FH Westküste
Wirtschaft und Technik

HAW
HAMBURG

Masterarbeit

Finn Lanz

Deep Reinforcement Learning zum maschinellen Erlernen von Strategien zur Lösung von Zauberwürfeln

FACHHOCHSCHULE WESTKÜSTE
Fachbereich Technik

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Fakultät Technik und Informatik
Department Informations- und Elektrotechnik

Finn Lanz

Deep Reinforcement Learning zum maschinellen Erlernen von Strategien zur Lösung von Zauberwürfeln

Masterarbeit eingereicht im Rahmen der Masterprüfung
im gemeinsamen Masterstudiengang Mikroelektronische Systeme
am Fachbereich Technik
der Fachhochschule Westküste
und
am Department Informations- und Elektrotechnik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Marc Hensel
Zweitgutachter: Prof. Dr. Sönke Appel

Eingereicht am: 11. Juli 2023

Finn Lanz

Thema der Arbeit

Deep Reinforcement Learning zum maschinellen Erlernen von Strategien zur Lösung von Zauberwürfeln

Stichworte

Maschinelles Lernen, Neuronale Netze, Deep Learning, Reinforcement Learning, Zauberwürfel, Rubik's Cube, Monte-Carlo-Baumsuche, A*-Algorithmus

Kurzzusammenfassung

In dieser Arbeit wird untersucht, wie neuronale Netzwerke trainiert werden können, damit diese Strategien zur Lösung von Zauberwürfeln entwickeln. Anschließend werden diese Netzwerke in geeignete Suchalgorithmen integriert, um die Effektivität und die Laufzeit bei der Lösungsfindung zu steigern.

Finn Lanz

Title of Thesis

Deep reinforcement learning to develop strategies for solving magic cubes with machine learning

Keywords

Machine learning, neural networks, deep learning, reinforcement learning, magic cube, Rubik's Cube, Monte Carlo tree search, A* search algorithm

Abstract

This thesis investigates how neural networks can be trained to develop strategies for solving magic cubes. Subsequently, these networks are integrated into suitable search algorithms to enhance the effectiveness and efficiency of the solution finding process.

Danksagung

Bevor ich die Leserin, den Leser auf meine Arbeit loslasse, will ich diese Gelegenheit nutzen, mich bei all denjenigen zu bedanken, die mich während des gesamten Erstellungszeitraums so wunderbar unterstützt haben. Seit euch sicher, dass ich es ohne euch nicht halb so gut durchgestanden hätte.

Zuallererst möchte ich meinem Betreuer und Erstprüfer Prof. Dr. Marc Hensel danken. Danke dafür, dass Sie mich von Anfang an ermutigt haben eine Thematik auszuwählen, die meine Neugierde weckt und mich so durchgehend motiviert gehalten hat. Ihre Ratschläge und vor allem Ihr Beistand waren zu jedem Zeitpunkt Gold wert. Danke auch an Prof. Dr. Sönke Appel für die Übernahme der Position des Zweitprüfers, welche in Anbetracht der Seitenzahlen dieser Thesis sicherlich auch einiges an Arbeit bedeutet.

Ohne Malte Müller und Carsten Klein hätte ich diese Masterarbeit wohl nicht geschrieben. Sie motivierten mich nach Abschluss meines Bachelors das Masterstudium gemeinsam mit ihnen fortzusetzen. Danke, dass wir das ganze Studium über eine Lerngruppe waren.

Und zum Schluss möchte ich meinen alltäglichen Unterstützern danken. Meine Familie unterstützt mich seit meinem Studienbeginn mental, finanziell und emotional. Danke dafür, dass ihr mir immer unter die Arme gegriffen habt, wenn ich alleine nicht weitergekommen bin. Danke für unzähliges Korrekturlesen, Zuhören und etliche 'Du schaffst das schon'. Und danke an meine Freundin, die sich dem angeschlossen und sich genauso liebevoll um mich gekümmert hat (Ich bin sicherlich nicht immer einfach). Danke an meinen Vater, der mir damals versicherte, ich würde die naturwissenschaftlichen Fächer schon noch irgendwann verstehen. Es würde irgendwann 'Klick' machen. Ich wünschte du könntest hier sein und sehen wie es 'Klick' gemacht hat.

Ich bin euch allen wirklich von Herzem dankbar. Deswegen noch einmal: **Danke**, **Danke**, **Danke**!

Inhaltsverzeichnis

Abbildungsverzeichnis **ix**

Tabellenverzeichnis **xiii**

1 Einleitung **1**

1.1 Stand der Technik . . . 2

1.2 Ziel dieser Arbeit . . . 2

2 Grundlagen: Künstliche Intelligenz **4**

2.1 Reinforcement Learning (RL) . . . 4

2.1.1 Reinforcement Learning vs. Supervised Learning . . . 5

2.1.2 Markov Decision Process (MDP) . . . 6

2.1.3 Policy- und Value-Funktionen . . . 9

2.1.4 Optimale Policy . . . 10

2.1.5 Q-Learning . . . 11

2.1.6 Epsilon-Greedy-Strategie . . . 14

2.2 Deep Learning . . . 16

2.2.1 Neuronales Netzwerk . . . 16

2.2.2 Gradientenabstiegsverfahren . . . 23

2.2.3 Backpropagation . . . 27

2.2.4 Gewichtsinitialisierung . . . 33

2.2.5 Hyperparameter . . . 36

2.2.6 Probleme beim Trainieren eines neuronalen Netzes . . . 49

2.2.7 Arten von neuronalen Netzen . . . 52

2.3 Deep Reinforcement Learning . . . 58

3 Grundlagen: Mathematik **60**

3.1 Zauberwüfel . . . 60

3.1.1 Terminologie und Notation . . . 61

3.1.2 Stand der Technik 67
3.2 Suchalgorithmen 70
3.2.1 Darstellung 71
3.2.2 Breitensuche 73
3.2.3 Dijkstra Algorithmus 77
3.2.4 Heuristische Suche 78
3.2.5 A*-Algorithmus 80
3.2.6 Bestensuche 81
3.2.7 Tiefensuchen 82
3.2.8 Monte-Carlo-Baumsuche 82

4 Anforderungsanalyse 90
4.1 Beschreibung 90
4.1.1 Software 90
4.1.2 Hardware 91
4.2 Stakeholder 91
4.3 Anwendungsfälle 92
4.4 Anforderungen 95
4.4.1 Software 96
4.4.2 Hardware 98

5 Konzept 100
5.1 Software 101
5.1.1 Entwicklungsumgebung 101
5.1.2 Trainingsumgebung (Simulation) 102
5.1.3 Zustandsrepräsentation 102
5.1.4 Neuronales Netz 108
5.1.5 Training des neuronalen Netzwerkes 109
5.1.6 Anwendung des neuronalen Netzes 110
5.1.7 Bedienoberfläche 113
5.2 Hardware 113

6 Softwareentwicklung: Custom Environment für OpenAI Gym 116
6.1 Bibliotheken 116
6.1.1 OpenAI Gym 117
6.1.2 Pygame 117

6.2 Würfelumgebung . . . 118
6.2.1 Zustände . . . 118
6.2.2 Aktionen . . . 119
6.2.3 OpenAI Gym Funktionen . . . 120
6.2.4 Weitere Funktionen . . . 125

7 Softwareentwicklung: Bedienoberfläche 127
7.1 Bibliothek . . . 127
7.2 Aufbau . . . 128
7.2.1 Bereich: Darstellungen . . . 128
7.2.2 Bereich: Einstellungen . . . 130
7.2.3 Kamera . . . 130

8 Algorithmusentwicklung: Monte-Carlo-Baumsuche 133
8.1 Datenrepräsentation . . . 133
8.2 Neuronales Netz . . . 134
8.2.1 Architektur . . . 134
8.2.2 Hyperparameterauswahl . . . 135
8.3 Training des neuronalen Netzwerkes . . . 140
8.4 Anwendung des trainierten Netzwerkes . . . 143
8.4.1 Breitensuche . . . 149
8.4.2 Stapelverarbeitung . . . 151

9 Algorithmusentwicklung: A*-Suchalgorithmus 152
9.1 Datenrepräsentation . . . 152
9.2 Neuronales Netz . . . 152
9.2.1 Architektur . . . 152
9.2.2 Hyperparameterauswahl . . . 154
9.3 Training des neuronalen Netzwerkes . . . 158
9.4 Anwendung des trainierten Netzwerkes . . . 159

10 Evaluation 163
10.1 Evaluierung der Software: Suchalgorithmen . . . 163
10.1.1 MCTS . . . 164
10.1.2 Batched MCTS . . . 170
10.1.3 A*-Algorithmus . . . 173
10.2 Evaluierung der Software: Bedienoberfläche . . . 179

10.3 Evaluierung der Hardware 179
10.4 Auswertung der Ergebnisse 179
10.4.1 Software 179
10.4.2 Hardware 182

11 Schlussfolgerungen 183
11.1 Zusammenfassung 183
11.2 Offene Punkte 185
11.3 Ausblick 185

Literaturverzeichnis 189

A Anhang 193
Selbstständigkeitserklärung 194

Abbildungsverzeichnis

2.1 Interaktion zwischen dem Agent und seiner Umgebung in einem MDP . . 7
2.2 Spielfeld mit den zugehörigen Rewards und Punishments 12
2.3 Beispiel für ein Exploitation-Exploration-Ungleichgewicht 13
2.4 Beispiel für Bestrafung 15
2.5 Einfaches Perzeptron 17
2.6 Einfaches Perzeptron-Netzwerk 18
2.7 Perzeptron als NAND-Gatter 19
2.8 Vergleich zwischen einem Perzeptron-Netzwerk und Nand-Gattern 20
2.9 Zwei Funktionen zur Aktivierung eines künstlichen Neurons 21
2.10 Neuronales Netzwerk mit den zugehörigen Schichtnamen 22
2.11 Darstellung einer vereinfachten Kostenfunktion im zweidimensionalen Raum 24
2.12 Darstellung einer vereinfachten Kostenfunktion im zweidimensionalen Raum am Beispiel eines Balles 25
2.13 Beispiel für die Notationsweise von Bias, Gewicht und Aktivierung 28
2.14 Zusammenhang zwischen einem Neuron und den Neuronen der vorherigen Schicht 28
2.15 Verdeutlichung der Gleichung für die Änderungsrate der Kostenfunktion im Bezug zu den Gewichten in einem Netzwerk 32
2.16 Vergleich zwischen Gleichverteilungs- und Xavier-Initialisierung 34
2.17 Vergleich zwischen der Gewichtsinitialisierung nach Glorot und Bengio und der nach Xavier 35
2.18 Einfluss der Lernrate im Gradientenabstieg 36
2.19 Beispiel für die Lernratenfindungsmethode 37
2.20 Zyklische Lenrate 38
2.21 Kurvenverlauf der Sigmoid Aktivierungsfunktion 41
2.22 Kurvenverlauf der ReLU Aktivierungsfunktion 42
2.23 Kurvenverlauf der Leaky ReLU Aktivierungsfunktion 43
2.24 Kurvenverlauf der eLU Aktivierungsfunktion 43

2.25 Optimierer im Vergleich 47
2.26 Convolutional Neural Network: Verbinden eines Feldes des Bildes mit dem ersten Eingangsneuron 53
2.27 Convolutional Neural Network: Verbinden eines Feldes des Bildes mit dem zweiten Eingangsneuron 53
2.28 Feature Maps in einem Convolutional Neural Network 55
2.29 Max-Pooling in einem Convolutional Neural Network 55
2.30 Beispielhaftes Convolutional Neural Network 56
2.31 Beispielhaftes Recurrent Neural Network 57
2.32 Ein Residual Block mit zugehöriger Skip-Connection 58
2.33 Reinforcement Learning in Kombination mit einem neuronalen Netzwerk . 59

3.1 Darstellung des Rubik's Cube im gelösten Zustand 60
3.2 Darstellung der Würfeltransformationen nach David Singmasters Notation 61
3.3 Orientierungunsmöglichkeiten eines Ecksteins 62
3.4 Orientierungunsmöglichkeiten eines Kantensteins 63
3.5 Notation der Würfelseiten 63
3.6 Reihenfolge bei der Beschreibung eines Würfelsteins anhand des Beispiels des URF-Cubies 64
3.7 Beispielkarte mit eingezeichneter Graphendarstellung 71
3.8 Verschiedene Positionierungsmöglichkeiten für Knotenpunkte in einem Graphen 72
3.9 Expandierung der Ringgrenze beim Breitensuchverfahren 73
3.10 Expandierung der Ringgrenze bei dem Breitensuchverfahren im Detail . . 74
3.11 Pfadfindung mittels Breitensuche und angepasstem Python-Code 75
3.12 Bewegungskosten: Vergleich zwischen Schritten und Distanz 76
3.13 Ausbreitung der Ringgrenze bei dem Verfahren nach Dijkstra im Vergleich zum Breitensuchverfahren 78
3.14 Vergleich zwischen Dijkstra und Greedy Best-First Search 79
3.15 Vergleich zwischen Dijkstra und Greedy Best-First Search in einer komplexeren Umgebung 80
3.16 Vergleich zwischen Dijkstra, Greedy Best-First Search und A* 81
3.17 Ablauf der Tiefensuche 82
3.18 Darstellung eines MDPs als ExpectiMax-Baum 83
3.19 Ablauf einer Sequenz innerhalb einer Monte-Carlo-Baumsuche 85
3.20 Selection: Auswahl eines Knotenpunktes 86

3.21 Expansion: Erweiterung eines Knotenpunktes 87
3.22 Algorithmus für die Auswahl und Erweiterung von Knotenpunkten 88
3.23 Simulation: Simulieren des Pfades des neuen Knotenpunktes bis zu einem Endzustand 88
3.24 Backpropagation: Aktualisieren der Statistiken aller angetroffenen Knotenpunkte 89

4.1 Anwendungsfalldiagramm des Zauberwürfellösers 93
4.2 Anteil der gelösten 2x2x2-Würfel in der Ausarbeitung von Max Lapan . . 99

5.1 Rubik's Cube in berechenbarer Form 106
5.2 Darstellung des Zauberwürfels in 4D 107
5.3 Ablaufdiagramm der Bedienoberfläche 114
5.4 3D-gedruckter 2x2x2-Zauberwürfellöser 115

6.1 2D-Darstellung des aktuellen Zustands in der Pocketcube-Umgebung . . . 124
6.2 3D-Darstellung des Pocketcubes 125

7.1 Aufbau der grafischen Benutzeroberfläche 129
7.2 Hilfefenster der Anwendung 132

8.1 Minimal angepasste Architektur des von McAleer et al. verwendeten neuronalen Netzwerkes (2018) 134
8.2 Lenratenfindung für die MCTS-Methode 137
8.3 Gesamtverlust beim Training mit dem Pocketcube mit einer Batchsize von 10.000 Würfelzuständen und einer Distanzgewichtung 138
8.4 Gesamtverlust beim Training mit dem Pocketcube mit einer Batchsize von 10.000 Würfelzuständen ohne Distanzgewichtung 139
8.5 Visualisierung der Trainingsdatengenerierung mittels ADI 142
8.6 Erweiterung eines Blattknotens um seine Kindknoten 148
8.7 Vereinfachtes Beispiel der MCTS-Methode 150

9.1 Architektur des von McAleer et al. verwendeten neuronalen Netzwerkes (2019) 153
9.2 Grober Aufbau der Architektur bei Verwendung eines zusätzlichen Zielnetzwerkes 154
9.3 Lernratenfindung für die Methode mit A*-Algorithmus 156

9.4 Lernkurvenverlauf bei dem Training eines neuronalen Netzwerkes unter Verwendung eines Error Thresholdes 158

10.1 Durchschnittliche Laufzeiten des MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen 165
10.2 Erfolgsquote des MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen . 166
10.3 Anzahl der benötigten Schritte pro Suche mittels MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen 167
10.4 Lösungsweglängen des MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen 168
10.5 Vergleich der erforderlichen Zeit pro Suchdurchlauf zwischen dem MCTS-Verfahren mit und ohne anschließender Breitensuche 169
10.6 Vergleich der Lösungsweglängen zwischen dem MCTS-Verfahren mit und ohne anschließender Breitensuche 170
10.7 Vergleich zwischen den von Lapan implementierten und den hier entworfenen darauf aufbauenden Methoden mit und ohne Batchverarbeitung . . 171
10.8 MCTS-Verfahren mit einer Batchverarbeitung der Größe $b = 1.000$ im Detail 172
10.9 Benötigte Laufzeit des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$ 174
10.10 Erfolgsquote des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$ 175
10.11 Anzahl der benötigten Iterationen des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$ 176
10.12 Lösungsweglänge des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$ 177
10.13 Ergebnisse der Gridsearch für die Findung der optimalen Parameterkonfiguration 178

11.1 Darstellung der Funktionsweise des von Kyo Takano entwickelten Verfahrens 187

Tabellenverzeichnis

2.1 Initiale Q-Values des Beispielspiels in Tabellenform 12
2.2 Aktualisierte Q-Tabelle des Beispiels nach Zeitschritt $t = 1$ der ersten Episode. 16

4.1 Anwendungsfall: Würfelzustand erfassen 93
4.2 Anwendungsfall: Würfelzustand lösen 94
4.3 Anwendungsfall: Würfelzustand anzeigen 94
4.4 Anwendungsfall: Würfelzustand eingeben 95
4.5 Anwendungsfall: Würfelseiten verdrehen 95
4.6 Funktionale Anforderungen an die Software. 96
4.7 Nicht funktionale Anforderungen an die Software. 97
4.8 Funktionale Anforderungen an die Hardware. 99

5.1 Größenvergleich der Zustandsräume des Zauberwürfels nach Darstellungsart 107
5.2 Übersicht über die neuronalen Netzwerkarten aus Abschnitt 2.2.1. 108
5.3 Angepasste funktionale Anforderung an die Software. 113

10.1 Auswertung der funktionalen Anforderungen, welche an die Software gestellt wurden. 180
10.2 Auswertung der nicht funktionalen Anforderungen, welche an die Software gestellt wurden. 181
10.3 Auswertung der funktionalen Anforderungen, welche an die Hardware gestellt wurden. 182

1 Einleitung

Bereits seit ein paar Jahrzehnten begeistert der Zauberwürfel, auch Rubik's Cube (3x3x3) genannt, die Menschen und es gibt kaum jemanden, der noch nicht von ihm gehört hat. Er ist eines der meistverkauftesten Puzzlespiele der Welt und fasziniert auf Grund seines einfachen Aufbaus. Dennoch gehört er mit zu den schwersten kombinatorischen Puzzlespielen, obwohl es sogar Kindern mittlerweile möglich ist, ihn zu lösen. Inzwischen sind etliche Algorithmen bekannt, mit denen ein vollständig verdrehter Zauberwürfel wieder in den Ursprungszustand zurückversetzt werden kann. Ohne diese sähe die Situation natürlich ganz anders aus. Denn der Cube besitzt um einiges mehr an unterschiedlichen Zuständen als es Menschen auf dieser Erde gibt. Somit ist es ohne Anleitung gar nicht so einfach, bis völlig unmöglich, den alleinigen Lösungszustand durch planloses Verdrehen der sechs Seiten zu erreichen. Es könnte sehr viel Zeit in Anspruch nehmen, bis ein Mensch alleine zuverlässige Algortihmen entwickelt, um den Rubik's Cube aus jedem beliebigen Zustand zur gewünschten Lösung zu bringen.

Wie sieht das Ganze jedoch aus, wenn eine künstliche Intelligenz versucht sich das kombinatorische Puzzlespiel selbst beizubringen? Mit dieser Frage und deren Lösung soll sich diese Thesis beschäftigen und wird im Laufe von dieser ausgearbeitet. Dabei wird der Fokus zunächst auf der kleineren Version des Zauberwürfels, dem Pocketcube (2x2x2), liegen. Dieser besitzt weitaus weniger mögliche Zustände und ermöglicht ein einfacheres Ausarbeiten von Lösungsansätzen als der Rubik's Cube. Es wird eine Form des maschinellen Lernes namens Deep Reinforcement Learning angewendet werden, welche eine Kombination zwischen Deep Learning und Reinforcement Learning darstellt. Hierbei wird ein neuronales Netz mit Hilfe von Belohnungen positiv in richtigen Verhaltensweisen bestärkt. In diesem Fall, wenn es den Lösungszustand des Zauberwürfels durch korrekte Auswahl der möglichen Aktionen erreicht.

1.1 Stand der Technik

Diese Arbeit stützt sich vor allem auf die wissenschaftlichen Artikel „Solving the Rubik's Cube with Approximate Policy Iteration“ [21] und „Solving the Rubik's cube with deep reinforcement learning and search“ [22], welche von einer Gruppe aus Wissenschaftlern von der University of California (UCI) ausgearbeitet wurden. Des Weiteren wurde der erste Artikel der Forschungsgruppe McAleer et al. im Buch von Max Lapan „Deep Reinforcement Learning Hands-On“ [18] näher untersucht und diente auch als Hilfe für diese Arbeit. Vor dem Beitrag der Gruppe der UCI gab es zwei andere große Ansätze um den Zauberwürfel zu lösen [18]:

- Gruppentheorie: Verringern des zu untersuchenden Zustandsraumes (*State Space*). (Beispiel: Kociemba Algorithmus[1])
- Brute force: Kombiniert mit manueller Heuristik um die Suche in eine vielversprechende Richtung zu lenken. (Beispiel: Korf's Algorithmus [17])

Die veröffentlichten Paper zeigen eine dritte Möglichkeit auf: Das Trainieren eines neuronalen Netzwerkes mit verdrehten Zauberwürfeln, um eine Verhaltensstrategie (*Policy*) und/oder Zustandsbewertung (*Value*) zu erhalten, welche die Richtung zum Lösungszustand weisen sollen [21, 22]. Dafür wird kein menschliches Vorwissen benötigt, nur eine simulierte Umgebung (*Environment*), mit welcher das Netz trainiert wird. Die anderen beiden Lösungen setzen eine Menge Wissen über die Mathematik hinter dem Würfel voraus, um diese in einen Code umzusetzen.

1.2 Ziel dieser Arbeit

Das Ziel dieser Arbeit ist es, die in den Artikeln [21, 22] beschriebenen (und von Max Lapan [18] umgesetzen) Verfahren zum Lösen des Zauberwürfels zu rekreieren und in einem eigenem Code umzusetzen. Dabei sollen die benötigten Grundlagen erarbeitet und eingängig erläutert werden, damit es späteren auf dieser Thesis aufbauenden Arbeiten erleichtert wird, einen Einstieg in das Thema *Deep Reinforcement Learning* zu finden. Außerdem wurde von Prof. Dr. Marc Hensel ein Demonstrator entworfen, welcher mit dem Pocketcube kompatibel ist. Dieser soll mit den entworfenen Lösungen verbunden werden

[1] http://kociemba.org/cube.htm - Zugriffsdatum: 07.07.23

und es ermöglichen einen von Hand verdrehten Würfel zu erkennen, die entsprechende Lösung zu finden sowie diese umzusetzen.

2 Grundlagen: Künstliche Intelligenz

In diesem Kapitel sollen die theoretischen Grundlagen vermittelt werden, die notwenig sind, um die erarbeiteten Lösungen und die Schritte, die nötig waren um dorthin zu gelangen, nachvollziehen zu können. Dabei soll grundlegend auf Reinforcement und Deep Learning eingegangen werden, und wie diese beiden Bereiche sinnvoll kombiniert werden können, damit ein neuronales Netz eine Tätigkeit, wie das Lösen des Zauberwürfels, erlernen kann.

2.1 Reinforcement Learning (RL)

Reinforcement Learning ist ein populärer Bereich des maschinellen Lernens. Es hat sich in den letzten Jahren drastisch weiterentwickelt, was vor allem mit seiner Kombination des Deep Learning zu Deep Reinforcement Learning zu tun hat [8]. Dabei wird ein sogenannter Agent (Entscheidungsträger beim maschinellen Lernen) über ein Belohnungs- -/Bestrafungssystem (*Rewards* bzw. *Punishments*) traininert, in seiner Umgebung die richtigen Aktionen zu wählen. Dieser versucht den möglichen Gewinn innerhalb dieses Systems durch wiederholt richtige Entscheidungen zu maximieren. Durch Auswerten der dabei gesammelten Erfahrungen ist der Agent im Stande, sich weiterzuentwickeln und künftig noch bessere Entscheidungen in seiner Umgebung auszuwählen. Es unterscheidet sich zu anderen maschinellen Lernarten darin, dass das neuronale Netzwerk hierbei eigenständig lernt und keine Überwachung (wie beim Supervised Learning) oder komplette Rechenmodelle des Environments notwendig sind.

Bei der Auseinandersetzung mit dieser Thematik werden folgende Fachbegriffe am häufigsten angetroffen [8]:

- Agent (*Agent*): Hauptentscheider über die zu treffenden Aktionen. Beispiel: Zauberwürfellöser.

