952.

[35] Hado van Hasselt, Arthur Guez und David Silver. „Deep Reinforcement Learning with Double Q-learning“. In: *CoRR* abs/1509.06461 (2015). URL: https://arxiv.org/abs/1509.06461.

[36] Ziyu Wang, Nando de Freitas und Marc Lanctot. „Dueling Network Architectures for Deep Reinforcement Learning“. In: *CoRR* abs/1511.06581 (2015). URL: https://arxiv.org/abs/1511.06581.

[37] David Silver u. a. „Mastering the game of Go without human knowledge“. In: *Nature* 550.7676 (Okt. 2017), S. 354–359. DOI: 10.1038/nature24270.

[38] C. B. Browne u. a. „A Survey of Monte Carlo Tree Search Methods“. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4.1 (März 2012), S. 1–43. DOI: 10.1109/TCIAIG.2012.2186810.

[39] Hui Wang, Michael Emmerich und Aske Plaat. „Monte Carlo Q-learning for General Game Playing“. In: *CoRR* abs/1802.05944 (2018). URL: https://arxiv.org/abs/1802.05944.

[40] Max Jaderberg u. a. „Population Based Training of Neural Networks“. In: *CoRR* abs/1711.09846 (2017). URL: https://arxiv.org/abs/1711.09846.

[41] Felipe Petroski Such u. a. „Deep Neuroevolution: Genetic Algorithms Are a Competitive Alternative for Training Deep Neural Networks for Reinforcement Learning“. In: *CoRR* abs/1712.06567 (2017). URL: https://arxiv.org/abs/1712.06567.

[42] Volodymyr Mnih u. a. „Playing Atari with Deep Reinforcement Learning“. In: *CoRR* abs/1312.5602 (2013). URL: https://arxiv.org/abs/1312.5602.