- Aktion (*Action*): Handlung, die ein Agent in einer ihn einbindenden Umgebung trifft. Beispiel: Verdrehen der Zauberwürfelseiten.
- Umgebung (*Environment*): Problemkontext, den der Agent zu lösen versucht. Beispiel: Verdrehten Zauberwürfel zurück zum Urzustand bringen.
- Zustand (*State*): Zustand der Umgebung. Beispiel: Anordnung der farbigen Sticker auf den verdrehten Zauberwürfelseiten.
- Belohnung (*Reward*): Numerischer Wert, den der Agent als Reaktion auf eine von ihm getroffene Aktion erhält. Beispiel: Bei Erreichen des Zielzustandes +1 Punkt.

Zusammengefasst handelt es sich beim Reinforcement Learning um das Maximieren des möglichen Rewards in einer („Problem-")Umgebung. Dabei sollte auch darauf geachtet werden, dass der Agent lernt, langfristig zu „denken", da eine Entscheidung, die eine sofortige Belohnung einbringt, möglicherweise negativen Einfluss auf den späteren Verlauf hat und nicht zum maximal möglichen Gewinn führt.

2.1.1 Reinforcement Learning vs. Supervised Learning

Reinforcement Learning (RL) unterscheidet sich vom Supervised Learning (SL) vor allem darin, wie Entscheidungen getroffen werden. Dabei verhält sich RL dynamisch, das bedeutet, dass der dort trainierte Agent abhängig von den bereits getroffenen Aktionen und der ihn einschließenden Umgebung seinen nächsten Schritt auswählt. Supervised Learning verhält sich dagegen statisch und ähnelt mehr einer mathematischen Funktion, welche unabhängig von bereits getätigten Entscheidungen jede Dateneingabe X auf eine Ausgabe Y abbildet. Beim Reinforcement Learning werden die Trainingsdaten von der Umgebung (Environment) des Agenten und dessen Aktionen erzeugt. Dieser muss eigenständig Muster in den Daten erkennen und dadurch erlernen, welches Verhalten richtig und welches falsch ist. Im Gegensatz dazu wird beim Supervised Learning ein neuronales Netzwerk mit bereits vorhandenen und korrekt gekennzeichneten Probedaten (beispielsweise „richtig" oder „falsch") trainiert und es muss nur noch überprüft werden, ob das Netz neue Daten anschließend richtig zuordnen kann.

2.1.2 Markov Decision Process (MDP)

Das Kernkonzept des Reinforcement Learnings stellt der *Markov Decision Process* (*Markow-Entscheidungsproblem*) dar. Dieser beschreibt das Verfahren zur Findung einer optimalen Verhaltensstrategie eines Agenten für das Environment, in welches dieser platziert wurde. Er erhält dabei eine Repräsentation seines Umgebungszustandes und wählt die seiner Auffassung nach beste Handlung in diesem aus. Danach befindet sich der Agent in dem durch diese Aktion kreierten Folgezustand und erhält eine entsprechende Belohnung (*Reward*) oder Bestrafung (*Punishment*) als Konsequenz. Somit besteht der Markov Decision Process aus folgenden Komponenten:

- Agent
- Umgebung, in welcher sich der Agent befindet
- Alle möglichen (hier finiten) Zustände $\mathcal{S}$
- Alle möglichen (hier finiten) Aktionen $\mathcal{A}$
- Alle möglichen (hier finiten) Belohnungen $\mathcal{R}$

Das Ziel des Agenten ist es die Belohnungen, die er durch seine Handlungen erhält, zu maximieren. Dabei sind nicht nur die gemeint, die er unmittelbar nach einer Entscheidung erhält, sondern auch die, welche sich im Verlauf des Prozesses aufsummieren. Der aktuelle Zustand wird mit s_t und die zugehörige Aktion, welche in s_t getätigt wird, wird mit a_t gekennzeichnet, wobei t für den aktuellen Zeitschritt steht. Basierend auf dem Zustand $s_t \in \mathcal{S}$ in dem sich der Agent befindet, wählt er eine Aktion $a_t \in \mathcal{A}$. Zusammen ergeben sie das sogenannte Zustand-Aktions-Pärchen (*State-Action-Pair*) (s_t, a_t). Für jede Aktion, die der Agent tätigt, erhält dieser im nächsten Zeitschritt eine Belohnung $r_{t+1} \in \mathcal{R}$. Das Erhalten eines Rewards kann somit als eine Funktion verstanden werden, welche Zustand-Aktions-Pärchen auf Belohnungen abbildet [32]:

$$f(s_t, a_t) = r_{t+1} \tag{2.1}$$

Der sequentielle Verlauf lässt sich vereinfacht darstellen wie folgt:

$$s_0 \rightarrow a_0 \rightarrow r_1 \rightarrow s_1 \rightarrow a_1 \rightarrow r_2 \rightarrow s_2 \rightarrow a_3 \rightarrow r_4 ...$$

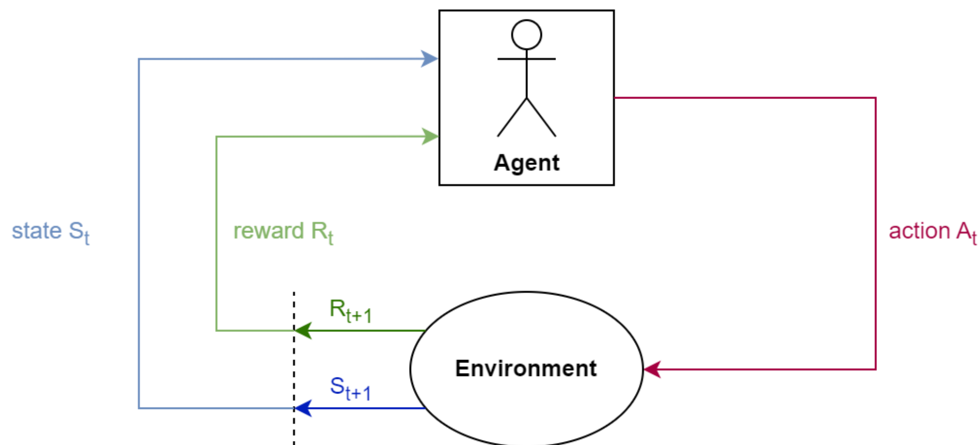


Abbildung 2.1: Interaktion zwischen dem Agent und seiner Umgebung in einem MDP [32, modifiziert].

Die Abfolge von mehreren Zuständen, beziehungsweise States, formen hierbei eine Kette, welche auch als *Markov Chain* bekannt ist. Die Zustandswechsel werden durch Übergangswahrscheinlichkeiten beschrieben, die nur abhängig sind von dem aktuellen Zustand des Environments. Wird jedem Übergangswechsel in einer Markov Chain ein numerischer Wert (dem Reward) zugewiesen, nennt man dies auch *Markov Reward Process*. [8] Abbildung 2.1 soll das Vorgehen noch einmal verdeutlichen.

Der MDP kann somit wie folgt zusammengefasst werden:

- Der Agent befindet sich in einem Zustand s_t seiner Umgebung.
- Er wählt die für ihn beste Aktion a_t im Zustand s_t.
- Durch die Aktion geht der Agent in einen neuen Zustand s_{t+1} der Umgebung über.
- Der Agent erhält einen entsprechenden Reward r_{t+1}.

Die Theorie des MDP soll also dabei helfen, die bestmögliche Aktion a in einem Zustand s zu treffen, indem die Übergangswahrscheinlichkeiten und die zu erhaltenen Rewards durch Transitionen berücksichtigt werden. Dadurch erlernt ein Agent ein auf seine Umgebung zugeschnittenes Regelwerk (*Policy*) mit Hilfe dessen er Entscheidungen trifft, welche seinen Gewinn maximieren.

Übergangswahrscheinlichkeiten

Da die Anzahl der Zustände $\mathcal{S}$ und die Belohnungen $\mathcal{R}$ endlich sind, besitzen die zugehörigen Zufallsvariablen r_t und s_t jeweils eine klar definierte Wahrscheinlichkeitsverteilung. Aufgrund dessen besitzen alle möglichen Werte von r_t und s_t eine teilweise verbundene Wahrscheinlichkeit, welche vom vorherigen Zustand und der vorherigen Aktion zur Zeit $t-1$ abhängt. Somit ist es möglich, dass $s' \in \mathcal{S} = s_t$ und $r \in \mathcal{R} = r_t$ ist. Die zugehörige Wahrscheinlichkeit wird durch die Werte des vorherigen Zustandes $s \in S$ und der Aktion $a \in \mathcal{A}(s)$ bestimmt. Wobei $A(s)$ alle Aktionen beinhaltet, die im Zustand s möglich waren. [32]

Die Wahrscheinlichkeit p, dass ein Übergang in Zustand s' mit Belohnung r durch Anwendung der Aktion a im aktuellen Zustand s stattfindet, lässt sich somit wie folgt definieren [32]:

$$p(s', r|s, a) = Pr\{s_t = s', r_t = r|s_{t-1} = s, a_{t-1} = a\} \tag{2.2}$$

Zu erwartender Gewinn

Der zu erwartende Gewinn (*Expected Return*) ist die Summe der im weiteren Verlauf zu erwartenden Belohnungen in einem Schritt t [32]:

$$g_t = r_{t+1} + r_{t+2} + r_{t+3} + ... + r_T \quad \text{mit T = Letzter Zeitschritt} \tag{2.3}$$

Das Konzept des zu erwartenden Gewinns g_t ist ein wichtiger Bestandteil des MDP, da es das Ziel des Agenten ist diesen zu maximieren und somit das ist, was diesen „antreibt“. Die Interaktionen mit seiner Umgebung lassen sich in Einzelsequenzen (sogenannten *Episoden*) aufteilen, wobei eine Sequenz beispielsweise eine Runde eines Spiels darstellt. Im Anschluss einer Sequenz wird der ursprüngliche Zustand s_{init} wiederhergestellt oder es wird durch Zufallsprinzip einer aus der Menge aller Zustände $\mathcal{S}$ ausgewählt. Jede Episode beginnt unabhängig von der vorherigen und deren Ausgang. Dabei unterscheidet man zwischen Aufgaben, welche innerhalb einer Episode erledigt werden (*Episodic Tasks*) und zwischen denen, die unaufhörlich fortlaufen (*Continuous Tasks*, beispielsweise Roboter an einem Fließband). Das Problem bei fortlaufenden Aufgaben ist, dass der zu erwartende Gewinn unendlich sein könnte, da auch die Zeit t unendlich fortläuft (vgl. Formel

2.3). Deshalb werden die zukünftigen Belohnungen r mit jedem Zeitschritt t weniger berücksichtigt, indem diese mit einem *Reduzierungsfaktor* γ (*Discount Factor*) multipliziert werden. Dieser befindet sich in einem Bereich zwischen 0 und 1 ($\gamma \in [0, 1]$). Somit wird Formel 2.3 entsprechend angepasst:

$$g_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + ... + \gamma^T \cdot r_T \tag{2.4}$$

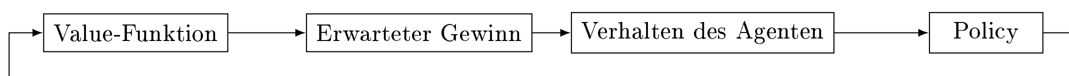
$$= \sum_{k=0}^{\infty} \gamma^k \cdot r_{t+k+1} \tag{2.5}$$

$$= r_{t+1} + \gamma g_{t+1} \tag{2.6}$$

Je nachdem wie der Faktor gewählt wird, beeinflussen unmittelbare Belohnungen das Verhalten des Agenten stärker als solche, die weiter entfernt sind. Durch γ lässt sich somit auch Einfluss darauf nehmen, wie langfristig der Agent „denkt". Denn je kleiner der Reduzierungsfaktor, desto weniger Gewicht haben die Belohnungen späterer Aktionen und desto mehr fokussiert sich der Agent auf unmittelbare Rewards.

2.1.3 Policy- und Value-Funktionen

Eine Policy(-Funktion) drückt aus, wie wahrscheinlich es ist, dass der Agent eine bestimmte Aktion a in einem Zustand s auswählt. Notiert wird diese mit $\pi(a|s)$, wobei π eine Wahrscheinlichkeitsverteilung für jeden Zustand $s \in \mathcal{S}$ über die Aktionen $a \in \mathcal{A}(s)$ ist. Sie bildet dafür die Zustände auf die zugehörigen Wahrscheinlichkeiten der dort möglichen Aktionen ab. Die Value-Funktion sagt hingegen aus, wie zielführend eine Aktion a oder ein Zustand s für den Agenten sind, und stellt ein Gütemaß für solche dar. Als Vergleichswert dient hier der zu erwartende Gewinn g_t. Da dieser durch das Verhalten des Agenten und dessen Handlungsentscheidungen beeinflusst wird, hat auch die Policy Einfluss auf die Rückgabe der Value-Funktion: [32]



Es kann zwischen zwei Value-Funktionen unterschieden werden, der *State-Value-Function* (*Zustandswertfunktion*) und der *Action-Value-Function* (*Aktionswertfunktion*) [32].

State-Value-Function: Notiert als v_π und dient als Gütemaß für die jeweiligen Zustände:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi \left[g_t \mid s_t = s \right] && (2.7) \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right] && (2.8) \end{aligned}$$

Mit $\mathbb{E}_\pi$ als Erwartungswert einer Zufallsvariable folgt der Agent Policy π zu jedem Zeitpunkt t.

Action-Value-Function: Notiert als q_π und sagt aus, wie wertvoll eine Aktion a in einem Zustand s ist:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi \left[g_t \mid s_t = s, a_t = a \right] && (2.9) \\ &= \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a \right] && (2.10) \end{aligned}$$

Die Action-Value-Funktion wird auch als *Q-Funktion* bezeichnet und besitzt als Rückgabewert den sogenannten Q-Value, wobei Q für Quality steht. Das *Q-Learning*, welches ein Kernkonzept des Reinforcement Learnings darstellt, basiert vor allem auf Formel 2.9.

2.1.4 Optimale Policy

Beim Reinforcement Learning ist das Ziel eine Policy zu finden, welche dem Agenten mehr Gewinn einbringt als alle anderen Policies. Dabei wird eine Policy π als besser oder gleich definiert, wenn der zu erwartende Gewinn für alle möglichen Zustände größer oder gleich einer anderen Policy π' ist [32]:

$$\pi \geq \pi' \quad \text{wenn} \quad v_\pi(s) \geq v_{\pi'}(s) \quad \forall s \in \mathcal{S} \tag{2.11}$$

Ist eine Policy besser oder mindestens genauso gut wie alle anderen, spricht man auch von der *optimalen Policy*. Die dazugehörigen Value-Funktionen (Formel 2.7 und 2.9) werden mit einem Stern gekennzeichnet und erbringen den größtmöglichen Gewinn für jeden möglichen Zustand beziehungsweise für jedes Zustand-Aktions-Pärchen [32]:

- State-Value-Funktion: $v_*(s) = \max_\pi v_\pi(s) \quad \forall s \in \mathcal{S}$
- Action-Value-Funktion: $q_*(s, a) = \max_\pi q_\pi(s, a) \quad \forall s \in \mathcal{S} \quad \text{und} \quad \forall a \in \mathcal{A}(s)$

Ein fundamentaler Bestandteil der Action-Value-Funktion q_* ist hierbei das Bellman Optimalitätsprinzip:

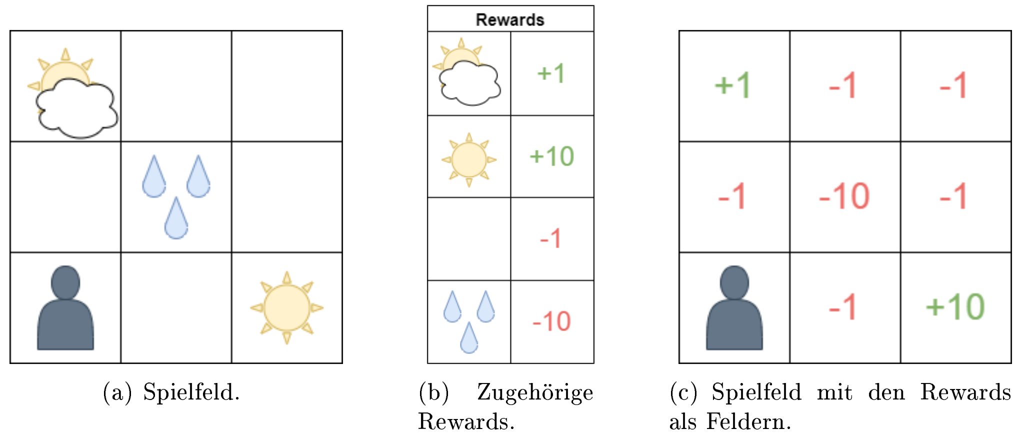
$$q_*(s, a) = \mathbb{E}[r_{t+1} + \gamma \max_{a'} q_*(s', a')] \tag{2.12}$$

Folgt man der Policy π_*, nachdem im Zustand s Aktion a gewählt wurde, ist der zu erwartende Gewinn die Summe aus der Belohnung r_{t+1} und der maximal erreichbaren reduzierten Belohung $\gamma \max_{a'} q_*(s', a')$ für jedes mögliche State-Action-Paar zum Zeitpunkt t. Da der Agent der optimalen Policy folgt, wird der Folgezustand s' die bestmögliche nächste Aktion a' zur Zeit $t + 1$ ermöglichen. Dies bedeutet, dass das Bellman Optimalitätsprinzip hilft die optimale Q-Funktion und somit die optimale Policy zu finden, indem durch diese für jeden Zustand s der maximale Q-Wert für diese gefunden werden kann. [32]

2.1.5 Q-Learning

Beim Q-Learning werden die Q-Werte für jedes State-Action-Paar iterativ mittels Bellmann Optimalitätsprinzip aktualisiert. Dies wird solange wiederholt, bis diese gegen ihre Optima q_* konvergieren (*Value Iteration*).

Anhand eines Gridworld-Beispiels lässt sich dieses Vorgehen verdeutlichen. In Abbildung 2.2 ist das Spielfeld dargestellt. Der Spieler befindet sich links unten auf dem Startfeld. Er kann sich innerhalb des 3x3-Spielfeldes nach oben, unten, links oder rechts frei bewegen. Betritt er das mittlere Feld, wird das Spiel terminiert und er wird bestraft (Punishment).



(a) Spielfeld.

(b) Zugehörige Rewards.

(c) Spielfeld mit den Rewards als Feldern.

Abbildung 2.2: Spielfeld mit den zugehörigen Rewards und Punishments.

Tabelle 2.1: Initiale Q-Values des Beispielspiels in Tabellenform

Aktionen Zustände	Hoch	Runter	Links	Rechts
Reward +1	0	0	0	0
Leer (oben mitte)	0	0	0	0
Leer (oben rechts)	0	0	0	0
Leer (mitte links)	0	0	0	0
End -10	0	0	0	0
Leer (mitte rechts)	0	0	0	0
Leer (unten links)	0	0	0	0
Leer (unten mitte)	0	0	0	0
End +10	0	0	0	0

Erreicht er die Sonne rechts unten, ist das Spiel ebenfalls zu Ende, jedoch wird er belohnt (Reward). Für das Betreten des Wolkenfeldes oben links erhält er nur einen Punkt als Belohnung. Auf allen anderen (leeren) Feldern wird ihm ein Punkt abgezogen und er somit bestraft. Zu Beginn seines Lernprozesses weiß der Agent nichts über seine Umgebung, weshalb die entsprechenden Q-Values, welche sich tabellarisch darstellen lassen, mit null initialisiert werden (vgl. Tabelle 2.1).

Die Werte in der Tabelle werden iterativ durch Erfahrungen aktualisiert, die der Agent während des Durchlaufens des Spiels in den jeweiligen Episoden sammelt. Im späteren Verlauf kann sich der Agent auf diese Erfahrungen bei neuen Entscheidungsfindungen berufen und wählt in seinem aktuellen Zustand die Aktion mit dem höchsten Q-Wert aus. Da die Werte zu Beginn allerdings mit null initialisiert sind, kann der Agent sich noch

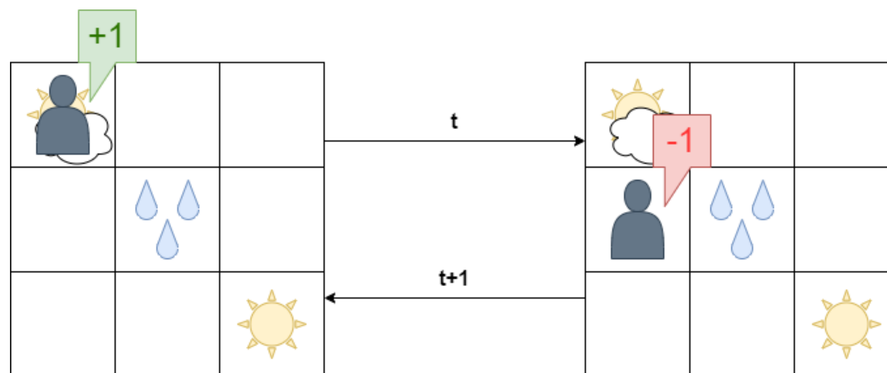


Abbildung 2.3: Mögliches Szenario, wenn Exploitation und Exploration nicht ausbalanciert sind. Der Agent hängt in einer Endlosschleife gleicher sich wiederholender Aktionen fest.

auf keine Erfahrungen stützen. Auch in den darauffolgenden Episoden ist es möglich, dass die Q-Tabelle noch nicht genügend richtungsweisend ist, um die passenden Entscheidungen zu treffen. Der Agent muss seine Umgebung zunächst ausreichend erkunden (*Exploration*), bevor er versuchen kann, die Belohnung pro Episode zu maximieren, indem er diese „ausbeutet" (*Exploitation*). Mit Erkundung ist hierbei gemeint, dass der Agent neue Bereiche seiner Umgebung erforscht und unsichere Aktionen austestet, um im späteren Verlauf bessere Entscheidungen treffen zu können. Dies beinhaltet das Durchführen von Aktionen, welche unter Umständen nicht unmittelbar Belohnungen einbringen, dafür allerdings Informationen über das Environment. Bei der Ausbeutung nutzt der Agent bereits gelerntes Wissen über seine Umgebung, um die besten bekannten Handlungen auszuwählen, die den Gewinn für seinen jetzigen Wissenstand maximieren. Es muss eine hinreichende Balance zwischen Exploration und Exploitation gefunden werden, um optimale Lösungen im Reinforcement Learning zu finden.

Versucht der Agent seine Umgebung nur „auszubeuten" und erreicht das Feld mit dem 'Reward +1' vor dem Feld mit der höheren 'Reward +10' könnte er dadurch beispielsweise in eine Endlosschleife sich wiederholender Aktionen geraten, bei welcher er das Feld mit dem 'Reward +1' immer wieder betritt, anschließend verlässt ('Punishment -1') und wieder betritt ('Reward +1') (vgl. Abbildung 2.3). Dies liegt daran, dass er versucht seinen Gewinn mit den entdeckten Möglichkeiten zu maximieren und ihm das Feld mit der höheren Belohnug nicht bekannt ist. Er sollte aber auch nicht nur versuchen seine Umgebung auszukundschaften. Er könnte dadurch die Potenzialität verpassen sein eigentliches

Ziel umzusetzen: Den zu erwartenden Gewinn zu maximieren. Man spricht hierbei auch von dem *Exploration-Exploitation-Dilemma*.

2.1.6 Epsilon-Greedy-Strategie

Eine Möglichkeit eine Balance zwischen Ausbeutung (Exploitation) und Erkundung (Exploration) zu finden ist die *Epsilon-Greedy-Strategie*. Hierbei wird die Explorationsrate (*Exploration Rate*) ϵ eingeführt, welcher sich im Bereich zwischen [0,1] befindet. Hat ϵ einen Wert von null, erkundet der Agent ausschließlich seine Umgebung; bei eins „beutet“ er sie nur aus [8]. Um ein Gleichgewicht zwischen beiden zu finden, wird vor dem Lernprozess ein Wert für ϵ festgelegt und eine zufällige Zahl r auch im Bereich [0,1] erzeugt. Ist $r > \epsilon$ wählt der Agent für seinen aktuellen Zustand die Aktion mit dem höchsten Wert aus der Tabelle aus. Andernfalls wählt er zufällig eine aus. Für jedes State-Action-Paar wird die Q-Tabelle mit Hilfe des Bellman Optimalitätsprinzip aktualisiert, damit diese im besten Fall gegen den Optimalwert konvergiert. Dafür wird iterativ immer wieder der aktuelle mit dem optimalen Q-Wert verglichen (siehe Formel 2.13) und entsprechend aktualisiert, wenn der Agent wieder auf das gleiche State-Action-Paar trifft.

$$q_*(s,a) - q(s,a) = \text{loss} \tag{2.13}$$

$$\mathbb{E}[r_{t+1} + \gamma \max_{a'} q_*(s',a')] - \mathbb{E}[\sum_{k=0}^{\infty} \gamma^k + r_{t+k+1}] = \text{loss} \tag{2.14}$$

Das Ziel dieses Vorgehens ist den Verlust (*Loss*) zu reduzieren, damit sich der Wert immer mehr seinem Optimum annähert. Ein weiterer wichtiger Aspekt ist hierbei, wie schnell der Agent Gelerntes wieder vergisst, also wie wertvoll neue Erfahrungen sind, die er gemacht hat. Dafür wird ein weiterer Parameter eingeführt: Die Lernrate α. Sie befindet sich in einem Bereich zwischen [0,1], wobei der Agent bei einer Lernrate von eins alte Erfahrungen durch neue direkt ersetzt und bei null neue Erfahrungen gar keinen Einfluss haben [32]. Damit ergibt sich folgende Formel zur Aktualisierung der Q-Werte [32]:

$$q_{new}(s,a) = (1-\alpha) \underbrace{q(s,a)}_{\text{Alter Wert}} + \alpha \underbrace{(r_{t+1} + \gamma \cdot \max_{a'}(q(s',a')))}_{\text{Neue Erfahrung}} \tag{2.15}$$

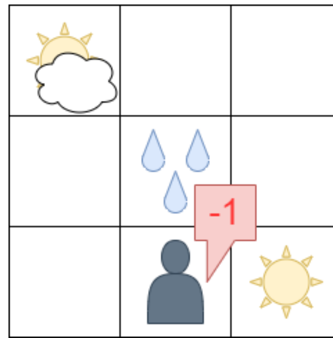


Abbildung 2.4: Agent betritt ein leeres Feld, welches als Konsequenz einen Punkt Abzug bedeutet.

Geht der Agent beispielsweise zu Beginn des Lernprozesses vom Startfeld auf das Feld rechts neben ihm (vgl. Abbildung 2.4), wird der Wert wie in Gleichung 2.16 berechnet (Anfangswerte mit null initialisiert, Lernrate α auf 0.7, Reduzierungsfaktor γ auf 0.99 festgelegt und der neue Wert ist -1).

$$\begin{aligned} q_{new}(s,a) &= (1-\alpha) \underbrace{q(s,a)}_{\text{Alter Wert} = 0} + \overbrace{\alpha}^{0.7} (\underbrace{r_{t+1}}_{\text{Neuer Wert} = (-1)} + \overbrace{\gamma}^{=0.99} \cdot \underbrace{\max_{a'}(q(s',a'))}_{\text{mit 0 initialisiert}}) \\ &= (1-0.7) \cdot 0 + 0.7 \cdot ((-1) + 0.99 \cdot 0) \\ &= 0.7 \cdot (-1) \\ &= (-0.7) \end{aligned} \tag{2.16}$$

Damit ergibt sich die aktualisierte Tabelle 2.2.

Dieses Verfahren wird für jeden Zeitschritt einer Episode wiederholt, bis der Prozess terminiert wird. Es lassen sich weitere Terminierungsgründe implementieren, damit die Episode nicht unnötig lange fortgeführt wird. Eine sinnvolle Randbedingung könnte hierbei die maximale Anzahl von Schritten sein, welcher der Agent pro Episode vollführen kann.

Tabelle 2.2: Aktualisierte Q-Tabelle des Beispiels nach Zeitschritt $t = 1$ der ersten Episode.

Aktionen Zustände	Hoch	Runter	Links	Rechts
Reward +1	0	0	0	0
Leer 1	0	0	0	0
Leer 2	0	0	0	0
Leer 3	0	0	0	0
End -10	0	0	0	0
Leer 4	0	0	0	0
Leer 5	0	0	0	-0.7
Leer 6	0	0	0	0
End +10	0	0	0	0

2.2 Deep Learning

Aufgaben, welche von Menschen mit Einfachheit erledigt werden können, wie beispielsweise das Erkennen von handgeschriebenen Ziffern, werden recht schwierig in der Beschreibung, gilt es, diese in Programmcode umzusetzen. Solch einen Algorithmus zu implementieren, beinhaltet eine Menge von bedingten Anweisungen und führt schnell zu komplexem und unübersichtlichem Code. Das Nutzen von neuronalen Netzen hingegen verfolgt einen anderen Ansatz: ein solches Netzwerk wird mit einer großen Anzahl von entsprechenden Daten trainiert, um eigenständig aus diesen Zusammenhänge und Regeln zu erlernen, welche diesem anschließend ermöglichen sollen, die gestellte Aufgabe wie Handschrifterkennung durchzuführen [25]. Im Nachfolgenden soll deshalb der Aufbau von neuronalen Netzen und deren Funtkionsweise erläutert werden.

2.2.1 Neuronales Netzwerk

Neuronale Netze bestehen aus einer Verbindung von mehreren künstliche Neuronen, welche die Funktionsweise menschlicher Neuronen nachahmen, in ihrem Aufbau jedoch wesentlich weniger komplex sind. Für einen besseren Überblick werden als erstes die künstlichen Neuronen näher erläutert, damit ein besseres Verständnis für die Funktionsweise von neuronalen Netzwerken geschaffen wird.

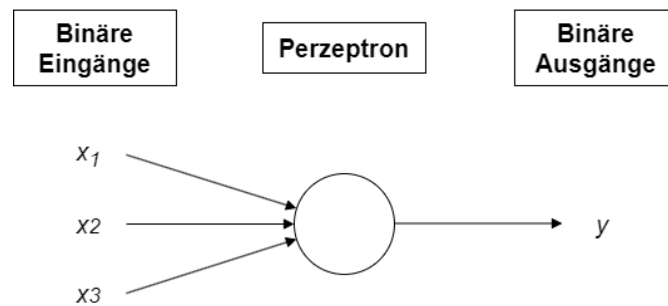


Abbildung 2.5: Einfaches Perzeptron mit drei binären Eingängen und einem binären Ausgang [25, modifiziert].

Perzeptron

Das Perzeptron ist ein in der Anwendung nicht mehr so verbreitetes künstliches Neuron, anhand dessen sich aber das grundlegende Funktionsprinzip gut erläutern lässt [25]. Ein Perzeptron kann mehrere binäre Eingänge $x_1, x_2, x_3, ...$ besitzen und produziert ein binäres Ausgangssignal y (vgl. Abbildung 2.5).

Der Wissenschaftler Frank Rosenblatt, welcher das Perzeptron in den 1950ern und 1960ern entwickelte, stellte zugleich eine einfache Berechnung für das Ausgangssignal auf, welche vor allem sogenannte Gewichte $w_1, w_2, w_3, ...$ beinhaltet, die die Wichtigkeit eines Neuroneneingangs ausdrücken [25]. Die aufsummierte Gewichtung $\sum_j w_j x_j$ der binären Eingänge muss für die Aktivierung der künstlichen Neuronen einen vorher festgelegten Schwellwert (*Threshold*) überschreiten. Wird dieser Wert überschritten, gibt dieses eine 1 aus, anderfalls eine 0:

$$z = \sum_j w_j x_j$$
$$y(z) = \begin{cases} 0: & z \leq \text{threshold} \\ 1: & z > \text{threshold} \end{cases} \tag{2.17}$$

Gewichte sowie Threshold sind beide aus dem reellen Zahlenbereich $\mathbb{R}$. In dem Buch „Neural Networks and Deep Learning“ von Micheal A. Niesen [25], schildert dieser ein vereinfachtes Beispiel, durch welches sich das Prinzip verdeutlichen lässt. Steht man beispielsweise vor einer Entscheidung an einer Aktivität teilzunehmen, könnte man dies von drei Bedingungen abhängig machen, welche für einen persönlich unterschiedliche

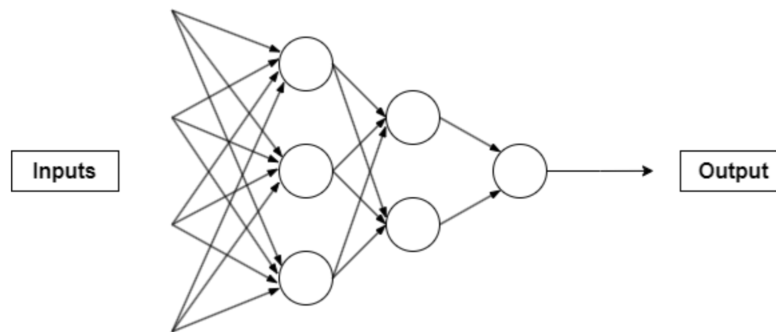


Abbildung 2.6: Einfaches Netzwerk aus mehreren Perzeptronen [25, modifiziert].

Gewichtung besitzen. Er stellt hierbei das Beispiel eines Festivals am Wochenende, dessen Besuch man an folgende Bedingungen knüpft:

1. Ist das Wetter gut?
2. Wird man von dem/der Partner:in begeleitet?
3. Ist dieses mit öffentlichen Verkehrsmitteln zu erreichen?

Diese Bedingungen sind die Eingänge des Perzeptrons und werden bei Erfüllung mit einer 1 und andernfalls mit einer 0 repräsentiert. Die Ausgabe des Perzeptrons hängt von der Gewichtung der Eingänge und vom Threshold ab. Würde man einen Grenzwert von 5 festlegen und bewertet die Anforderungen an die Festivität mit $w_1 = 6$, $w_2 = 2$, $w_3 = 2$, wird deutlich, dass das Wetter an diesem Tag die wichtigste Bedingung für die Teilnahme an der Aktivität darstellt und die anderen beiden nicht einmal kombiniert ausreichen, um den Schwellwert zu überschreiten. Durch Veränderung der Gewichtung oder des Thresholdes kann also auch die Ausgabe des Perzeptrons geändert werden.

Perzeptron-Netzwerk Durch die Verbindung mehrerer künstlicher Neuronen zu einem komplexen Netzwerk (vgl. Abbildung 2.6) steigt die Komplexität der Entscheidungen eines Netzwerkes.

Für die erste Schicht von Neuronen hatte Nielsen ein einfaches Beispiel für das Verständnis beschrieben. Die zweite Schicht kann man sich so vorstellen, dass die sich dort befindenden Neuronen eine Entscheidung basierend auf der Ausgabe der vorherigen Schicht treffen und ihre Eingabe wieder anders gewichten, wodurch diese komplexere sowie abstraktere Entscheidungen treffen können, deren Komplexität mit jeder weiteren Schicht

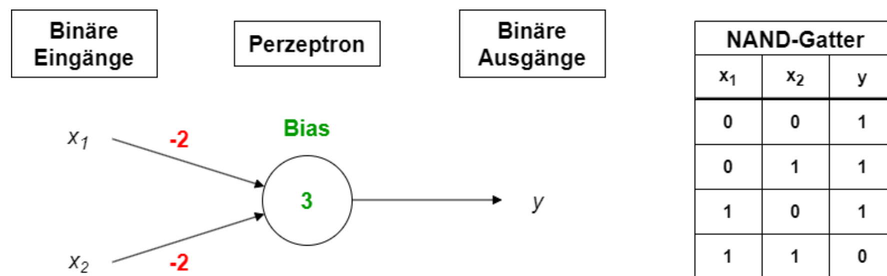


Abbildung 2.7: Perzeptron als NAND-Gatter [25, modifiziert].

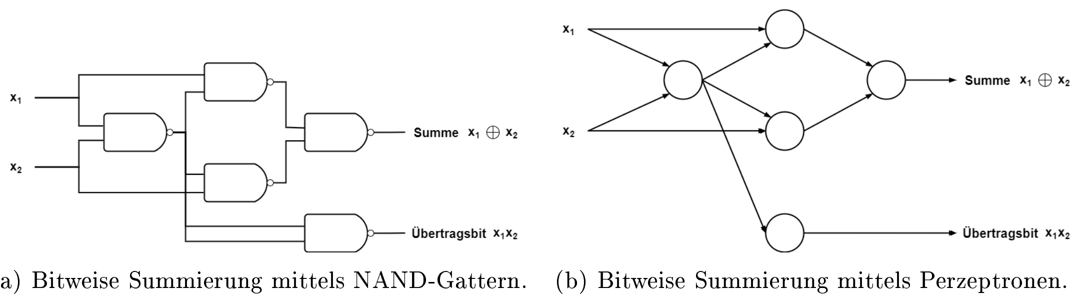
noch weiter zunehmen. Durch Anpassung der Formel 2.17 für die Aktivierung eines Perzeptrons kann diese noch weiter vereinfacht werden. Dafür wird die Summe durch das Skalarprodukt der entsprechenden Vektoren für x_j, w_j (Eingänge und deren Gewichtungen) ersetzt $\mathbf{w} \cdot \mathbf{x} = \sum_j w_j x_j$ und der Threshold wird auf die andere Seite der Gleichung übertragen, wodurch er zu dem besser bekannten *Bias* wird $b \equiv -threshold$ [25]. Damit ergeben sich folgende Regeln für das Perzeptron:

$$z = \mathbf{w} \cdot \mathbf{x} + b$$
$$y(z) = \begin{cases} 0: & z \leq 0 \\ 1: & z > 0 \end{cases} \tag{2.18}$$

Der Bias gibt dabei an, wie einfach ein Perzeptron aktiviert werden kann [25]. Künstliche Neuronen können zudem auch als elementare logische Funktionen betrachtet werden, wie in dem Beispiel in Abbildung 2.7 deutlich wird. Hierbei lässt sich die zugehörige Ausgabe mit Formel 2.18 berechnen. Für die Konfiguration $x_1 = 0$ und $x_2 = 0$ ergibt sich $(-2) \cdot 0 + (-2) \cdot 0 + 3 = 3$ und ist positiv. Ist nur einer der Eingänge aktiv, ist das Ergebnis der Formel noch immer positiv. Das Perzeptron bleibt inaktiv, solange beide Eingäge eingeschaltet sind.

Mit Hilfe von künstlichen Neuronen lässt sich jede Art von logischer Funktion erzeugen. Dabei ist das NAND-Gate besonders relevant für Berechnungen auf dem Computer. Mit NAND-Gattern kann beispielsweise das Summieren zweier Bits x_1, x_2 realisiert werden. Dafür ist die Berechnung der bitweisen Summe $x_1 \oplus x_2$ sowie die eines Übertragsbits erforderlich. Das Übertragsbit besitzt den Wert 1 sind beide Eingangsbit ebenfalls 1. Mit dem Perzeptron aus Abbildung 2.7 ergibt sich der Aufbau in Abbildung 2.8.

Die Gewichte und Bias von Perzeptronen lassen sich durch geeignete Lernalgorithmen wie das Gradientenabstiegsverfahren, welches im weiteren Verlauf der Arbeit näher er-



(a) Bitweise Summierung mittels NAND-Gattern. (b) Bitweise Summierung mittels Perzeptronen.

Abbildung 2.8: Vergleich zwischen einem Perzeptron-Netzwerk und Nand-Gattern bei der Umsetzung der bitweisen Summierung [25, modifiziert].

läutert werden soll, automatisch abstimmen, um diese für eine an das Netzwerk gestellte Aufgabe anzuzpassen. Dadurch müssen die Parameter wie bei einem Schaltkreis aus NAND-Gattern nicht händisch explizit ausgelegt werden, sondern das Netzwerk erlernt diese eigenständig. [25]

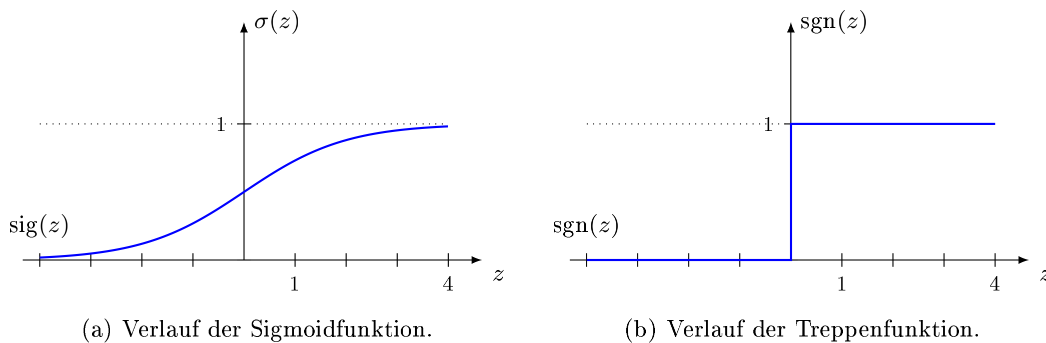
Sigmoid Neuron

Für das selbständige Erlernen der Gewichte und Bias ist es von Vorteil, wenn sich die Ausgabe eines neuronalen Netzwerkes minimal ändert, wird ein Parameter von diesem (ein Gewicht oder Bias) ebenfalls nur minimal geändert [25]. Durch schrittweise Anpassungen kann das Netzwerk somit dem gewünschten Verhalten angenähert werden. Das Problem mit Perzeptronen ist, dass diese oft auch bei nur kleinen Veränderungen ihren Ausgang vollständig umkehren, wodurch sich das Verhalten des ganzen Netzwerkes verändern kann. Die Lösung für dieses Problem ist die Nutzung einer anderen Art von künstlichem Neuron, dem Sigmoid Neuron. Dieses ist vom Aufbau identisch, unterscheidet sich jedoch darin, dass weder Eingänge noch Ausgang binär sind, sondern mit reellen Zahlen ($\mathbb{R}$) arbeiten. Der Ausgang lässt sich mit folgender Formel berechnen:

$$y = \sigma(w \cdot x + b) \tag{2.19}$$

$$\text{mit} \quad \sigma(z) \equiv \frac{1}{1 + e^{-z}} \tag{2.20}$$

Aus den Gleichungen kann man schließen, dass die Ähnlichkleit von Perzeptronen und Sigmoid Neuronen vor allem bei großen positiven oder negativen Eingangszahlen deut-



(a) Verlauf der Sigmoidfunktion.

(b) Verlauf der Treppenfunktion.

Abbildung 2.9: Zwei Funktionen zur Aktivierung eines künstlichen Neurons.

lich wird. Denn bei großem positiven Input wird $e^{-z} \approx 0$ und dadurch $\sigma(z) \approx 1$. Und bei großem negativen Input nähert sich $e^{-z} \to \infty$ und dadurch wird $\sigma(z) \approx 0$. Diese Ausgaben entsprechen für diese Fälle dem des binären Perzeptrons. In allen anderen Fällen unterscheiden sich die Ausgaben der beiden künstlichen Neuronen. Der Verlauf der Funktion ist in Grafik 2.9(a) dargestellt. Die Sigmoidfunktion ist eine geglätte Treppenfunktion, welche die Aktivierung eines Perzeptrons beschreibt (vgl. Abbildung 2.9(b)) [25]. Durch die Glättung der Funktion und die dadurch entstehenden Zwischenschritte ist es möglich, die Parameter eines Netzwerkes minimal anzupassen und dadurch auch den Ausgang eines künstlichen Neurons nur geringfügig zu ändern [25]. Somit können diese Parameter nach und nach optimiert werden, den gewünschten Anforderungen an das Netzwerk entsprechend. Dabei ist die Sigmoidfunktion nicht die einzige Möglichkeit ein künstliches Neuron zu aktivieren. Man spricht hierbei auch von Aktivierungsfunktionen, welche im nachfolgenden Abschnitt 2.2.5 ab Seite 40 detaillierter dargelegt werden sollen.

Architektur

Bisher wurden die Bestandteile eines neuronalen Netzes beschrieben, doch noch nicht die Terminologie des Netzes selbst. Ein Netzwerk lässt sich in unterschiedliche Schichten unterteilen, deren Name abhängig von deren Position ist. In Abbildung 2.10 befindet sich ganz links die Eingangs- und ganz rechts die Ausgangsschicht. Alle Schichten dazwischen werden auch als verborgene Schichten (*Hidden Layers*) bezeichnet, was nicht viel mehr bedeutet, als dass diese weder Eingangs- noch Ausgangsschicht sind [25]. Der gesamte Aufbau ist Abbildung 2.10 zu entnehmen. Das Auslegen der Anzahl von Neuronen für

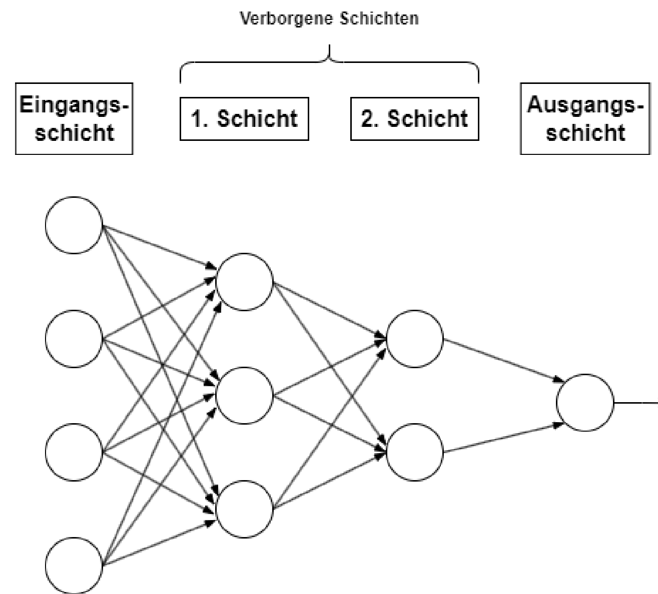


Abbildung 2.10: Neuronales Netzwerk mit den zugehörigen Schichtnamen [25, modifiziert].

Eingang sowie Ausgang gestaltet sich verhältnismäßig einfach. Nielsen gibt in seinem Buch [25] dafür als Beispiel die Handschrifterkennung einer 9 innerhalb eines 64x64 Pixel-Bildes an. Für dieses könnte man ein Netzwerk verwenden, welches $64 \cdot 64 = 4096$ Neuronen als Eingänge besitzt, denen der Graustufenwert des entsprechenden Pixels in einem Bereich von [0, 1] übergeben wird. Dieses Netz besäße dementsprechend nur einen Ausgang, welcher mit einem Wert ≥ 0.5 ausdrückt, dass es sich in dem Bild um eine 9 handelt. Die Gestaltung der verborgenen Schichten ist dagegen nicht so eindeutig und es gibt keine Faustregeln dafür [25]. Allerdings dauert das Anlernen eines Netzwerkes typischerweise länger, je mehr Schichten in diesem vorhanden sind. Bislang wurden nur sogenannte Feedforward-Netze dargestellt und beschrieben, welche die Ausgabe einer Schicht als Eingang für die nächste verwenden. Dies ist jedoch nicht die einzige Art von Netzwerk. Werden Ausgänge einer Schicht auch wieder als Eingang derselben oder vorheriger Schichten verwendet, spricht man von rekurrenten Netzwerken, mit Hilfe welcher sich dynamische Verhalten modellieren lassen, wie zum Beispiel ein Gedächtnis. Diese sind bis dato im Bereich des maschinellen Lernens weniger einflussreich gewesen, da die Lernalgorithmen weniger leistungsfähig sind [25].

2.2.2 Gradientenabstiegsverfahren

Der Kern des Lernes von einem Netzwerk ist der zugehörige Algorithmus, welcher die optimalen Gewichte und Bias für eine gestellte Aufgabe findet. Um das Netzwerk anzulernen, ist eine Menge von Trainingsdaten notwending, anhand welcher dieses erkennen kann, welche Ausgabe $y(x)$ für welche Eingabe x erforderlich ist. Diese Trainingsdaten sind mit der richtigen Ausgabe gelabelt und werden zur Korrektur der Netzwerkparameter verwendet. Das Netz erhält also als Eingabe ein Trainingsdatum und erzeugt eine Ausgabe a. Nun ist es notwendig festzustellen, wie gut die aktuelle Ausgabe a des Netzwerkes ist, um adäquate Anpassungen an den Gewichten und Bias vorzunehmen. Für die Quantifizierung der Ausgabe werden Kostenfunktionen $C(w, b)$ eingesetzt, welche im Abschnitt 2.2.5 auf Seite 47 genauer dargelegt werden. Das Ziel des Lernprozesses ist somit die Minimierung einer gewählten Kostenfunktion $C(w, b)$ und das Finden der Gewichte und Bias, welche dies ermöglichen. Für den Findungsprozess wird häufig das Gradientenabstiegsverfahren genutzt. Da sowohl die Gewichte w als auch die Bias b mehrdimensionale Vektoren seien können und sich die Findung eines globalen Minimums der Kostenfunktion $C(w, b)$ mittels Techniken der Analysis vor allem im höherdimensionalen Bereichen als äußert komplex heraustellt, zeigt Nielsen in seinem Buch ein vereinfachtes Beispiel, bei welchem die Kostenfunktion $C(v_1, v_2)$ nur von zwei Variablen v_1 und v_2 abhängt (vgl. Abbildung 2.11).

Dieses Beispiel lässt sich gedanklich auch auf Funktionen mit mehreren Variablen übertragen, bei welchen die Darstellung hingegen nicht mehr so simpel ist wie in Abbildung 2.11. Wie bereits erläutert, ist die Berechnung des globalen Minimums für den mehrdimensionalen Raum nicht so einfach wie in diesem Beispiel. Anhand eines zweidimensionalen Beispiels lässt sich das Gradientenabstiegsverfahren besser nachvollziehen. Man kann sich die Funktion aus Abbildung 2.11 wie ein Tal vorstellen, in welches man auf einem beliebigen Startpunkt einen Ball legt und anschließend dessen Bewegung abwärts dem Tal entlang simuliert. Diese Simulation geschieht durch die Berechnung der ersten und gegebenenfalls zweiten Ableitungen von C, welche ausreichend Auskunft über die „Beschaffenheit“ der Umgebung des Balles geben. Wird der Ball in Richtung v_1 um Δv_1 und in Richtung v_2 um Δv_2 bewegt, ändert sich auch C wie folgt:

$$\Delta C \approx \frac{\partial C}{\partial v_1}\Delta v_1 + \frac{\partial C}{\partial v_2}\Delta v_2 \tag{2.21}$$

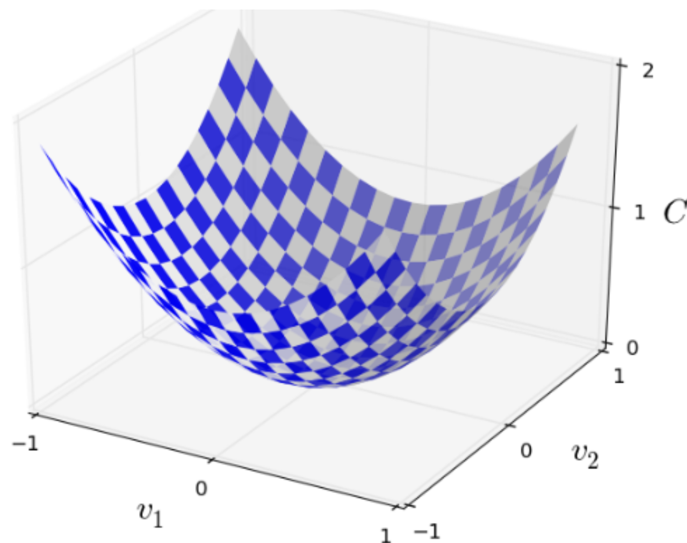


Abbildung 2.11: Darstellung einer vereinfachten Kostenfunktion $C(v_1, v_2)$ im zweidimensionalen Raum [25].

Durch die richtige Wahl der beiden Bewegungen $\Delta v_1, \Delta v_2$ wird ΔC negativ und der Ball „rollt das Tal hinab“. Dabei ist es einfacher Δv als Vektor der Änderungen von v zu definieren: $\mathbf{\Delta v} \equiv (\Delta v_1, \Delta v_2)^T$. Durch Definition des Gradienten von C als Vektor seiner partiellen Ableitungen ergibt sich

$$\nabla C \equiv \begin{pmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_2} \end{pmatrix} \tag{2.22}$$

woraus folgt:

$$\Delta C \approx \nabla C \cdot \mathbf{\Delta v} \tag{2.23}$$

Der Gradient zeigt somit als Ableitung in die Richtung des stärksten Anstiegs. Um zu einem Minimum zu gelangen, bewegt man sich deshalb in die entgegengesetzte Richtung. Formel 2.23 zeigt auf, dass Veränderungen von ∇C Veränderungen in C hervorufen und somit lässt sich bestimmen, wie $\mathbf{\Delta v}$ gewählt werden muss, um ein negatives ΔC zu erhalten:

$$\mathbf{\Delta v} = -\eta \nabla C \tag{2.24}$$

Wobei η, besser bekannt als Lernrate, ein kleiner positiver Parameter ist und Einfluss auf die Größe der Schrittweite der Bewegungen hat. Durch Einsetzen von Formel 2.24 in

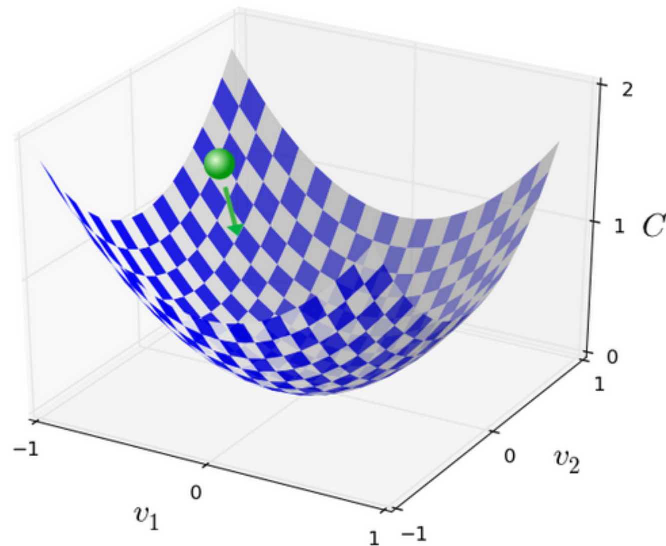


Abbildung 2.12: Darstellung einer vereinfachten Kostenfunktion $C(v_1, v_2)$ im zweidimensionalen Raum mit einem „Ball der hinabrollt“ [25].

2.23 ergibt sich der Zusammenhang:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta ||\nabla C||^2 \geq 0 \tag{2.25}$$

Aufgrunddessen, dass $||\nabla C||^2$ immer größer als null ist, wird C immer weiter abnehmen, wenn $\Delta \mathbf{v}$ entsprechend Formel 2.24 gewählt wird. Anschließend wird der Ball um den berechnten Wert für $\Delta \mathbf{v}$ bewegt:

$$v \rightarrow v' = v - \eta \nabla C \tag{2.26}$$

Anschließend nutzt man die Aktualisierungsregelung erneut, um die nächste Bewegung durchzuführen. Und dies wird solange wiederholt, bis ein (bestenfalls das globale) Minimum erreicht worden ist [25]. Beim Gradientenabstiegverfahren wird also wiederholt der Gradient ∇C berechnet und anschließend eine Bewegung in die entgegengesetzte Richtung getätigt. Das Beispiel mit dem Ball ist in Abbildung 2.12 noch einmal nachvollziehbar dargestellt.

Vorraussetzung für eine korrekte Funktionsweise des Verfahrens ist es die Lernrate η klein genug zu wählen, damit Gleichung 2.23 eine gute Annäherung ist. Auf der anderen

Seite sollte η nicht zu klein gewählt werden, da die Veränderungen in $\Delta\mathbf{v}$ ebenfalls sehr gering ausfallen würden und somit das gesamte Gradientenabstiegsverfahren verlangsamt wird [25]. Durch eine zu kleine Lernrate wird des Weiteren das Hängenbleiben in einem lokalen Minimum begünstigt. Das aufgezeigte Beispiel von Nielsen erläuterte das Verfahren mit einer Kostenfunktion, welche nur von zwei Variablen abhängig ist. Aber dieses funktioniert genauso für Funktionen im höherdimensionalen Bereich $C(v_1, v_2, ..., v_n)$:

$$\nabla C \equiv \begin{pmatrix} \frac{\partial C}{\partial v_1} \\ \frac{\partial C}{\partial v_n} \end{pmatrix} \tag{2.27}$$

Die Aktualisierungsregelung 2.26 ist dieselbe und durch wiederholtes Anwenden führt dieses zu einem (bestenfalls das globale) Minimum von C. Um das Gradientenabstiegsverfahren für die Findung der (optimalen) Gewichte und Bias eines neuronalen Netzwerkes zu nutzen, werden die Formeln und die Aktualisierungsregel enstprechend angepasst, sodass sich ergibt:

$$w_k \rightarrow w_k' = w_k - \eta \frac{\partial C}{\partial w_k} \tag{2.28}$$

$$b_l \rightarrow b_l' = b_l - \eta \frac{\partial C}{\partial b_l} \tag{2.29}$$

Ein Problem des Gradientenabstiegsverfahrens ist, dass der Gradient ∇C_x theoretisch für jedes Trainingsdatum X einzeln berechnet werden muss und anschließend der Mittelwert gebildet wird [25]. Da dies sehr langsam ist, kann der stochastische Gradientenabstieg verwendet werden, um dem Problem entgegenzuwirken. Dieses schätzt den Gradienten ∇C durch die Berechnung von ∇C_X für eine kleine Menge X zufälliger Trainingsdaten. Die berechneten Gradienten werden wiederum aufsummiert und gemittelt, wodurch eine gute Schätzung des eigentlichen Gradienten erreicht werden kann:

$$\frac{1}{m} \sum_{j=1}^{m} \nabla C_{X_j} \approx \frac{1}{n} \sum_{x} \nabla C_x = \nabla C \tag{2.30}$$

Die Menge der zufälligen Daten wird auch als (Mini-)Batch bezeichnet. Mit Formel 2.30 ergeben sich für die Aktualiserung der Gewichte und Bias:

$$w_k \rightarrow w'_k = w_k - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial w_k} \tag{2.31}$$

$$b_l \rightarrow b'_l = b_l - \frac{\eta}{m}\sum_j \frac{\partial C_{X_j}}{\partial b_l} \tag{2.32}$$

Das Netzwerk wird wiederholt mittels zufällig ausgewählten (Mini-)Batches X_j aus den Trainingsdaten trainiert, bis alle Trainingsdaten verwendet wurden [25]. Dies entspricht einer *Epoche* des Trainings. In jeder Epoche werden die Daten zu neuen Batches gruppiert und das Verfahren beginnt von Neuem.

2.2.3 Backpropagation

Um das Kapitel des Gradientenabstieges zu ergänzen, gilt es noch zu erläutern, wie der Gradient einer Kostenfunktion berechnet werden kann. Hierfür wird das Verfahren der Backpropagation (*Fehlerrückführung*) eingesetzt. Wie bereits aus den Formeln im vorherigen Abschnitt hervorgeht, ist dafür vor allem die Berechnung der partiellen Ableitungen eines jeden Gewichtes $\frac{\partial C}{\partial w}$ und der Bias $\frac{\partial C}{\partial b}$ im Netzwerk notwendig. Es ist zweckmäßig zunächst ein Verständnis für die Notation zu schaffen, mit welchem sich einzele Gewichte (beziehungsweise Bias) beschreiben lassen. [25]

Notation

Es ist üblich, ein Gewicht des Netzwerkes über die Schicht l in der es sich befindet und der Position des Neuron k der vorherigen Schicht $l-1$, welches mit einem Neuron j der aktuellen Schicht l verbunden wird, zu beschreiben. Damit ergibt sich als Notation für ein Gewicht: w^l_{jk}. Für die Bias und die Aktivierung eines Neurons ergibt sich eine ähnliche Notation. Hier wird wieder mit l die Schicht beschrieben und mit j die Position in dieser. Für eine bessere Nachvollziehbarkeit ist die Notation in Abbildung 2.13 dargestellt. Die Aktivierung eines Neurons a_j in einer Schicht l des Netzwerkes ist abhängig von den Aktivierungen in der vorherigen Schicht $(l-1)$ aller der sich dort befindenden künstlichen Neuronen, welche multipliziert werden mit den zugehörigen Gewichten und anschließend

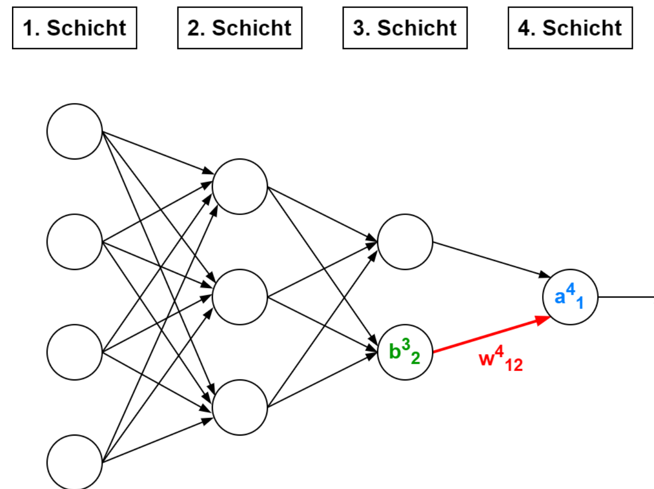


Abbildung 2.13: Beispiel, um die Notationsweise von Bias, Gewicht und Aktivierung aufzuzeigen [25, modifiziert].

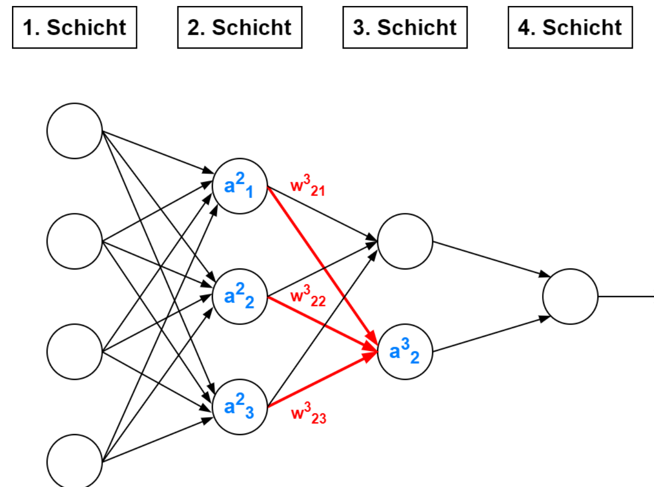


Abbildung 2.14: Zusammenhang zwischen einem Neuron und den Neuronen der vorherigen Schicht [25, modifiziert].

aufsummiert werden (vgl. Abbildung 2.14):

$$a_j^l = \sigma\left(\sum_k w_{jk}^l a_k^{l-1} + b_j^l\right) \tag{2.33}$$

Der Zusammenhang der Neuronen einer Schicht l zur der vorherigen Schicht $l-1$ aus Gleichung 2.33 lässt sich in Matrixschreibweise darstellen. Dafür wird eine Gewichtsmatrix

$\underline{\mathbf{W}}^l$ für jede Schicht definiert, deren Koeffizienten aus den Gewichten bestehen, welche die $(l-1)$-te Schicht mit der l-ten Schicht verbinden. Die Größe der Gewichtsmatrix ist dabei abhängig von der Anzahl der Neuronen, die sich in den beiden verbundenen Schichten befinden. Besitzt die l-te Schicht N Neuronen und die $(l-1)$-te Schicht M Neuronen, dann handelt es sich um eine $N \times M$-Matrix. Durch zusätzliches Vektorisieren der restlichen Variablen in Gleichung 2.33 und der genutzten Aktivierungsfunktion, hier σ, wodurch diese auf jedes Element eines Vektors angewendet wird, ergibt sich die kompakte Schreibweise in Gleichung 2.34. [25]

$$\mathbf{a}^l = \sigma(\underline{\mathbf{W}}^l \mathbf{a}^{l-1} + \mathbf{b}^l) \tag{2.34}$$

$$\text{mit} \quad \underline{\mathbf{W}}^l = \begin{bmatrix} w_{11} & w_{12} & \cdots & w_{1k} \\ w_{21} & w_{22} & \cdots & w_{2k} \\ \vdots & \vdots & \ddots & \vdots \\ w_{j1} & w_{j2} & \cdots & w_{jk} \end{bmatrix}$$

Nielsen erläutert, dass durch diesen Ausdruck die Verständlichkeit der Vorgehensweise verdeutlicht wird:

1. Anwenden der Gewichtsmatrix auf die Aktivierungen in Schicht $(l-1)$.
2. Addieren der Bias.
3. Anwenden der Aktivierungsfunktion, hier: σ.

Erwähnenswert ist außerdem, dass die Zwischenmenge mit $\mathbf{z}^l \equiv (\underline{\mathbf{W}}^l \mathbf{a}^{l-1} + \mathbf{b}^l)$ notiert und als *gewichteter Input* bezeichnet wird. Dadurch vereinfacht sich die Gleichung 2.34 zu:

$$\mathbf{a}^l = \sigma(\mathbf{z}^l) \tag{2.35}$$

Funktionsweise

Backpropagation handelt von dem Verständnis, wie das Verändern von Gewichten und Bias in einem Netzwerk die Ausgabe der Kostenfunktion verändert. Somit ist das Ziel die Berechnung der partiellen Ableitungen $\frac{\partial C}{\partial w^l_{jk}}$ und $\frac{\partial C}{\partial b^l_j}$ der Kostenfunktion für jedes Gewicht und jeden Bias. Nielsen erläutert, damit die Fehlerrückführung funktioniere, müssen zuvor zwei Annahmen getroffen werden: [25]

1. Die Kostenfunktion lässt sich als Mittelwert über alle Kostenfunktionen C_X der einzelnen Trainingsdaten schreiben: $C = \frac{1}{n}\sum_X C_X$
2. Die Kostenfunktion kann als Funktion der Netzwerkausgänge beschrieben werden: $C = C(\mathbf{a}^L)$

Die erste Annahme hierbei sei notwending, da Backpropagation eigentlich die partiellen Ableitungen $\frac{\partial C_X}{\partial w}$ und $\frac{\partial C_X}{\partial b}$ für ein einzelnes Trainingsdatum berechne und diese anschließend gemittelt werden müssen um $\frac{\partial C}{\partial w}$ und $\frac{\partial C}{\partial b}$ zu erhalten. Die zweite Annahme beschreibt die vermeintliche Abhängigkeit der Funktion von den einzelnen Parametern, da diese die Aktiverungen der letzten Netzwerkschicht a^L maßgeblich beeinflussen. Dabei sei die Kostenfunktion nicht abhängig von y, da dieses einen fixen Wert besitzt und nicht durch Änderungen die im Netz vorgenommen werden, beeinflusst werden kann. Um die partiellen Ableitungen $\frac{\partial C}{\partial w^l_{jk}}$ und $\frac{\partial C}{\partial b^l_j}$ für jedes Neuron in dem Netzwerk zu ermitteln, führt Nielsen zunächst eine weitere Zwischenmenge ein: den Fehler δ^l_j, welcher sich am j-ten Neuron der l-ten Schicht befinde. Mit Hilfe der Fehlerrückführung ließe sich dieser Fehler bestimmen und anschließend Rückschlüsse auf die entsprechenden Gewichte und Bias ziehen. Der Fehler δ^l_j lässt sich so verstehen, dass dieser eine minimale Änderung an dem gewichteten Input Δz^l_j eines Neurons hervoruft, wodurch der Output $\sigma(z^l_j)$ von diesem ebenfalls geändert wird $\sigma(z^l_j + \Delta z^l_j)$. Dieser Fehler zieht sich durch das gesamte Netzwerk und hat somit Einfluss auf die Kostenfunktion $\frac{\partial C}{\partial z^l_j}\Delta z^l_j$. Besitzt $\frac{\partial C}{\partial z^l_j}$ einen hohen (positiven oder negativen) Wert, können die Kosten stark verringert werden, indem Δz^l_j so gewählt wird, dass es das entgegengesetzte Vorzeichen von $\frac{\partial C}{\partial z^l_j}$ aufweist [25]. Ist $\frac{\partial C}{\partial z^l_j}$ nah an dem Wert null, kann kaum eine Änderung mittels Δz^l_j erreicht werden und die Parameter des Neuron befinden sich bereits in einem guten Wertebereich. Somit ist $\frac{\partial C}{\partial z^l_j}$ in einem heuristischen Sinne bereits eine Messung für den Fehler eines Neurons und lässt sich wie folgt definieren:

$$\delta^l_j \equiv \frac{\partial C}{\partial z^l_j} \tag{2.36}$$

Anhand δ^l_j lässt sich also feststellen, wie Δz^l_j ausgelegt werden muss, damit die Kosten verringert werden können. Hierbei ist δ^l wieder ein Vektor, welcher mit der Schicht l assoziiert wird. Nielsen erläutert, dass vier Gleichungen für das Verfahren der Backpropagation notwendig sind, mittels derer der Fehler δ^l und der Gradient der Kostenfunktion berechnet werden können.

1. Fehler in der Ausgangsschicht δ^L

Die Bestandteile für δ^L sind gegeben durch:

$$\delta_j^l = \frac{\partial C}{\partial a_j^L} \sigma'(z_j^L) \tag{2.37}$$

Wobei $\frac{\partial C}{\partial a_j^L}$ angibt, wie schnell sich die Kosten in Abhängigkeit von Neuron j verändern. Hängt C nicht stark von einem Ausgangsneuron ab, ist δ_j^l beispielsweise klein. $\sigma'(z_j^L)$ beschreibt wie schnell sich die Aktivierungsfunktion (hier σ) an z_j^L ändert. Die exakte Form von $\frac{\partial C}{\partial a_j^L}$ ist an die gewählte Kostenfunktion gebunden. [25] Gleichung 2.37 lässt sich wieder in Matrixschreibweise umformulieren, was sinnvoll ist, da die gesamte Fehlerrückführung auf Rechenoperationen mit Matrizen beruht.

$$\delta^L = \nabla_a C \odot \sigma'(\mathbf{z}^L) \tag{2.38}$$

In Formel 2.38 ist $\nabla_a C$ als Vektor definiert, welcher die partiellen Ableitungen $\frac{\partial C}{\partial a_j^L}$ beinhaltet und als Änderungsrate von C in Bezug auf die Aktivierungen der Ausgangsschicht verstanden werden kann. Des Weiteren wird in der Gleichung das Hadamard-Produkt $\odot$ [24] verwendet, welches eine elementweise Multiplikation der Vektoren beschreibt.

2. Fehler δ^l in Bezug auf den Fehler der nachfolgenden Schicht δ^{l+1}

$$\delta^l = ((\underline{\mathbf{W}}^{l+1})^T \delta^{l+1}) \odot \sigma'(\mathbf{z}^l) \tag{2.39}$$

Mit $(\underline{\mathbf{W}}^{l+1})^T$ als Gewichtsmatrix und δ^{l+1} als Fehler der $l + 1$ Schicht. Ist der Fehler der nächsten Schicht bekannt, so könne man sich vorstellen, dass dieser durch das Netzwerk zurückgeführt wird, zu der aktuellen Schicht l. Dies geschieht durch die Multiplikation mit der Gewichtsmatrix $(\underline{\mathbf{W}}^{l+1})^T$. Durch das anschließende Berechnen des Hadamard-Produktes mit der Änderungsrate der Aktivierungsfunktion, wird der Fehler weiter durch die Aktivierung geführt, wodurch man den Fehler δ^l des gewichteten Inputs in der Schicht l erhält. Indem zunächst der Fehler der Output-Schicht über Gleichung 2.37 ermittelt und anschließend mit Gleichung 2.39 weiter durch das Netz geführt wird, lässt sich so der Fehler δ^l für jede Schicht eines Netzes berechnen.

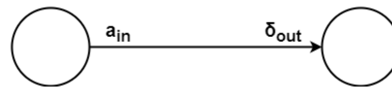


Abbildung 2.15: Verdeutlichung der Gleichung 2.43 [25, modifiziert].

3. Änderungsrate der Kostenfunktion in Bezug zu den Bias in einem Netzwerk

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{2.40}$$

Der Fehler δ_j^l enstpricht also der Änderungsrate und lässt sich somit über die Gleichungen 2.37 und 2.39 berechnen. Wird δ_j^l am selben Neuron evaluiert wie der Bias, lässt sich Formel 2.40 ohne Indexierung darstellen:

$$\frac{\partial C}{\partial b} = \delta \tag{2.41}$$

4. Änderungsrate der Kostenfunktion in Bezug zu den Gewichten in einem Netzwerk

$$\frac{\partial C}{\partial w_{jk}^l} = a_k^{l-1} \delta_j^l \tag{2.42}$$

Mit Gleichung 2.42 können die partiellen Ableitungen $\frac{\partial C}{\partial w_{jk}^l}$ in Bezug auf den Fehler δ_j^l und die Aktivierungen der vorherigen Schicht a_k^{l-1} berechnet werden. Diese lässt sich wieder mit verringerter Indexierung darstellen:

$$\frac{\partial C}{\partial w} = a_{in} \delta_{out} \tag{2.43}$$

Mit a_{in} als Aktivierung des Neurons, welches als Eingabe für das Gewicht w verwendet wird und δ_{out} als Fehler, welcher von diesem Gewicht erzeugt wird (vgl. Abbildung 2.15).

Aus Formel 2.43 wird ersichtlich, dass die Gewichte von Neuronen mit geringer Aktivierung a langsamer „lernen“. Dies liegt darin begründet, dass der zughörige Gradient $\frac{\partial C}{\partial w}$ klein wird, ist die Aktivierung a_{in} gering.

Zusammenfassung

Aus Gleichung 2.37 wird ersichtlich, dass die Neuronen einer Schicht nur sehr langsam lernen, wenn die Änderungsrate $\sigma'(\mathbf{z}^L)$ der Ausgangsschicht null beziehungsweise die Aktiverungsfunktion $\sigma(\mathbf{z}^L)$ hoch (≈ 1) oder niedrig (≈ 0) ist. Man spricht auch von „gesättigten“ Neuronen. Selbiges gilt auch für die Bias. In Gleichung 2.39 wird deutlich, dass der Fehler δ_j^l gering wird, wenn ein Neuron sich nah an der Sättigung befindet. Und dies führt wieder dazu, dass Gewichte, welche zu einem *gesättigten* Neuron geführt werden, langsam lernen. Besitzt also das Eingangsneuron eine niedrige Aktivierung oder ist das Ausgangsneuron gesättigt, kann das Gewicht dazwischen nichts mehr dazulernen. Die vier Formeln sind auch für andere Aktivierungsfunktionen gültig und können genutzt werden um Aktivierungsfunktionen zu designen, welche gewünschte Eigenschaften besitzen sollen. Die zugehörigen Beweise zu den Formeln lassen sich in Nielsens Buch [25] finden und nachvollziehen.

2.2.4 Gewichtsinitialisierung

Die anfänglichen Gewichte eines Netzwerkes richtig zu initialiseren hat viele Vorteile für den Lernprozess. Einerseits verursacht ein ungünstiges Initialiseren, dass Neuronen von Beginn an zur Sättigung geführt werden können und die zugehörigen Gewichte nur noch langsam oder gar nicht lernen und den ganzen Prozess dadurch verlangsamen würden. Andererseits begünstigen schlecht gewählte Anfangsgewichte Komplikationen, wie das Problem der Vanishing oder Exploding Gradients, welche in Abschnitt 2.2.6 auf Seite 49 näher erläutert werden. Diese beinhalten, dass die Verlustfunktion beim Trainieren eines Netzes auf einmal beginnt gegen null zu sinken oder stark zu divergieren und das Netzwerk infolgedessen keine sinnvollen Ergebnisse mehr produziert. Dies liegt in den wiederholten Matrixmultiplikationen begründet. Sind die Werte der Gewichtsmatrizen zu klein oder zu groß, können diese die beschriebenen Verhaltensweisen hervorrufen, wodurch das neuronale Netz nicht mehr in der Lage ist Fortschritte zu machen.

Xavier-Initialisierung

Eine der meistgenutzen Initialisierungstechniken ist die Xavier oder auch Glorot-Initialisierung. Sie wurde entwickelt, um dem Vanishing/Exploding Gradient Problem der

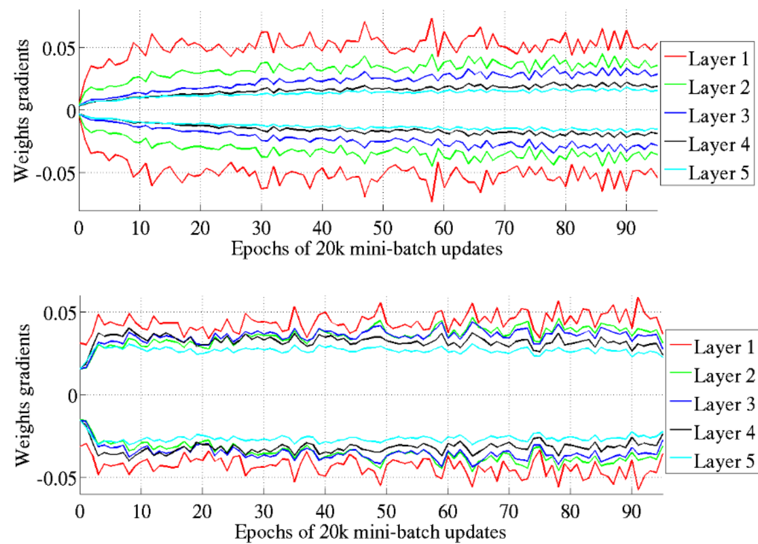


Abbildung 2.16: Oben: Initialisierung der Gewichte mit Gleichverteilung. Unten: Initialisierung der Gewichte nach Glorot und Bengio [6].

Sigmoid- und tanh-Aktivierungsfunktionen in Verbindung mit einer Gleichverteilungsinitialisierung [-1,1] entgegenzuwirken [6]. Glorot und Bengio schlagen deshalb eine andere Art der Initialisierung vor, welche jetzt als Xavier bzw. Glorot Initialization bekannt ist. Die Gewichte werden dabei zufällig aus dem Bereich der Gaußverteilung gewählt; mit einem Mittelwert von 0.0 und einer Varianz nach Formel

$$\sigma^2(w_i) = \pm\frac{\sqrt{6}}{\sqrt{n_i + n_{i+1}}} = \frac{1}{fan_{in} + fan_{out}}. \tag{2.44}$$

n_i bezieht sich dabei auf auf die Nummer der eingehenden (fan_{in}) und n_{i+1} auf die der ausgehenden (fan_{out}) Netzwerkverbindungen. Glorot und Bengio sind in ihrem Artikel [6] der Auffassung, dass ihre Strategie die Varianz der Aktivierung und die zurückgeführten Gradienten in den Schichten eines Netzwerkes aufrechterhalten. Bei einem Experiment mit einem fünfschichtigen Netzwerk beobachteten sie, dass es diesem mit Hilfe Ihrer Initialisierungstechnik möglich war, die Varianz der Gewichtsgradienten über alle Schichten hinweg beizubehalten. Die Initialisierung über eine Gleichverteilung, welche zuvor standardmäßig genutzt wurde, besitzt hierbei viel größere Unterschiede zwischen der Varianz in den unteren und den oberen Schichten des Netzes. Die Ergebnisse ihres Experimentes sind Abbildung 2.16 zu entnehmen.

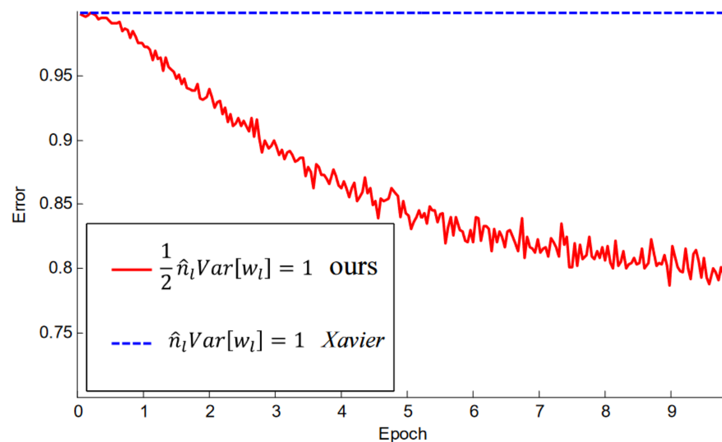


Abbildung 2.17: Training eines neuronalen Netzwerkes mit 30 Schichten und ReLU-Aktivierungsfunktionen. Gewichtsinitialisierung nach Glorot und Bengio in Blau und nach He et al in rot [13].

Kaiming-Initialisierung

Glorot und Bengio entwickelten ihre Glorot-Initialisierung vor allem für Aktivierungsfunktionen wie Sigmoid oder tanh, da diese zu der Zeit am häufigsten verwendet wurden. Diese wurde im Laufe jedoch vermehrt von ReLU-Funktionen abgelöst, weil solche bereits gute Maßnahmen gegen das Vanishing beziehunsgweise Exploding Gradient Problem sind. Es stellte sich jedoch heraus, dass die Xavier-Initialisierung für ReLU-Aktivierungsfunktionen nicht gut geeignet ist (vgl. Abbildung 2.17). Aufgrund dessen puplizierten 2015 He et al. einen Artikel [13], wo sie eine der Xavier-Methode sehr ähnliche Gewichtsinitialisierungstechnik vorstellen. Auch diese stützt sich auf die Grundidee, die Gewichte unter Berücksichtigung der Größe der vorherigen Schicht zu initialisieren:

$$\sigma^2(w_i) = \frac{2}{fan_{in}} \tag{2.45}$$

In Formel 2.45 ist fan_{in} wieder als die eingehenden Netzwerkverbindungen definiert. In einem Versuch, in welchem He et al. ein Convolutional Neural Network mit 30 Schichten trainierten, lernte dieses nur bei der Verwendung ihrer eigenen Initialisierungsmethode und mit Hilfe der Xavier Initialisrung gar nicht. Die Versuchsergebnisse sind in Abbildung 2.17 dargestellt.

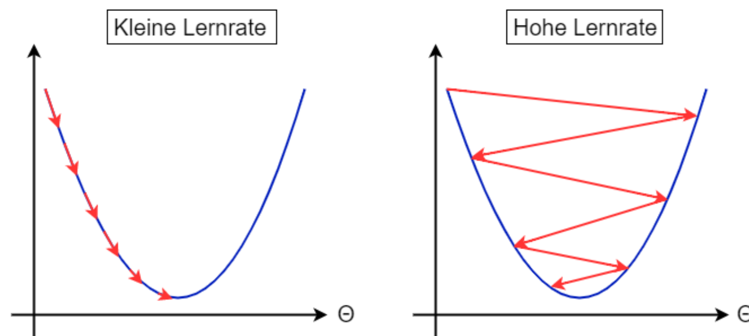


Abbildung 2.18: Gradientenabstieg mit sehr kleiner (links) und großer (rechts) Lernrate.

2.2.5 Hyperparameter

Parameter, welche nicht während des Trainings des Netzes erlernt werden, werden auch als Hyperparameter bezeichnet. Die richtige Auswahl der Hyperparameter für ein Modell ist ausschlaggebend für den Lernerfolg und die anschließende Leistung. Im nachfolgenden sollen die gängigsten kurz erläutert werden.

Lernrate η

Einer für den Trainingserfolg eines neuronalen Netzes entscheidender Parameter ist die auf Seite 24 eingeführte Lernrate. Sie bestimmt die Lerngeschwindigkeit eines Modells und reguliert wie stark die Gewichte und Bias des trainierten Netzwerkes angepasst werden. Je kleiner diese gewählt wird, desto kleiner sind die Schritte, die beim Gradientenabstiegsverfahren getätigt werden, und um so langsamer ist der Lernprozess. Wird sie jedoch zu groß gewählt, hindert sie die Konvergenz zu einem (globalen) Minimum und kann sogar zu Divergenz führen [29]. Der Einfluss der Lernrate auf das Gradientenabstiegsverfahren ist anschaulich in Abbildung 2.18 dargestellt. Oft wird fälschlicherweise angenommen, dass η der Schrittweite bei der Gradientenoptimierung entspricht. Diese wird allerdings erst durch eine Multiplikation von der Lernrate mit dem Gradienten ∇C berechnet (vgl. Gleichung 2.24 auf Seite 24).

Somit ist es wichtig, dass die optimale Rate vor dem Lernprozess gefunden und gegebenenfalls währendessen noch angepasst wird (vgl. Abschnitt Lernratenscheduler). Um die anfänglich optimale Lernrate zu ermitteln, kann zunächst eine sehr kleine Rate eingestellt werden, welche innerhalb kürzerer Abstände schrittweise vergrößert wird. Ist sie zu klein,

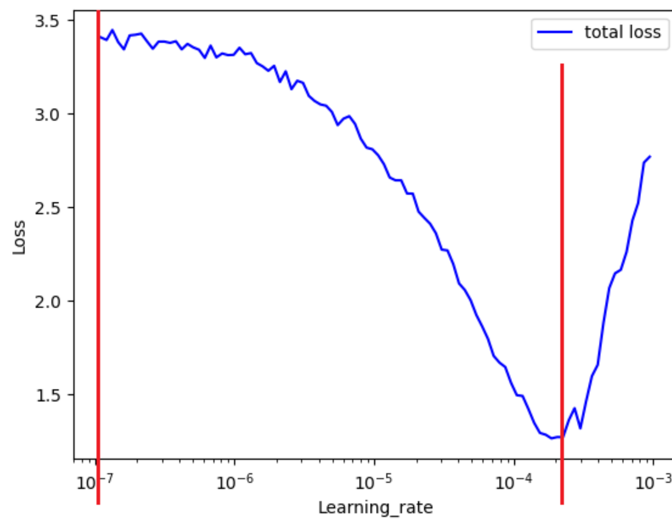


Abbildung 2.19: Gesamtverlust bei der Erhöhung der Lernrate in jeder Iteration um den Faktor $1,1$.

ist in Abbildung 2.19 ersichtlich, dass das neuronale Netz nur sehr langsam lernen kann. Zudem können bei einer zu geringen Lernrate lokale Minima nicht übersprungen werden und das Training beginnt zu stagnieren. Ist sie zu groß, beginnt die Verlustfunktion stark zu divergieren. Die optimale Lernrate befindet sich zwischen dem Start des Abfallens und kurz vor dem Divergierungsprozess (rot markiert in Abbildung 2.19) [15]. Häufig wird der Punkt des steilsten Abstiegs in diesem Bereich als Lernrate eingestellt. Eine andere Strategie besteht darin, die Lernrate zu halbieren oder zu zehnteln, wenn der Verlust anfängt, stark anzusteigen.

Lernratenscheduler

Um die Lernrate während des Trainings anzupassen, kann ein Lernratenscheduler verwendet werden. Diese passen die Größe der Lernrate während des Lernprozesses an, wenn beispielsweise die Kostenfunktion ein Threshold unterschreitet oder eine voreingestellte Anzahl von Trainingsepochen überschritten wird [29]. Eine weitere Herausforderung bei dem Training eines Netzes ist das Minimieren von konkaven Fehlerfunktionen, ohne in ihren zahlreichen lokalen Minima gefangen zu bleiben.

Häufig verwendete Scheduler sind:

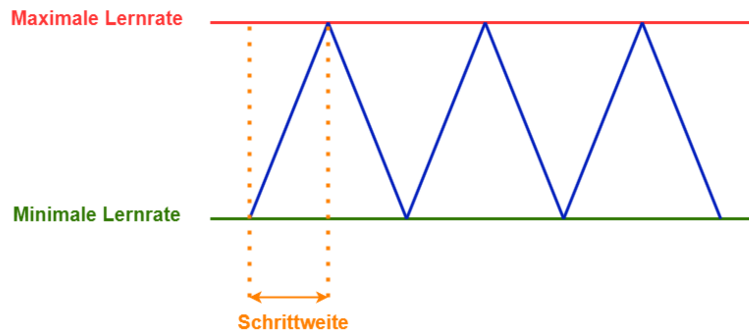


Abbildung 2.20: Zyklische Lernrate in Dreiecksform (blau) [31, modifiziert].

- Step Scheduler: Verringert nach einer vordefinierten Anzahl von Epochen oder Iterationen die Lernrate um einen Faktor Gamma γ.
- Adaptive Scheduler: Reduziert die Lernrate beim Erreichen eines Lernplateaus.
- Cyclical Scheduler: Reduziert und erhöht die Lernrate in einer bestimmten Periode. Beispiel: Lenratenänderung in Sinus- oder Dreiecksform.

Die bisherige Konvention war, dass es von Vorteil ist, wenn die Lernrate beim Trainieren eines neuronalen Netzes mit voranschreitender Zeit verringert wird. Quelle [3] argumentiert hingegen, dass wenige Fortschritte beim Lernprozess nicht mit schlechten lokalen Minima zu tun haben, sondern mit Sattelpunkten, also Punkten, an denen eine Dimension nach oben und eine andere nach unten geneigt ist. Dort befinden sich meist Regionen desselben Fehlerniveaus, die es dem SGD erschweren, diese zu verlassen, da der Gradient in allen Dimensionen nahezu null ist [29]. Die Gradienten am Sattelpunkt seien zu klein und verlangsamen der Lernfortschritt. Durch eine Korrektur der Rate nach oben, könnten Sattelpunkte schneller verlassen werden, weshalb sich zyklische Lernraten besonders für das Training von neuronalen Netzen eignen. Dabei werden eine maximale und eine Basisrate eingestellt, zwischen denen der Scheduler langsam wechselt (vgl. Abbildung 2.20).

Batchsize

Die *Batchsize* entscheidet über die Anzahl an zu verarbeitenden Trainingsdaten, bevor die Modellparameter von diesem angepasst werden. Mit diesem Parameter wird die Geschwindigkeit des Trainings und die resultierende Genauigkeit des Netzes beeinflusst.

Eine große Batchsize beschleunigt das Anlernen des neuronalen Netzwerkes, benötigt aber auch mehr Speicher bei der Berechnung. Bei einer geringen Batchsize dauert das Training länger, aber das Netz kann die Daten dafür besser generalisieren, was zu einer verbesserten Leistung führt.

Batch Gradientenabstieg Der Batch Gradientenabstieg (*Batch Gradient Descent*, auch *Vanilla Gradient Descent*) berechnet den Gradienten der Verlustfunktion basierend auf dem gesamten Trainingsdatenset. Das bedeutet, dass dem Netzwerk die gesamten Daten übergeben werden, bevor eine Aktualisierung der Modellparameter durchgeführt wird. Dadurch ist diese Vorgehensweise besonders langsam und speicherintensiv. Hinzukommt, dass während das Netzwerk die Daten verarbeitet, keine weiteren Daten zum aktuellen Datensatz hinzugefügt und Berechnungen für viele ähnliche Daten redundant werden können. [29]

Stochastischer Gradientenabstieg (SGD) Hierbei werden für jedes Trainingsdatum im Datensatz die Modellparameter des Netzes angepasst. Das Vorgehen ist deswegen um einiges schneller als der Batch Gradientenabstieg und es können noch Daten zum Trainingsdatensatz hinzugefügt werden. SGD führt häufige Aktualisierungen mit einer hohen Varianz durch, was die Verlustfunktion stark schwanken lässt. Diese Schwankungen ermöglichen allerdings, dass unter Umständen lokale Minima verlassen und ein besseres Minimum gefunden werden kann. Jedoch erschwert dies auch zum exakten globalen Minimum zu konvergieren, da SGD dazu tendiert über das Ziel hinauszuwandern (*Overshooting*). [29]

Mini-Batch Gradientenabstieg Eine Kombination von SGD und Batch-Gradientenabstieg vereint die positiven Eigenschaften von beiden. Durch zufälliges Auswählen einer Anzahl von n Trainingsdaten, mittels welcher der Gradient der Kostenfunktion berechnet wird, kann die Varianz der Modellparameterupdates reduziert werden und dies führt wiederum zu einer stabileren Konvergenz. Typische Größen sind hierbei Werte zwischen 50 und 256. Oftmals werden Mini-Batch Gradientenabstiege auch als SGD bezeichnet. [29]

Trainingsepochen

Die Anzahl der Trainingsepochen entscheidet darüber, wie oft das Netzwerk das gesamte Trainingsdatenset zugeführt bekommt. Ist diese Zahl nicht hoch genug, kann das Netz nicht genügend aus den Daten lernen und erzielt am Ende keine guten Ergebnisse. Sieht es diese zu oft, kann es das Erlernte nicht genügend generalisieren und besitzt nur für das Trainingsdatenset eine gute Performance. Man spricht hierbei auch von *Overfitting*.

Versteckte Schichten

Die Auswahl, wie viele Schichten ein Netz besitzen soll, hat Einfluss auf die Komplexität der Zusammenhänge, welche es aus einem Datensatz gewinnen kann. Ab zwei oder mehr versteckten Schichten spricht man von einem *Deep Neural Network* [25]. Je tiefer ein Netzwerk ist, desto komplexer die Erkentnisse. Allerdings nimmt auch die Dauer des Trainingsvorgangs und die Gefahr des Overfittings mit der Anzahl der Schichten zu.

Anzahl der Neuronen pro Schicht

Die getätigten Aussagen über die Anzahl der Schichten treffen genauso bei der Anzahl der Neuronen pro Schicht zu. Sie besitzen wieder Einfluss auf Komplexität und Trainingsdauer.

Aktivierungsfunktionen

Ein weiterer wichtiger Hyperparameter bei der Auslegung eines neuronalen Netzes ist die Auswahl entsprechender Aktivierungsfunktionen, welche auf die Ergebnisse der Neuronen angewendet werden. Sie bestimmen den Output am Ende eines Knotenpunktes innerhalb des Netzes und können grob als eine Art Schalter aufgefasst werden, der je nach Input ein- (1) oder ausgeschaltet ist (0). Dabei ist es typisch die Aktivierungsfunktion innerhalb der verborgenen Schichten eines Netzwerkes nicht mehrfach zu wechseln, sondern durchgehend mit einer zu arbeiten. Bei der Ausgabe des Netzwerkes kann diese jedoch je nach Problemstellung angepasst werden, da für Klassifizierungs- und numerische Probleme unterschiedliche Funktionen verwendet werden. Die am häufigsten anzutreffenden Aktivierungsfunktionen im Bereich neuronaler Netzwerke sind dabei ReLU, eLU, Leaky ReLU, Sigmoid und Softmax und sollen dementsprechend vorgestellt werden.

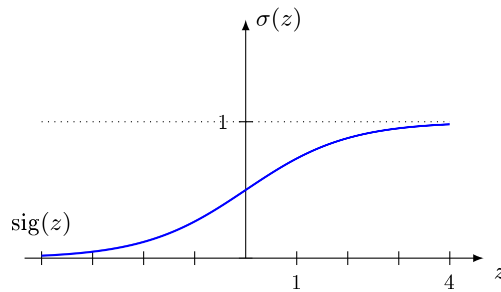


Abbildung 2.21: Kurvenverlauf der Sigmoid Aktivierungsfunktion.

Sigmoid Die Sigmoid-Funktion besitzt die Form eines „S“ und bildet eine reelle Zahl auf einen Wert, zum Beispiel den einer Wahrscheinlichkeit, zwischen 0 und 1 ab. Sie agiert dabei nach folgender Formel:

$$S(z) = \frac{1}{1 + e^{-z}} \tag{2.46}$$

Die Funktion ist besonders gut für Anwendungsbereiche geeignet, in denen eine Wahrscheinlichkeit hervorgesagt werden soll, wie bei Klassifizierungsproblemen (Beispiel: Spam-Detektion im E-mail Postfach). Der glatte Gradient der Sigmoid-Funktion verhindert abrupte Änderungen in der Ausgabe des Netzwerkes. Dies wird durch die stetige Differenzierbarkeit der Funktion an jeder Stelle gewährleistet. Diese Glätte ermöglicht außerdem stabilere und konsistentere Aktualisierungen der Modellgewichte während des Trainingsvorganges. Die Ausgabe der Funktion befindet sich hierbei in einem Bereich von [0,1], wodurch sich die Ausgaben von Neuronen normalisieren lassen. Ein großer Nachteil der Sigmoid-Aktiverungsfunktion ist, dass diese für das Problem anfällig ist, welches als „Gradientenschwinden“ bekannt ist (Siehe Abschnitt 2.2.6 auf Seite 49). Wird der Wert der Funktion zu groß oder zu klein, ist der resultierende Wert der Richtungsableitung viel kleiner als eins ($\ll 1$) und das Netzwerk gewinnt keine neuen Informationen dazu. Hinzukommt, dass die Ausgabe des Netzes nicht um null herum zentriert ist und infolgedessen die Effizienz der Gewichtsaktualisierungen reduziert. Das Berechnen der Exponentialfunktion in Gleichung 2.46 ist anspruchsvoll für Computerberechnungen und verlangsamt den Lernprozess.

ReLU (Rectified Linear Unit) Die ReLU-Funktion ist null für alle negativen Werte von x und besitzt den Wert x für alle Werte größer oder gleich null (vgl. Abbildung

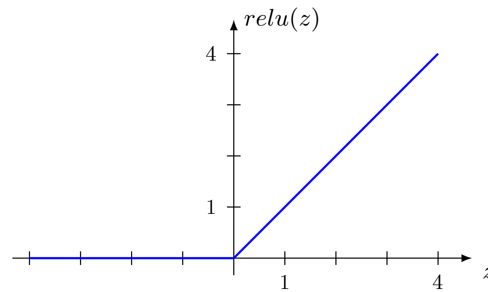


Abbildung 2.22: Kurvenverlauf der ReLU Aktivierungsfunktion.

2.22):

$$f(x) = \begin{cases} x, & x \geq 0 \\ 0, & x < 0 \end{cases} . \tag{2.47}$$

Diese Art der Aktivierungsfunktion ist im Bereich des maschinellen Lernens zurzeit aufgrund folgender Vorteile (beispielsweise gegenüber der Sigmoid-Funktion) sehr populär:

- Bei positivem Input können die Gradienten nicht in Sättigung fahren, wodurch das Netzwerk nicht mehr im Stande wäre etwas Neues zu lernen.
- Durch ihren linearen Verlauf wird eine schnelle Rechengeschwindigkeit ermöglicht.

Doch diese Funktion besitzt auch einen großen Nachteil, weshalb sie nicht für jede Problemstellung geeignet ist:

- Dying ReLU Problem: Ein Neuron dieser Funktion wird als „tot“ (*Dead*) bezeichnet, wenn es auf der negativen Seite verharrt. Dies geschieht vor allem, wenn sich der Input des Netzwerkes vermehrt im negativen Bereich befindet. Da die Steigung in diesem Bereich null beträgt, ist es unwahrscheinlich, dass es diese Seite wieder verlässt. Ursachen für das Problem können beispielsweise eine zu hohe Lernrate oder ein zu großer negativer Bias sein.

Leaky ReLU Diese Aktivierungsfunktion soll dem *Dying ReLU Problem* entgegenwirken. Dies geschieht dadurch, dass der Bereich kleiner null eine minimale Steigung besitzt (typischerweise $a = 0,01$) und nicht mehr nur den Wert null ausgibt (vgl. Abbildung 2.23):

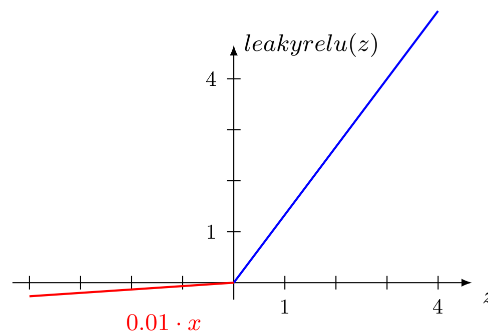


Abbildung 2.23: Kurvenverlauf der Leaky ReLU Aktivierungsfunktion mit $a = 0.05$ aus Darstellungsgründen.

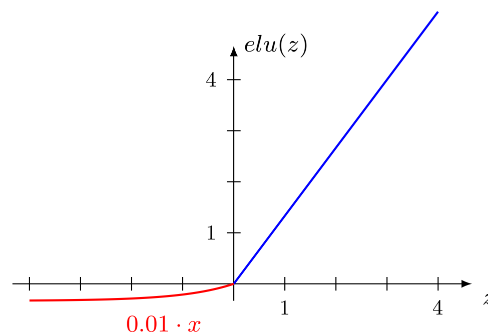


Abbildung 2.24: Kurvenverlauf der eLU Aktivierungsfunktion mit $a = 0.25$ aus Darstellungsgründen.

$$f(x) = \begin{cases} x, & x > 0 \\ a \cdot x, & x \leq 0 \end{cases} . \tag{2.48}$$

Leaky ReLU soll hierbei alle Vorteile besitzen, die auch die ReLU Aktivierungsfunktion besitzt. Es ist jedoch nicht nachgewiesen, dass Leaky ReLU tatsächlich in jedem Fall bessere Ergebnisse erzielt. Sie sollte deshalb nur als Alternative zur ReLU-Funktion betrachtet werden.

eLU (Exponential Linear Unit) Auch die eLU-Funktion soll dem Problem der ReLU Funktion entgegenwirken. Ähnlich wie bei Leaky ReLU besitzt die eLU Funktion dafür im Bereich kleiner null eine geringe Steigung. Anders als bei Leaky ReLU ist der Verlauf der Steigung jedoch exponentiell (vgl. Abbildung 2.24):

$$f(x) = \begin{cases} x, & x > 0 \\ a \cdot (e^x - 1), & x \leq 0 \end{cases} . \tag{2.49}$$

Auch diese Art der Aktivierung weist wieder alle Vorteile der ReLU Funktion auf. Sie soll durch die Verschiebung des Mittelwertes näher an null heran bessere Ergebnisse erzielen als ReLU-Funktionen (ähnlich der *Batch Normalization*[1]). Nachteilig ist allerdings, dass diese Funktion durch ihren nichtlinearen Verlauf rechenintensiver ist und dieser zusätzlich ermöglicht, dass die Gradienten für sehr große negative Eingabewerte in Sättigung fahren.

Softmax Die Softmax-Funktion findet hauptstächlich bei Klassifizierungsproblemen Anwendung. Sie bildet dafür einen Vektor mit reellen Zahlenwerten auf einen Vektor mit den entsprechenden Wahrscheinlichkeiten ab. Dies geschieht gemäß folgender Formel:

$$f(x_i) = \frac{e^{x_i}}{\sum_j e^{x_j}} \tag{2.50}$$

Der Kurvenverlauf entspricht dem einer Sigmoid Aktivierungsfunktion (vgl. Abbildung 2.21).

Optimierer

Der Optimierer (*Optimizer*) reguliert wie die Gewichte und Bias eines neuronalen Netzes nach jedem Durchlauf angepasst werden, basierend auf der gewählten Kostenfunktion und der gewählten Lernrate. Die wichtigsten Optimierer sollen hier kurz näher erläutert werden. Die zugehörigen Aktualisierungsregeln können im Artikel [29] von Sebastian Ruder nachvollzogen werden.

SGD Der stochastische Gradientenabstieg wurde in dieser Arbeit bereits in Abschnitt 2.2.2 näher beschrieben. Er ist die grundlegendste Möglichkeit Gewichte und Bias in

[1] Stabilisierungstechnik für das Training von neuronalen Netzen. Beinhaltet Neuzentrierung und Reskalierung der Netzwerkschichten [16].

einem Netz anzupassen und es lässt sich nur die Schrittweite mittels der Lernrate einstellen. Da das Verfahren Schwierigkeiten besitzt „Schluchten“, also lokale Minima, der Verlustfunktion wieder zu verlassen, ist es möglich ihm ein *Momentum* zu geben.

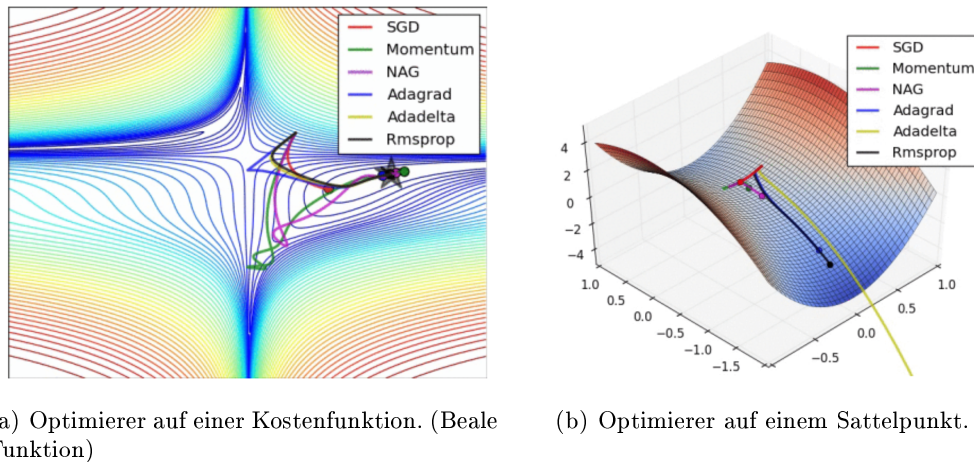
SGD mit Momentum Das Momentum hilft SGD in eine Richtung zu verstärken, indem ein Bruchteil des Aktualisierungsvektors des vorherigen Schrittes zu dem neuen Aktualisierungsvektor hinzuaddiert wird.Verglichen werden kann dies mit einem Ball, der ein Tal hinunterrollt und dabei Geschwindigkeit aufnimmt. Dadurch werden Oszillationen in der Kostenfunktion reduziert und eine beschleunigte Konvergenz erzielt. Damit das Momentum nicht unkontrolliert dem Gradienten mit gleichbleibender Geschwindigkeit folgt, sondern langsamer wird, wenn die Kostenfunktion wieder beginnt anzusteigen, kann das *Nesterov Accelerated Gradient* (kurz NAG, *beschleunigter Nesterov Gradient*) Verfahren genutzt werden. Beim klassischen Gradientenabstieg wird der Gradient berechnet und eine Bewegung in die umgekehrte Richtung ausgeführt. NAG schätzt hingegen den zukünftigen Positionsgradienten ab, um den nächsten Schritt zu modifizieren. Dadurch kann die Richtung des Gradienten, basierend auf der Geschwindigkeit des vorherigen Schrittes, korrigiert werden, sodass diese der Umgebung entsprechend angepasst wird. [29]

Adagrad Adagrad passt die Lernrate den Modellparametern des Netzwerkes an, indem vergangene Gradientenwerte berücksichtigt werden. Durch das Aufsummieren der bisherigen quadrierten Gradienten für jeden Modellparamater kann Adagrad verfolgen, wie stark sich der Gradient dort während des Trainings verändert hat. Anschließend wird die Lernrate für die Parameter bestimmt, indem die zuvor eingestellte Lenrate durch die berechnete Summierung der quadrierten Gradienten geteilt wird, wodurch die individuelle Skalierung erreicht wird. Große Gradienten erhalten hierdurch eine niedrigere Lernrate, damit diese an weiteren starken Änderungen gehindert werden. Kleine Gradienten erhalten dementsprechend eine höhere Lernrate. Dadurch erhalten Parameter mit seltenen oder großen Gradienten eine niedrigere Lernrate, während Parameter mit häufigen oder kleinen Gradienten eine höhere Lernrate erhalten. Aufgrunddessen ist Adagrad gut geeignet für spärliche Datensätze und eleminiert manuelles Tuning der Lernrate. Der große Nachteil des Optimierers ist, dass dieser die Lernrate immer weiter verringert, bis diese infinitesimal klein ist und keine Aktualisierungen mehr stattfinden, wodurch der Lernprozess schlussendlich verhindert wird. [29]

Adadelta Adadelta soll dem Nachteil von Adagrad entgegenwirken, indem nicht mehr alle bisherigen Gradienten in die Aufsummierung pro Modellparameter miteinbezogen werden, sondern nur noch eine bestimmte Anzahl. Dadurch passt sich die Lernrate schneller Änderungen in der Kostenfunktion an und wird weniger stark von bisherigen Gradienten beeinflusst. Theoretisch muss nicht einmal eine Lernrate zu Beginn eingestellt werden, da Adadelta diese bei den Berechnungen nicht berücksichtigt. [29]

RMSProp RMSProp wurde zur selben Zeit und aus demselben Grund entwickelt wie Adadelta und soll das Problem von Adagrad beheben. Auch RMSProp ist eine adaptive Methode zur Anpassung der Lernrate während des Trainings, auf welche die Idee von Adadelta aufbaut. RMSProp nutzt genau wie Adadelta eine kumulierte Quadratsumme der Gradienten, die mit einem abnehmenden Durchschnitt gewichtet wird. Dadurch werden allerdings nur vergangene Gradienten bei der Aktualisierung miteinbezogen. Adadelta berücksichtigt nicht nur diese, sondern auch die bereits getätigten Aktualisierungsschritte, wodurch eine noch genauere Anpassung ermöglicht wird. Da dies jedoch mit erhöhtem Rechenaufwand verbunden ist, ist die Nutzung von RMSProp noch immer weit verbreitet.

Adam Adam (kurz für *Adaptive Moment Estimation*, *Adaptive Momentschätzung*) ist eine weitere Optimierungsmethode zur adaptiven Anpassung der Lernrate eines jeden Modellparameters. Die Methode vereint Ideen der momentumbasierten und adaptiven Lernratenoptimierung. Dafür werden zwei Momenta für jeden Modellparameter abgeschätzt und die Lernrate diesen entsprechend angepasst. Des Weiteren kommt eine Bias-Korrektur zum Einsatz, welche die Genauigkeit der Schätzungen erhöhen soll. In Abbildung 2.25 sind die Verläufe der vorgestellen Optimierer dargestellt, welche diese auf der Kontur einer Verlustfunktion (Beale Funktion) gewählt haben. Es ist zu erkennen, dass Adagrad, Adadelta und RMSProp direkt zum Minimum konvergieren, während NAG und Momentum zunächst aus der Spur geführt wurden. Abbildung 2.25 zeigt das Verhalten derselben Optimierer auf einem Sattelpunkt. SGD, Momentum und NAG besitzen deutliche Schwierigkeiten diesen zu verlassen, im Gegensatz zu Adagrad, Adadelta und RMSProp. [29]



(a) Optimierer auf einer Kostenfunktion. (Beale Funktion)

(b) Optimierer auf einem Sattelpunkt.

Abbildung 2.25: Optimierer im Vergleich [29].

Verlustfunktion

Die Verlustfunktion (*Loss Function*), auch bekannt als Kostenfunktion (*Cost Function*) oder Fehlerfunktion (*Error Function*), quantifiziert die Diskrepanz zwischen den ausgegebenen Werten des Modells und den tatsächlichen Zielwerten. Durch sie wird die Modellqualität bewertet und dient als Leitfaden bei der Optimierung der Modellparameter während des Trainings (beispielsweise durch das Gradientenabstiegsverfahren). Ziel beim Training ist die Minimierung des berechneten Verlustes, um die Vorhersagegenauigkeit des neuronalen Netzwerkes zu verbessern. Die Kostenfunktion ist ein entscheidender Bestandteil des Modelltrainings und deshalb ist es umso wichtiger eine passende Funktion für den jeweiligen Aufgabenbereich auszuwählen. Zu den typischen Problemen im maschinellen Lernen gehören Klassifikation und Regression. Bei Klassifikationsproblemen werden Daten in vordefinierte Klassen oder Kategorien eingeteilt und bei Regressionsproblemen wird ein numerischer Wert hervorgesagt. Aufgrunddessen sind für diese Probleme unterschiedliche Verlustfunktionen einzusetzen. Im nachfolgenden sollen kurz häufig anzutreffende Kostenfunktionen aufgeführt und erläutert werden.

Mittlere quadratische Abweichung Die mittlere quadratische Abweichung (*Mean Squared Error*, MSE) ist eine typische Fehlerfunktion bei Regressionsproblemen. Große Fehler werden hierbei höher gewichtet und besitzen mehr Einfluss auf die Optimierung des Modells. Werden die tatsächlichen Werte mit y und die vorhergesagten Werte mit $\hat{y}$ bezeichnet, ergibt sich für den MSE die Formel 2.51 [9].

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2 \tag{2.51}$$

Die Anzahl der berücksichtigten Datenpunkte wird hier mittels n gekennzeichnet.

Mittlere absolute Abweichung Die mittlere absolute Abweichung (*Mean absolute Error*, MAE) ist weniger anfällig gegenüber Ausreißern, da sie diese genauso stark gewichtet wie die anderen Fehler. Auch sie findet bei Regressionsproblemen Verwendung. Die zugehörige Gleichung zum MAE ist Formel 2.52 zu entnehmen [9].

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{m}|(y_i - \hat{y}_i)| \tag{2.52}$$

Kreuzentropie Die Kreuzentropie (*Cross-Entropy*) ist eine häufig verwendete Verlustfunktion bei Klassifikationsproblemen. Je größer der berechnete Wert ist, desto größer ist die Abweichung zwischen der vorhergesagten und der tatsächlichen Klasse. Sollen Vorhersagen für mehrere Klassen gemacht werden, spricht man von der kategorischen Kreuzentropie (*Categorical Cross Entropy*). Sind nur zwei Klassen vorhanden, wird die binäre Kreuzentropie (*Binary Cross Entropy*) genutzt. Formel 2.53 wird für die Berechnung der Kreuzentropie angewandt [9]. Das negative Vorzeichen in der Gleichung resultiert aus der Anwendung des Logarithmus.

$$\text{CE} = -\frac{1}{n}\sum_{i=1}^{n}\sum_{j=1}^{m} y_{ij} \cdot \log(\hat{y}_{ij}) \tag{2.53}$$

Die Anzahl der Datenpunkte wird wieder über n und die Anzahl der Klassen über m gekennzeichnet. Ist nur zwischen zwei Klassen zu unterscheiden, vereinfacht sich die Formel wie folgt [9]:

$$\text{CE} = -\frac{1}{n}\sum_{i=1}^{n}[y_i \cdot \log(\hat{y}_i) + (1 - y_i) \cdot \log(1 - \hat{y}_i)] \tag{2.54}$$

2.2.6 Probleme beim Trainieren eines neuronalen Netzes

Beim Training eines neuronalen Netzes kann es passieren, dass die Qualität der Netzwerkausgaben plötzlich abnimmt, der Lernprozess stagniert und möglicherweise gar nicht konvergiert oder die Verlustfunktion sogar wieder stark zu steigen beginnt und divergiert. Dies hat meist mit den Berechnungen während der Fehlerrückführung und den zugehörigen Gradienten zu tun, welche während des Trainings sehr groß oder sehr klein geworden sind. Hierbei spricht man von dem Exploding beziehunsgweise Vanishing Gradient Problem (*Explodierender* bzw. *verschwindender Gradient*), welche hier kurz näher erläutert werden sollen.

Schwindende Gradienten

Sind die Modellparameter eines Netzes (vor allem die Gewichte) vorwiegend kleiner als eins, was bei Gewichtsinitialisierungen nach Gauß beispielsweise der Fall ist, werden die Gradienten bei der Backpropagation, bis die Eingangsschicht erreicht wird, immer kleiner. Durch den Aufbau eines neuronalen Netzes ist es somit typisch, dass die letzten Schichten schneller lernen als die ersten. Besitzt das Netz viele Schichten, kann dies den Lernprozess erschweren. Denn bei der Fehlerrückführung ist es bei vielschichtigen Netzwerken durchaus möglich, dass die Fehlergradienten beim Erreichen des Input-Layers so klein geworden sind, dass dort kaum Veränderungen beim Aktualisieren der Gewichte verursacht werden. Man spricht hierbei auch von dem schwindenden Gradientenproblem (*Vanishing Gradient Problem*). Dies erschwert die Findung der richtigen Richtung zur Reduktion der Kostenfunktion, weshalb diese beispielsweise stagniert. [15]

Es lässt sich anhand folgender Merkmale feststellen, ob ein neuronales Netz unter dem schwindenden Gradienten Problem leidet:

- Das Modell lernt nur langsam, gar nicht oder stagniert zu einem frühen Zeitpunkt.
- Die Gewichte des Modells sind sehr klein oder gar null.
- Parameter der höheren Schichten ändern sich stark und die der tieferen nur wenig bis gar nicht.

Mittels folgender Möglichkeiten kann bei einem *Vanishing Gradient*-Problem Abhilfe geschaffen werden:

- Batch Normalization: Ermöglicht, dass sich die Ausgabe der Neuronen in einem festen Bereich ([0,1]) befinden.
- ReLU-Aktivierungsfunktion: Sättigt nur in eine Richtung und ist somit wiederstandsfähiger.
- Residual Network Architecture: Durch *Skip-Connections* verliert das Signal nicht an Stärke mit zunehmender Layer-Anzahl, da die höheren Layer mit den tieferen stärker verbunden sind.

Explodierende Gradienten

Das Exploding Gradient Problem wird durch sehr große Fehlergradienten verursacht, welche sich während der Trainingsphase ansammeln und zu entsprechend großen Aktualisierungen der Gewichte führen [15]. Dadurch wird das Modell unfähig nützliche Informationen vom Ende des Netzes zum Anfang zurückzuführen. Dabei gibt ein Fehlergradient Aufschluss über die Richtung und Größe der Gewichtsaktualisierungen, um zum globalen Minimum zu konvergieren. In tieferen Netzwerken kommt es jedoch vor, dass sich diese Fehlergradienten aufsummieren und zu (zu) großen Gradienten und aus diesen folgenden Aktualisierungen führen. Dadurch kann das Lernverhalten instabil werden und die Gewichte numerisch überlaufen. Dies wird häufig über *NaN* gekennzeichnet, was für *Not a Number* steht und bedeutet, dass ein numerischer Wert nicht mehr korrekt berechnet werden konnte. Werden wiederholt Gradienten > 1 miteinander multipliziert, entsteht ein exponentielles Wachstum und die Gewichte „explodieren“, woher auch der Name stammt. Somit ist das Netz nicht mehr in der Lage sinnvolle Informationen herauszufiltern und es findet kein Lernprozess statt, wie in Abbildung 8.4 aus der Sektion über die Batchsize zu erkennen ist.

Es lässt sich anhand folgender Merkmale feststellen, ob ein neuronales Netz unter dem explodierendem Gradienten Problem leidet:

- Das Modell ist nicht in der Lage zu lernen. Dies ist an dem Verlauf des Verlustes erkennbar (stagniert beisielsweise).
- Das Modell ist sehr instabil. Der Verlust unterliegt sehr großen Schwankungen von Aktualisierung zu Aktualisierung.
- Der Verlust kann nicht mehr korrekt ermittelt werden (NaN).

Folgende Details sind etwas weniger offensichtlich, aber auch eindeutige Anzeichen dieses Problems:

- Gewichte des Netzes werden sehr schnell sehr groß.
- Gewichte könnten nicht mehr richtig berechnet werden (NaN).
- Fehlergradienten sind konstistent > 1.

Mittels folgender Möglichkeiten kann bei einem *Exploding Gradient* Problem Abhilfe geschaffen werden:

- Neuentwurf des neuronalen Netzes: Es kann von Vorteil sein, weniger Schichten zu nutzen oder auch eine geringere Batchsize zu verwenden.
- Gradient Clipping: Mit Hilfe von Gradient Clipping ist es möglich einen Threshold festzulegen, damit Gradienten diesen nicht übersteigen.
- Weight Regularization: Hinzufügen einer Bestrafungsregelung für die Verlustfunktion des Netzwerkes bei zu groß werdenden Gewichten.

Instabile Gradienten Problem

Der Gradient einer früheren Schicht innerhalb eines Netzes ist ein Produkt der Gradienten seiner nachfolgenden Schichten. Dies führt zu einer instabilen Situation, da alle Schichten nur mit gleicher Geschwindigkeit lernen können, wenn sich die Terme innerhalb des Produktes ausgleichen. Da dies ohne Gegenmaßnahme eher unwahrscheinlich ist, führt dies zu einem instabilen Verhalten, wodurch die Schichten mit unterschiedlichen Geschwindigkeiten lernen. [25]

Lösungsansätze

Eine große Hilfe bei dem Debugging des neuronalen Netzes und den zu tunenden Hyperparametern ist die browserbasierte Anwendung „TensorboardX“[1]. Dies ist ein Profiling-Tool, welches speziell für Deep Learning entwickelt wurde. Es ermöglicht eine Live-Überwachung der Trainingsparameter (Gewichte und Biases) und Metriken (Verluste)

[1] https://tensorboardx.readthedocs.io/en/latest/tutorial.html
- Zugriffsdatum: 07.07.23

mittels automatisierter Visualisierungsdiagramme [27]. Besonders nützlich zum Feinjustieren der Hyperparameter eignet sich die Verlaufsfunktion des Tools, womit beobachtet werden kann, welche Parameter den besten Einfluss besitzen. Durch Anwendungen wie TensorboardX lassen sich die Modellparameter genauer analysieren, da das Tool Anhaltspunkte gibt, an welchen Stellen des Trainings Schwierigkeiten auftreten. Auch die Gradienten lassen sich damit verfolgen, was die Erkennung des Vanishing/Exploding Gradient Problems und die Umsetzung entsprechender Maßnahmen erleichtert.

2.2.7 Arten von neuronalen Netzen

Es gibt unterschiedliche Arten von neuronalen Netzen mit verschiedenen architektonischen Aufbauten. In diesem Abschnitt sollen die drei am häufigsten verwendeten vorgstellt werden.

Feedforward Neural Network

Das Feedforward Netzwerk (kurz FFN) ist die grundlegendste und einfachste Art für eine Netzwerkstruktur, deren Aufbau und Funktionsweise in den vorerhigen Abschnitten bereits ausreichend erläutert wurde.

Convolutional Neural Network

Das Convolutional Neural Network (kurz CNN) ist augrund seiner Struktur besonders gut für Aufgaben im Bereich der Bild- und Videoanalyse geeignet. Dies liegt darin begründet, dass dieses im Gegensatz zum FFN keine vollständig verbundenen Schichten nutzt, also nicht jedes Neuron einer Schicht mit jedem Neuron der nächsten Schicht in Verbindung steht. Dies besitzt außerdem den Vorteil, dass sich solche Netze schneller trainieren lassen und eine tiefere Netzwerkstruktur ermöglichen. CNN stützt sich hierbei auf drei Grundideen: lokale rezeptive Felder, geteilte Gewichtsparameter und Pooling. [25]

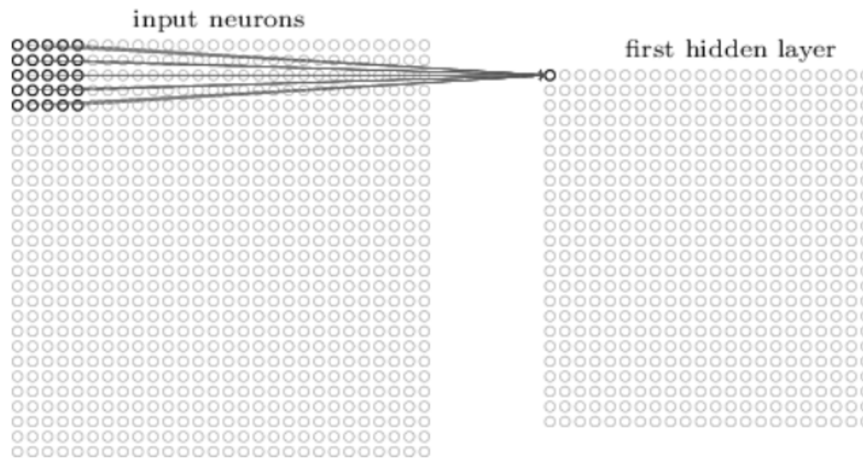


Abbildung 2.26: Verbinden eines Feldes des Bildes mit dem ersten Eingangsneuron [25].

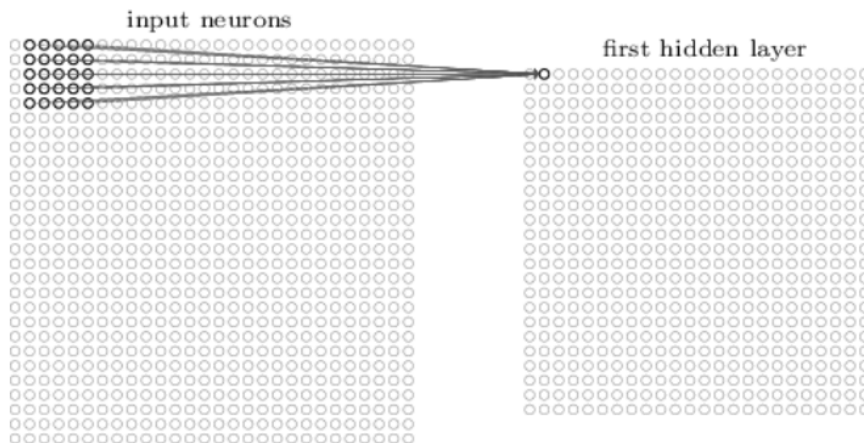


Abbildung 2.27: Verbinden eines Feldes des Bildes mit dem zweiten Eingangsneuron [25].

Lokale rezeptive Felder Nielsen erklärt lokale rezeptive Felder anhand eines Beispiels mit einem Bild als Netzwerkinput. Statt dass jedes Pixel eines Bildes mit einem Eingangsneuron verbunden wird, erhält jedes Neuron einen kleinen Bereich als Eingabe: ein lokales rezeptives Feld (vgl. Abbildung 2.26)

Jedem möglichen Feld eines Bildes wird ein Eingangsneuron zugewiesen, welches nur diesen Abschnitt analysiert (vgl. Abbildung 2.27).

Geteilte Gewichte Jedes Neuron des CNN besitzt einen Bias und so viele Gewichte, wie es an Verbindungen zu dem lokalen Feld besitzt (bei einem 5x5 Feld wären dies 25). Dabei teilen sich die Neuronen einer versteckten Schicht dieselben Bias und Gewichte (*Shared Weights and Bias*). Für das (j,k)-te Neuron ergibt sich die Ausgabe

$$\sigma\left(b + \sum_{l=0}^{n} \sum_{m=0}^{n} w_{l,m} a_{j+l,k+m}\right). \tag{2.55}$$

mit:

σ – Aktivierungsfunktion beispielsweise Sigmoidfunktion

b – Geteilter Biaswert

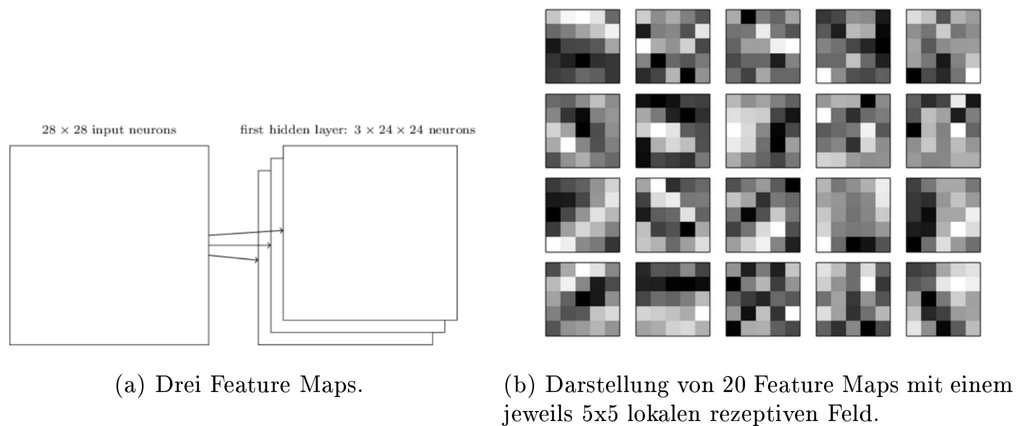
$w_{l,m}$ – Array der geteilten Gewichte mit Größe n

$a_{j+l,k+m}$ – Aktivierung an Position j+l,k+m

In den Schichten eines CNNs wird Kreuzkorrelation (vgl. Gleichung 2.55) verwendet, um Informationen aus den Eingabedaten zu erfassen. Dennoch hat sich umgangssprachlich der Begriff der Faltung durchgesetzt, da die beiden mathematischen Operationen sich in Eigenschaften sowie Ergebnissen ähneln. Jedoch unterscheiden sich Faltung und Kreuzkorrelation in ihrer Definition. Durch die Verwendung von geteilten Modellparametern detektieren die Neuronen alle dieselbe Charakteristik an unterschiedlichen Stellen des Bildes [25]. Aufgrund dessen nennt man die Verbindungen zwischen Eingangsneuronen und der ersten verstecken Schicht auch *Feature Map* (*Merkmal Abbild*) und die geteilten Gewichte sowie Bias auch Kernel oder Filter. Damit von dem Netzwerk nicht nur ein einziges Merkmal erkannt werden kann, benötigt dieses mehrere Feature Maps (vgl. Abbildung 2.28).

Ein großer Vorteil der geteilten Parameter ist die geringe Anzahl, die benötigt wird im Gegensatz zu beispielsweise einem Feedforward Netzwerk und der daraus resultierenden kürzeren Trainingszeit [25]. Sind die verwendeten Eingangsdaten jedoch sehr groß (beispielsweise Bilddaten), wird auch mehr Speicherplatz benötigt, wodurch das Training wieder verlangsamt wird.

Pooling Eine weitere Schicht, welche überlichweise in CNNs zum Einsatz kommt, ist die Pooling-Schicht. Diese soll die Informationen, welche in der Faltungsschicht ermittelt wurden, komprimieren, bevor diese weiterverwendet werden. Dafür werden Regionen der



(a) Drei Feature Maps.

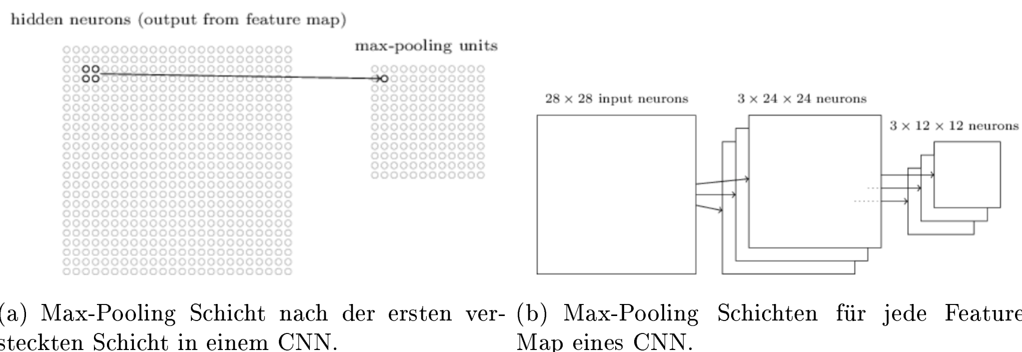
(b) Darstellung von 20 Feature Maps mit einem jeweils 5x5 lokalen rezeptiven Feld.

Abbildung 2.28: Feature Maps in einem Convolutional Neural Network für die Erkennung mehrerer Merkmale [25].

Größe $n \times n$ betrachtet. Im Max-Pooling ist beispielsweise nur der größte in der Region enthaltene Wert relevant und wird als eine Ausgabe der Pooling Schicht verwendet. In Abbildung 2.29 ist das Verfahren des Max-Pooling noch einmal dargestellt.

Mittels Max-Pooling lässt sich ein Merkmal in einer bestimmten Region des Bildes detektieren, von welchem die genaue Position zunächst irrelevant ist [25]. Max-Pooling ist nicht die einzige Pooling-Variante, es gibt noch eine Vielzahl weiterer Möglichkeiten die Informationen zu komprimieren.

Ein beispielhaftes CNN mitsamt der beschriebenen Komponenten ist der Abbildung 2.30 zu entnehmen. Das CNN wurde hierbei um eine vollständig verbundene Schicht ergänzt, welche üblicherweise genutzt wird, um die vorherigen Komponenten des Netzes mit der



(a) Max-Pooling Schicht nach der ersten versteckten Schicht in einem CNN.

(b) Max-Pooling Schichten für jede Feature Map eines CNN.

Abbildung 2.29: Max-Pooling in einem Convolutional Neural Network [25].

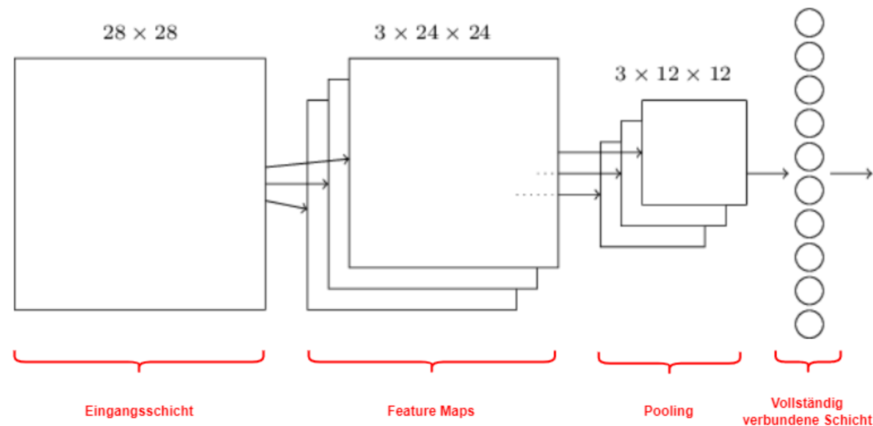


Abbildung 2.30: Beispielhaftes Convolutional Neural Network [25, modifiziert].

Ausgangsschicht zu verbinden. Die Gleichungen aus dem Abschnitt *Backpropagation* sind für dieses Netzwerk nicht ohne Modifikationen nutzbar, da diese für vollständig verbundene Schichten ausgelegt sind, welche das CNN nicht durchweg besitzt.

Recurrent Neural Network

Anders als beim Feedforward Netz wird bei einem rekurrenten (*Recurrent*) Netzwerk (kurz RNN) die Ausgabe eines Neurons nicht nur durch die Aktivierungen der vorherigen Schicht bestimmt, sondern möglicherweise durch vergangene Aktivierungen des Neuronen selbst. Das liegt darin begründet, dass es im Gegensatz zum FFN eine dynamische Architektur besitzt, bei der die Ausgänge eines Neurons entweder als Eingang für dieselbe Schicht oder für eine vorherige Schicht verwendet werden können. Dies soll die Art und Weise wie das menschliche Hirn funktioniert noch besser nachahmen, da auch dort nicht alles nur von einem Eingang zu einem Ausgang weitergegeben wird. Dadurch ist das RNN besonders gut geeignet für Aufgaben, die das Analysieren von Daten oder Prozessen, welche sich mit der Zeit ändern, beinhalten. Beispiele für solche Aufgabenfelder sind die Spracherkennung sowie -analyse und das Nachvollziehen von Python Programmcode [25]. Dabei inkorporiert das RNN eine Menge des Standard FFNs und ein Großteil der Konzepte lässt sich auf dieses übertragen. Durch den dynamischen Aufbau ist es allerdings schwerer zu trainieren und anfällig für Probleme (vgl. Abschnitt 2.2.6), welche während des Trainings auftreten können. Zur Verbesserung des instabilen Gradientenproblems wurden deshalb *Long short-term memory Einheiten* (kurz LSTM, *langes*

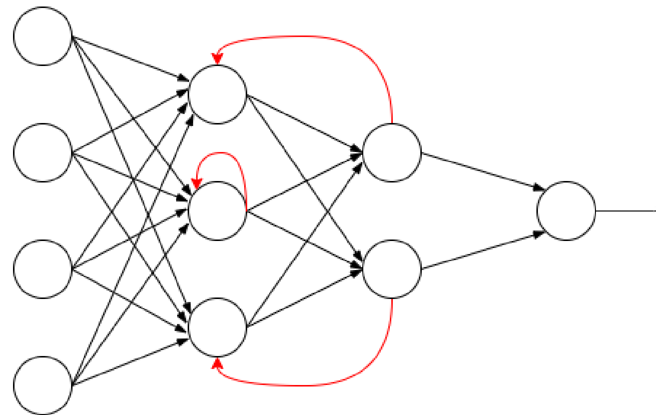


Abbildung 2.31: Beispielhaftes Recurrent Neural Network mit den rekurrenten Verbindungen in rot [25, modifiziert].

Kurzzeitgedächtnis) entwickelt, die dieses torpedieren sollen. In Abbildung 2.31 ist die Architektur eines rekurrenten neuronalen Netzwerkes beispielhaft dargestellt.

Residual Neural Network

Neuronale Netze mit tieferen Netzwerksturukturen sind schwieriger zu trainieren, da diese anfälliger sind für die in Abschnitt 2.2.6 angesprochenen Probleme. Mit steigender Anzahl von Schichten, steigt auch die Wahrscheinlichkeit, dass das Exploding/Vanishing Gradient Problem auftritt. Um diesen entgegenzuwirken stehen einige Techniken zu Verfügung, wie beispielsweise verbesserte Gewichtsinitialisierung oder zusätzliche Normalisierungsschichten [12]. Durch diese ist es tiefen Netzwerken möglich zu einem Minimum der Kostenfunktion zu konvergieren, sie weisen allerdings dafür ein neues Problem auf. Die Genauigkeit der Netzwerke beginnt ab einer gewissen Anzahl von Schichten zu stagnieren und verschlechtert sich im weiteren Verlauf sogar. Kaiming He et al. stellen in ihrem Artikel [12] eine Möglichkeit vor, um diesem Problem habhaft zu werden: das *Deep Residual Network*. Dort werden sogenannte Skip-Connections (*Verbindungssprünge*) genutzt, welche das Training von tieferen Netzen verbessern solleen. Anstelle, dass mehrere Schichten übereinandergestapelt werden wie beim FFN, was die Gradienten der Verlustfunktion in Bezug auf die Parameter der unteren Schichten sehr klein werden lässt, werden zusätzliche Skip-Connections implementiert, welche eine oder mehrere Schichten überspringen, um den Gradienten von früheren Ebenen zu späteren Ebenen ohne Informationsverluste des Inputs zu übertragen (vgl. Abbildung 2.32). Ein Residual Neural

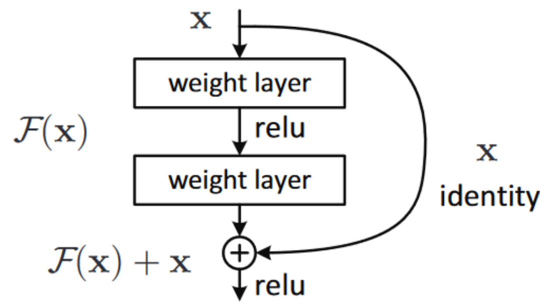


Abbildung 2.32: Ein Residual Block mit zugehöriger Skip-Connection [12].

Network (kurz ResNet) besteht hierbei aus mehreren Residual Blocks, welche einen Aufbau wie in Abbildung 2.32 besitzen können. Dadurch ist es dem ResNet im Vergleich zu einfacheren Netzwerktypen, wie zum Beispiel dem FFN, möglich, wesentlich tiefere Strukturen zu erreichen.

Anstatt direkt eine grundlegende Zuordung (*Underlying Mapping*) $\mathcal{H}(x)$ für die Netzwerkparameter (des gesamten Netzes oder nur eines Teiles) zu erlernen, wird hier eine Residual-Funktion $\mathcal{F}(x)$ ermittelt, welche den Unterschied zwischen Ein- (x) und Ausgabe eines Residual Blocks ($\mathcal{F}(x)$) ausdrückt:

$$\mathcal{F}(x) := \mathcal{H}(x) - x \tag{2.56}$$

2.3 Deep Reinforcement Learning

Deep Reinforcement Learning ermöglicht die Anwendung des Reinforcement Learnings in komplexeren Umgebungen durch die Verwendung von neuronalen Netzen. Umgebungen mit einem sehr großen oder sogar unendlichen Zustandsraum sind nicht mehr mit den Standardmethoden des Reinforcement Learnings zu bewältigen, da einem Agenten nicht mehr möglich ist eine optimale Strategie für jedes Zustand-Aktions-Paar zu erlernen. Ein Beispiel dafür ist der Zauberwürfel, welcher $4,33 \cdot 10^{19}$ Zustände besitzt und es einem RL Agenten unmöglich macht eine Policy zu ermitteln. Ein häufig eingesetztes Verfahren ist das Deep-Q-Learning. Hierbei wird ein neuronales Netz eingesetzt, um die Q-Funktion zu approximieren, welche den zu erwartenden Gewinn für jede durchführbare Aktion in einem Zustand abschätzt. Dafür wird dem Netzwerk der aktuelle Zustand als Eingabe übergeben, welcher wiederum die seiner Berechnungen nach beste Aktion ausgibt. Das

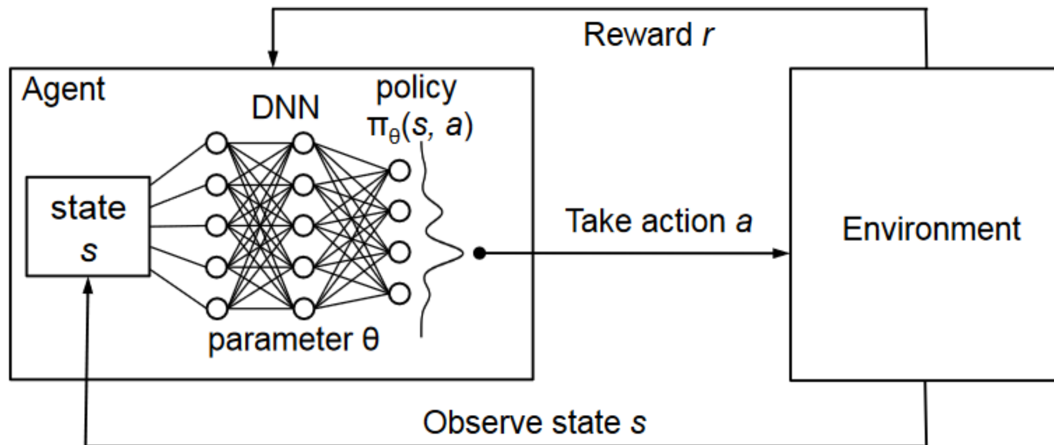


Abbildung 2.33: Reinforcement Learning in Kombination mit einem neuronalen Netzwerk [20].

Netz wird klassisch mittels stochastischem Gradientenabstieg trainiert, wobei die Kostenfunktion den Unterschied zwischen vorhergesagtem und dem tatsächlichen Q-Wert berechnet, welcher mittels Bellmann-Gleichung (2.12 auf Seite 11) ermittelt wird. Der Aufbau eines Deep-Q-Netzwerkes ist Abbildung 2.33 zu entnehmen.

3 Grundlagen: Mathematik

Um den Grundlagenteil abzuschließen, soll die Mathematik, auf welcher der Rubiks's Cube und seine Algorithmen beruhen, aufgegriffen werden, um Ideen für andere Lösungen anzuregen und zusätzlich ein Verständnis für das Enstehen bekannter Lösungsalgorithmen zu schaffen.

3.1 Zauberwüfel

In diesem Abschnitt soll der Zauberwürfel, welcher von Ernő Rubik 1974 erfunden wurde, näher erläutert und die verwendete Terminologie und Notation erklärt werden. Der nachfolgende Text bezieht sich dabei auf das von David Singmaster und Alexander Frey Jr. publizierte Buch „Handbook of Cubik Math“ [30], in welchem nicht nur die Bergifflichkeiten, sondern auch Wege zur Lösungsfindung, die auf dem mathematischen Prinzip der Gruppentheorie beruhen, erläutert werden. Da in dieser Arbeit jedoch eine künstliche Intelligenz selbst Wege zum Lösen eines Zauberwürfels finden soll, wird nicht im Detail auf die Mathematik hinter dem Würfel eingegangen und es werden lediglich mögliche Ansätze diskutiert.

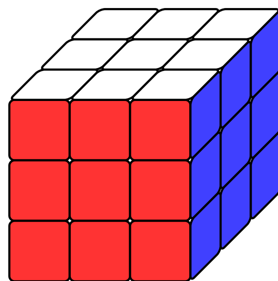
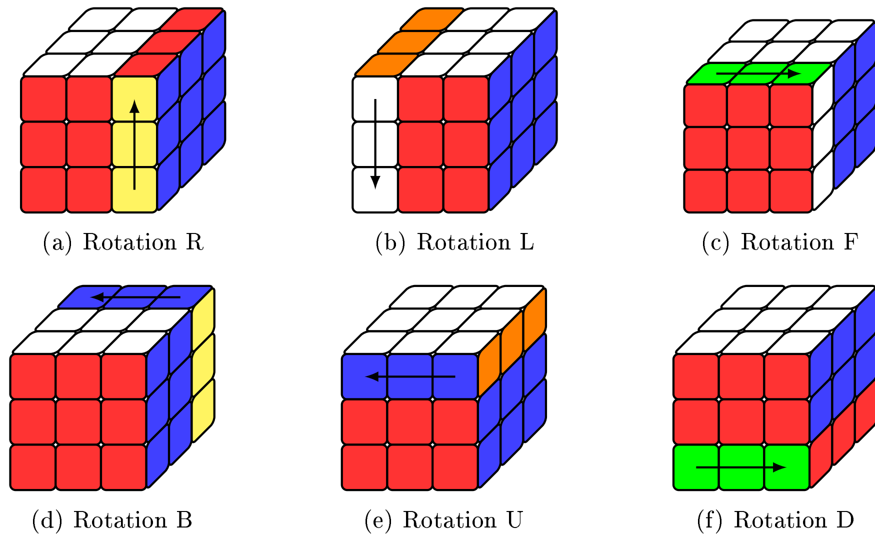


Abbildung 3.1: Darstellung des Rubik's Cube im gelösten Zustand.



(a) Rotation R (b) Rotation L (c) Rotation F

(d) Rotation B (e) Rotation U (f) Rotation D

Abbildung 3.2: Darstellung der Würfeltransformationen nach David Singmasters Notation [30]. Eine Rotation erfolgt hierbei um 90° im Uhrzeigersinn bei Blick auf die rotierte Seite. Die jeweilige inverse Rotation ist entegegen der eingezeichneten Richtung.

3.1.1 Terminologie und Notation

Der Zauberwürfel besitzt sechs unterschiedlich farbige Seiten und ist in jede Richtung in drei Teile aufgeteilt, sodass er den Aufbau eines 3x3x3-Arrays besitzt. Dieses ist mit kleineren Würfeln gefüllt, die *Cubies* genannt werden (vgl. Abbildung 3.1). Davon besitzt dieser eigentlich 27 Stück (3x3x3), wobei der Cubie in der Mitte nur imaginär ist und nicht mitgezählt wird, womit man auf eine Zahl von 26 kommt. Jede Seite des Würfels besteht aus 3x3 farbigen Flächen der Cubies, den sogenannten *Facelets*, wovon der Würfel insgesamt 6x9 = 54 Stück besitzt. Der Würfel gilt als gelöst, wenn alle Seiten aus jeweils einer Farbe bestehen (siehe Abbildung 3.1). Alle Seiten des Würfels lassen sich im oder gegen den Uhrzeigersinn rotieren, wodurch die Cubies und die dazugehörigen Facelets bewegt werden können (vgl. Abbildung 3.2).

Der Zauberwürfel besitzt drei unterschiedliche Arten von Cubies:

- Ecksteine (*Corner Piece*): besitzen drei sichtbare Facelets.
- Kantensteine (*Edge Piece*): besitzen zwei sichtbare Facelets. Sie sitzen zwischen zwei Ecksteinen.

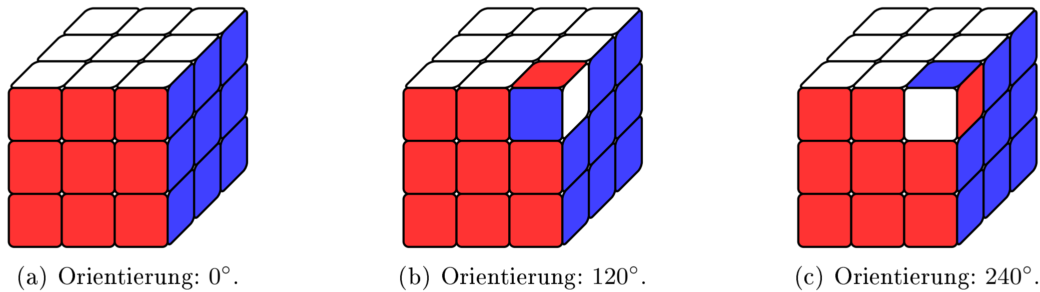

(a) Orientierung: 0°. (b) Orientierung: 120°. (c) Orientierung: 240°.

Abbildung 3.3: Orientierungunsmöglichkeiten eines Ecksteins.

- Mittelsteine (*Center Piece*): besitzen ein sichtbares Facelet. Sie sitzen in der Mitte einer Seite und bestimmen die Farbe der Würfelseite, auf der sie sich befinden.

Durch das Drehen der Seiten werden die Würfelsteine von einer Position zu der Position eines anderen Würfelsteins bewegt. Die Orte auf dem Würfel werden auch *Cubicles* genannt. Dabei bewegen sich Ecktseine immer nur zu einem Cubicle eines anderen Ecksteines, Kantensteine nur zu Cubicles von Kantensteinen und Mittelsteine können gar nicht bewegt und nur um sich selbst rotiert werden. Die Lage der Mittelsteine zueinander verändert sich hierbei nicht. Werden zwei Farben auf den Positionen des Nord- und Südpols platziert, ist die Reihenfolge der restlichen vier Farben immer gleich. Dadurch, dass die Mittelsteine die Farbe einer Seite vorgeben, lassen sich auch die Cubicles bestimmen, an welche die Cubies gebracht werden müssen, damit der Würfel als gelöst gilt. Ist ein Kantenstein zum Beispiel orange und grün, so muss dieser zwischen dem orangenen und grünen Mittelstein positioniert werden, sodass sein orangenes Facelet am orangenen Facelet des Mittelsteines liegt. Gleiches gilt für Ecksteine. Auch hier muss die Orientierung seiner Facelets zu den Farben der entsprechenden Seiten passen. Der Standort eines jeden Cubies, an welchem diese Vorraussetzung erfüllt ist, wird auch Heimstandort (*Home Location*) genannt. Es ist möglich, dass ein Cubie sich in seiner Home Location befindet, seine Farben sich jedoch auf den falschen Seiten befinden. Bei Ecksteinen spricht man davon, dass diese verdreht (*Twisted*, vgl. Abbildung 3.3) und bei Kantensteinen davon, dass diese umgedreht (*Flipped*, vgl. Abbildung 3.4) sind. Somit besitzt jeder Stein einen eigenen Heimstandort und eine zugehörige Orientierung, in welcher dieser gebracht werden muss.

Laut Singmaster ist der wichtigste Aspekt in der Terminologie und Notation, dass diese von allen Menschen, die über das Thema kommunizieren, in gleicher Weise genutzt und akzeptiert wird. Die am meisten verarbeitete und akzeptierte Terminologie und Notation

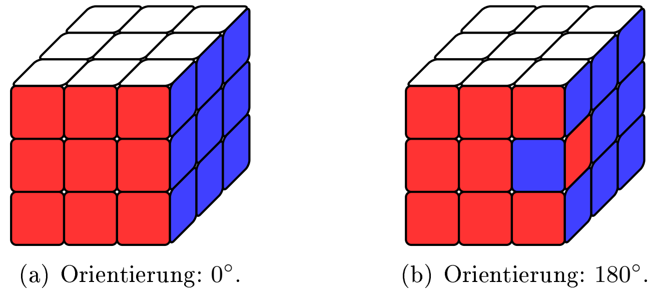


Abbildung 3.4: Orientierungunsmöglichkeiten eines Kantensteins.

ist dabei die von David Singmaster selbst entwickelte, auf welche sich auch diese Arbeit stützen wird. Dieser verwendet zur Beschreibung der Würfelseiten folgende Begrifflichkeiten (vgl. Abbildung 3.5):

- Front: F
- Back: B
- Right: R
- Left: L
- Up: U
- Down: D

Es werden die Positionen der Seiten relativ zur Person, welche den Wüfel hält, genutzt, da unterschiedliche Zauberwürfel auch unterschiedliche Farben besitzen können und es

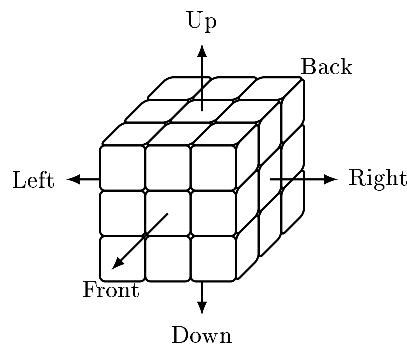


Abbildung 3.5: Notation der Würfelseiten.

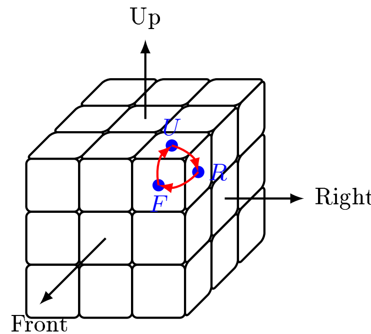


Abbildung 3.6: Reihenfolge bei der Beschreibung eines Würfelsteins anhand des Beispiels des URF-Cubies.

somit ansonsten zu Verwirrungen kommen könnte. Es ist hierbei Festlegungssache, welche Farbe sich an welcher Position befindet. In dieser Arbeit soll die weiße Farbe oben sein und die rote Seite zum Betrachter zeigen (vgl. Abbildung 3.1). Die Farbwahl für die jeweiligen Seiten definiert zugleich die Orientierung des Würfels. Wird beispielsweise der ganze Würfel gedreht, und die obere Seite befindet sich nun unten, ändert das nicht die Orientierung des Würfels, und die Farbe, welche der oberen Seite zugewiesen wurde, bleibt dieselbe. Es ist jedoch auch möglich den Würfel neu zu orientieren, wovon hier jedoch abgesehen werden soll. Die Kennzeichung der Seiten wird außerdem zur Beschreibung der Cubies, Cubicles und Facelets verwendet. Das Cubicle des Kantensteins zwischen der vorderen und oberen Seite wird beispielsweise als Up-Front oder Front-Up Cubicle bezeichnet. Der obere rechte Eckstein der Vorderseite erhält die Kennzeichung Up-Right-Front Cubie. Die Bezeichungen für Cubie und Cubicles werden durch Groß- bzw. Kleibuchstaben unterschieden. Die vier oberen Cubicles der Kantensteine lassen sich somit über *uf, ul, ub, ur* beziehungsweise *fu, lu, bu oder ru* beschreiben. Die zugehörigen Steine sind dementsprechend *UF, UL, UB, UR* beziehungsweise *FU, LU, BU oder RU*. Nach demselben Schema werden auch die anderen Cubicles und Cubies der restlichen Kanten- und Ecksteine beschrieben. Der obere rechte Eckstein wird mit *URF* abgekürzt und sein Heimstandort mit *urf*. Die Reihenfolge der Seitenkürzel wird hierbei immer im Uhrzeigersinn notiert, welche sich durch Betrachtung einer diagonalen Linie von der äußeren Ecke des Würfels zur Mitte des Würfels ergibt (vgl. Abbildung 3.6).

URF, RFU und *FUR* bezeichnen somit alle denselben Würfelstein, dessen Heimstandort der *urf* Cubicle ist. Die verwendete Reihenfolge der Seitenkürzel zeigt die Orientierung eines Steins in dem Cubicle an, weshalb *UFR, FRU* und *RUF* niemals verwendet werden. Befindet sich der *BLD* Cubie in dem *urf* Cubicle bedeutet dies, dass der hintere Eckstein

links-unten sich in der unteren rechten Position der Vorderseite befindet und die Farbe der hinteren Seite (Back) auf der Oberseite (Up), die Farbe der linken Seite (Left) auf de rechten Seite (Right) und die Farbe der unteren Seite (Down) auf der vorderen Seite (Front) befindet:

- $B \rightarrow u$
- $L \rightarrow r$
- $D \rightarrow f$

Die Reihenfolge der Facelets wird genutzt, um die Bewegung der Steine zu beschreiben, welche sich aus der Drehung der Würfelseiten ergeben. Wird ein Stein aus dem *ufl* Cubicle zum *drb* Cubicle bewegt, wird dies wie folgt notiert: $ufl \rightarrow drb$. Dadurch wird ausgedrückt, dass das Facelet der oberen zur unteren, von der vorderen zur rechten und von der linken zur unteren Seite bewegt wird:

- $u \rightarrow d$
- $f \rightarrow r$
- $l \rightarrow b$

Die gleiche Notationsweise wird für Kantensteine verwendet. Beispielsweise $rb \rightarrow bu$. Auch eine Verdrehung mit dem Uhrzeigersinn eines Ecksteins ($urf \rightarrow rfu$) sowie eine Umkehrung eines Kantensteins ($fl \rightarrow lf$) lassen sich mit dieser Kennzeichung aufzeigen.

Die Rotation der Würfelseiten wird über die Abkürzungen von diesen in Großbuchstaben gekennzeichnet: (U, D, F, B, R und L), wobei ein Kürzel für eine Vierteldrehung der jeweiligen Seite steht. Wird eine Seite um mehr als eine Vierteldrehung rotiert, wird der Buchstabe um die entsprechende Potenz ergänzt. Ein Quadrat bedeutet hierbei eine halbe Drehung (U^2, D^2, F^2, B^2, R^2, L^2) und ein Kubik eine dreiviertel Drehung (U^3, D^3, F^3, B^3, R^3, L^3). Dabei entspricht eine dreiviertel Drehung einer Vierteldrehung gegen den Uhrzeigersinn, welche auch als inverse Drehung bezeichnet und mit einer negativen Potenz gekennzeichet wird (U^{-1}, D^{-1}, F^{-1}, B^{-1}, R^{-1}, L^{-1}). Eine Abfolge von Bewegungen wird durch einfache Aneinanderreihung der Buchstaben ausgedrückt. Somit bedeutet FR zum Beispiel, dass zuerst die vordere und anschließend die rechte Seite des Würfels rotiert wird. Solch eine Abfolge wird auch Prozess (*Process*) genannt. Um zu zeigen, dass ein Cubicle zu einem anderen bewegt wird, ist es üblich, zunächst die Bewegung und anschließend den zugehörigen Prozess niederzuschreiben: $uf \rightarrow ru$: FR.

Jeder Prozess resultiert in einer Neuanordung der Facelets beziehungsweise Cubies. Diese Neuanordung wird auch als Permutation bezeichnet. Eine Rotation der oberen Seite wird mit der bereits erläuterten Kennzeichnung notiert:

- $uf \to ul$
- $ufl \to ulb$
- $ul \to ub$
- $ulb \to ubr$
- $ub \to ur$
- $ubr \to urf$
- $ur \to uf$
- $urf \to ufl$

Wird ein Cubicle nicht bewegt, so wird dieser auch nicht in der Liste mit aufgeführt. Die Verdrehung eines Ecksteins oder die Umkehrung eines Kantensteins wird entsprechend notiert. So bedeuted $ubr \to bru$ eine Verdrehung um 120° im Uhrzeigersinn. Um eine Reorientierung des Würfels zu kennzeichnen, verwendet David Singmaster kursive Großbuchstaben, wie zum Beispiel $\mathcal{R}$, womit das Rotieren des Würfels nach rechts ausgedrückt wird. Auch hier wird über Potenzen die Anzahl der Neuorientierungen gekennzeichnet.

Optimalität und Gottes Zahl

Erst einen Monat nachdem Ernő Rubik den Zauberwürfel erfand, entwickelte er einen Weg, diesen auch zu lösen. Mittlerweile gibt es eine Vielzahl von Algorithmen, um einen verdrehten Rubik's Cube wieder in seinen Urzustand zu versetzen. Darunter befindet sich auch eine anfängerfreundliche Methode, entwickelt von Jessica Fridrich, die sich noch immer großer Beliebheit erfreut[1]. Die unterschiedlichen Verfahren zum Lösen eines Würfels unterscheiden sich vor allem in der Anzahl der benötigten Verdrehungen der Würfelseiten. Methoden für Anfänger können um die 100 Züge benötigen, welche sich auf verschiedene Prozesse mit jeweils ungefähr 5-7 Aktionen verteilen. Fridrichs Verfahren benötigt im Durchschnitt um die 55 Züge, wobei man sich um die 120 Prozesse merken

[1] https://www.speedcube.de/fridrich.php - Zugriffsdatum: 07.07.23

muss [18]. Eine Frage, die sich bei dem Lösen des Würfels stellt, ist, wie viele Züge mindestens notwendig sind, um den Lösungszustand zu erreichen. Diese konnte bis zum Jahr 2010 nicht vollständig beantwortet werden konnte. Im Jahr 2010 entdeckte eine Forschergruppe von Google, dass höchstens 20 Züge erforderlich sind, um den Zauberwürfel aus jedem beliebigen Zustand in seinen Ursprungszustand zurückzuführen. Diese Zahl wird auch als *Gottes Zahl* bezeichnet [18]. Häufig ist die Anzahl der benötigten Aktionen sogar wesentlich geringer. Von der Forschungsgruppe wurde allerdings nur bewiesen, wie viele Schritte notwendig sind und nicht wie man zur optimalen Lösung gelangt.

3.1.2 Stand der Technik

Mittlerweile stehen eine Vielzahl unterschiedlicher Methoden zur Verfügung, den Rubik's Cube mit Hilfe eines Computers zu lösen. Diese lassen sich von ihrem Lösungsansatz her in zwei Kategorien aufteilen:

- Lösungsverfahren basierend auf Gruppentheorie [30] und Pattern-Datenbanken
- Lösen mittels künstlicher Intelligenz

Eines der bekanntesten Beispiele, welches mit Gruppentheorie und Datenbanken arbeitet, ist das Programm *Cube Explorer*[1], welches von Herbert Kociemba entwickelt wurde. Anstatt auf die verschiedenen Softwares einzugehen, welche dieselben Verfahren inkorporieren, ist es sinnvoller, die Methoden, auf welchen diese basieren, kurz vorzustellen.

Lösungsalgorithmen

Thistlewaite's Algorithmus: Der nachfolgende Abschnitt ist eine Zusammenfassung der Quelle[2]. Der Thistlewaite's Algorithmus ist ein etwas komplexerer Ansatz den Rubik's Cube zu lösen und war nichtsdestotrotz lange Zeit die Lösungsvariante mit den wenigsten durchschnittlichen Zügen (52). Entwickelt wurde diese Variante im Jahre 1981 von Morwen Thistlewaite. Eine besondere Eigenschaft des Algorithmus ist, dass er gleichzeitig an der korrekten Positionierung und Orientierung aller Cubies arbeitet und nicht jeden nach und nach zu seiner Home Location befördert. Dies erfolgt nach folgendem Ablauf:

[1] `http://kociemba.org/cube.htm` - Zugriffsdatum: 07.07.23
[2] `https://www.jaapsch.net/puzzles/thistle.htm` - Zugriffsdatum: 07.07.23

1. Anwendung von Transformationen bis ein bekannter Würfelzustand auftritt, welcher ohne Vierteldrehungen von U und D gelöst werden kann. Halbe Drehungen von U und D sind jedoch noch zulässig.
2. Anschließend wird der Würfel weiter gelöst, ohne die Anwendung von Vierteldrehungen von U sowie D, bis ein Zustand auftrifft, der zusätzlich keine Vierteldrehungen von F oder B benötigt.
3. Dies wird fortgeführt bis ein Würfelzustand eintritt, der gar keine Vierteldrehungen mehr für seine Lösung erfordert.

Mathematisch wird der Zustandsraum in Gruppen aufgeteilt, wobei jede nachfolgende Gruppe eine Untergruppe von der vorherigen ist und jeder neue Stand des Lösungsverfahrens eine Lookup-Tabelle, welche eine Lösung für jedes Element des Quotientenraumes [10] aufzeigt. Ein Überblick über die Gruppen ist der nachfolgenden Gleichung zu entnehmen:

$$\begin{aligned} G_0 &= \{L, R, F, B, U, D\} \\ G_1 &= \{L, R, F, B, U^2, D^2\} \\ G_2 &= \{L, R, F^2, B^2, U^2, D^2\} \\ G_3 &= \{L^2, R^2, F^2, B^2, U^2, D^2\} \end{aligned} \tag{3.1}$$

Kociemba Algorithmus: Der Kociemba Algorithmus[1], welcher vom gleichnamigen Forscher Herbert Kociemba in 1992 erfunden wurde, ist eine effektive Methode den Zauberwürfel in wenigen Zügen zu lösen. Das Verfahren ist auch bekannt als Zweiphasen-Algorithmus und baut auf Thistlewaite's Algorithmus auf. Allerdings werden bei dem Ansatz von Kociemba nur zwei Phasen benötigt statt der vier des Thistlewaite's. In der ersten Phase wird versucht, einen Würfelzustand zu erreichen, welcher nur mit Hilfe der Gruppe

$$G_1 = \{L^2, R^2, F^2, B^2, U, D\} \tag{3.2}$$

gelöst werden kann. Diese ist äquivalent zur Gruppe G_2 aus Gleichung 3.1 des Thistlewaite's Algorithmus. Dafür stehen die grundlegenden Transformationen, bestehend aus den Vierteldrehungen, zur Verfügung:

$$G_0 = \{L, R, F, B, U, D\} \tag{3.3}$$

[1] http://kociemba.org/cube.htm - Zugriffsdatum: 07.07.23

In der zweiten Phase wird der Würfel mit den zur Verfügung stehenden Aktionen aus G_1 gelöst. Hierbei benötigt Phase 1 maximal 12, Phase 2 maximal 18 und der Algorithmus insgesamt höchstens 30 Züge zum Lösen des Würfels.

Korf's Algorithmus: Korf's Algorithmus [17] ist dafür bekannt, eine optimale Lösung für den Zauberwürfel in einem realistischen Zeitraum zu finden. Die Methode wurde von Richard Korf im Jahre 1997 entwickelt, welche auf einem von ihm entworfenen Suchbaum-Algorithmus namens *Iterative Deepening A** (kurz IDA*) basiert. Mit dessen Hilfe konnte er nachweisen, dass die Gottes Zahl für den Rubik's Cube höchstens 20 beträgt, was bedeutet, dass jeder Würfel in maximal 20 Zügen gelöst werden kann, unabhängig vom Ausgangszustand. Der Korf Algorithmus ist eine Abwandlung des A*-Suchalgorithmus, welcher mit einer Tiefensuche kombiniert wurde, deren Effizienz mit zunehmender Rechenleistung steigt. Dieses Verfahren durchsucht den Zustandsraum des Rubik's Cube, indem es alle möglichen Konfigurationen systematisch generiert und untersucht. Um die Suche zu leiten, wird eine heuristische Kostenfunktion genutzt, welche die Distanz mit Hilfe einer unteren Grenze (*Lower Bound*) abschätzt. Diese Grenze gibt an, wie viele Aktionen noch mindestens notwendig sind, um den Würfel in den Lösungszustand zu bringen. Nicht vielversprechende Abzweigungen des Suchbaums werden hierbei „beschnitten“ und nicht weiter beachtet. Durch die Auswahl der Abzweigung, welche eine immer kleinere Schätzung bis zum Zielzustand besitzt als der aktuelle Zustand, wird sich der Lösung schrittweise angenähert. Es wird also nach Zuständen gesucht, welche immer weniger Schritte zur Lösung benötigen, bis der Zielzustand schlussendlich erreicht wird.

Lösungsverfahren mit künstlicher Intelligenz

Der große Zustandsraum des Rubik's Cube macht es besonders schwierig, eine künstliche Intelligenz darauf zu trainieren, den Würfel eigenständig zu lösen. Dies liegt vor allem daran, dass es nicht einfach ist, genügend Trainingsdaten für das Anlernen des neuronalen Netzes zu erzeugen. Diese sind für das Netzwerk notwendig, um Informationen aus den Daten zu generalisieren, damit dieses auch in Zuständen, die beim Training nicht vorgekommen sind, die richtige Aktion auswählt. Deshalb nutzen zwei der bekanntesten Verfahren Suchalgorithmen, welche mit einem neuronalen Netz kombiniert werden, um die Suche zu leiten. Da diese Arbeit die beiden Verfahren als Grundlage verwen-

det, werden diese im weiteren Verlauf noch näher erläutert und hier zunächst nur kurz vorgestellt.

DeepCube DeepCube [21] wurde von der Forschungsgruppe um Stephen McAleer entwickelt und kombiniert ein neuronales Netz mit einer Monte-Carlo-Baumsuche (*Monte-Carlo Tree Search*, kurz MCTS). Der Algorithmus verwendet ein tiefes neuronales Netzwerk, um die Güte und die beste Aktion in dem aktuellen Würfelzustand vorherzusagen, und wendet anschließend MCTS an, um den Suchraum zu erkunden und eine Lösung zu finden.

DeepCubeA DeepCubeA [22] ist eine Weiterentwicklung von DeepCube und basiert auf demselben Ansatz wie sein Vorgänger. Auch dieses Verfahren wurde von McAleer et al. entwickelt und kombiniert ein neuronales Netz mit einem Suchalgorithmus. Es wird jedoch die A*-Suche verwendet, welche wieder von einem neuronalen Netzwerk geleitet wird und ähnlich wie die Methode von Korf für jeden angetroffenen Zustand die Entfernung bis zum Zielzustand abschätzt.

3.2 Suchalgorithmen

Suchalgorithmen besitzen eine Reihe von möglichen Einsatzgebieten und werden beispielsweise verwendet, um einen Pfad zwischen zwei gegebenen Punkten zu finden. Dabei ist nicht immer nur die Strecke dieses Pfades von Interesse, sondern auch wie lange es dauert, diese zurückzulegen. In den meisten Fällen wird der kürzeste Weg gesucht und es können beispielsweise sogenannte Graphensuchverfahren eingesetzt werden. Dafür werden zu lösende Probleme, wie zum Beispiel das Finden der kürzesten Strecke zwischen zwei Punkten, als Graph dargestellt, um diese in eine abstrakte Netzstruktur zu zerlegen und mit Hilfe solcher Algorithmen zu lösen. Je nach Anwendung, zur Verfügung stehender Zeit und Ressourcen gibt es eine Vielzahl verschiedener solcher Algorithmen. Da im späteren Verlauf dieser Arbeit solche Verfahren zum Einsatz kommen, sollen einige grundlegende, teilweise aufeinander aufbauende Algorithmen erläutert werden. Vor allem die Monte-Carlo-Baumsuche und der A*-Algorithmus sind für diese Arbeit von Bedeutung und da Letzterer quasi eine Weiterentwicklung der Breitensuche (*Breadth-First Search*, kurz BFS) beziehungsweise des Dijkstra-Algorithmus ist, sollen diese zunächst näher betrachtet werden. Die folgenden Abschnitte sind eine Zusammenfassung einer

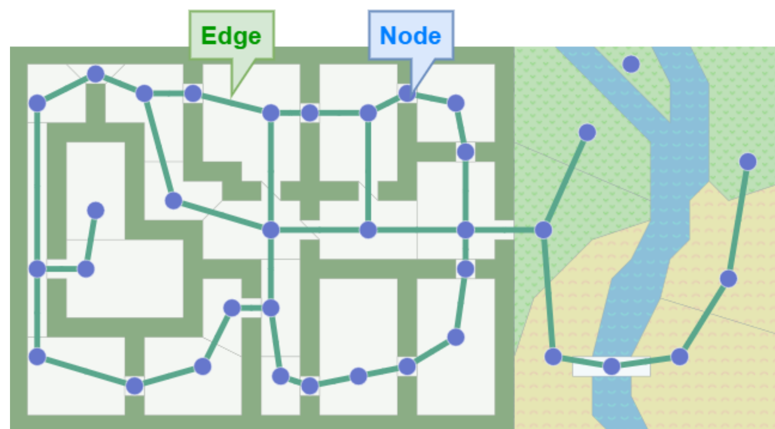


Abbildung 3.7: Beispielkarte mit eingezeichneter Graphendarstellung [26, modifiziert].

sehr gelungenen Darstellung von Amit Patel (Alumni Stanford Universty) und seiner Internetseite [26], wo sich die Algorithmen mit Animationen noch einfacher nachvollziehen lassen.

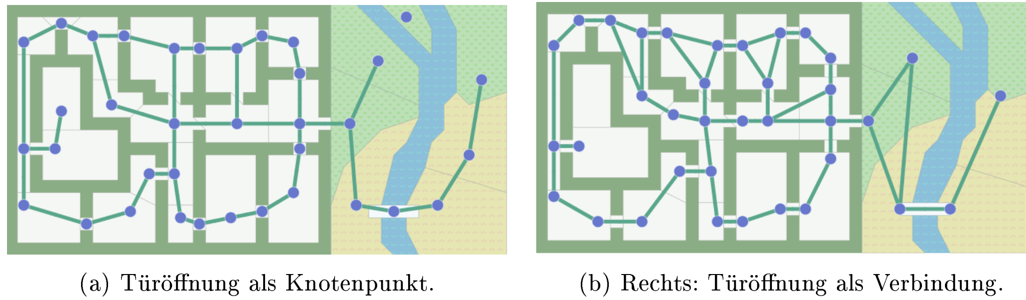
3.2.1 Darstellung

Patel erläutert, dass zuallerserst Eingabe und Ausgabe eines Algorithmus verstanden und festgelegt werden müssen. Bei Graphensuchverfahren sind Graphen, welche aus Knoten (*Nodes*) und Verbindungen (*Edges*) zwischen diesen bestehen, der Input. Ein Beispiel ist in Abbildung 3.7 gegeben, wo Wegpunkte (Knoten) und deren Verbindungen auf einer Karte eingezeichnet wurden.

Auch der Output besteht aus einem Graphen, welcher jedoch nur den Lösungsweg beinhaltet. Die gefundene Lösung eines solchen Verfahrens ist stark abhängig von der Art und Weise wie das zu lösende Problem dargestellt wird, denn wo die Knoten gesetzt werden, ist reine Festlegungssache (vgl. Abbildung 3.8). Für weitere Informationen zur Graphendarstellung kann die Internetseite[1] genutzt werden. Dem Vorbild von Patel entsprechend werden aus Gründen der einfacheren Nachvollziehbarkeit nachfolgend ausschließlich Graphen mit Gitterstruktur verwendet.

Wie bereits erwähnt, ist der A*-Algorithmus eine Weiterentwicklung des Dijkstra Algorithmus, welcher wiederum selbst auf der Breitensuche aufbaut. Deshalb erscheint es

[1] http://theory.stanford.edu/~amitp/GameProgramming/MapRepresentations.html - Zugriffsdatum: 07.07.23



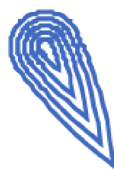
(a) Türöffnung als Knotenpunkt. (b) Rechts: Türöffnung als Verbindung.

Abbildung 3.8: Verschiedene Positionierungsmöglichkeiten für Knotenpunkte in einem Graphen [26].

zweckmäßig diese in umgekehrter Reihenfolge näher zu beschreiben. Nachfolgend ist eine kurze Übersicht dargestellt, welche [26] entnommen ist und minimal abgeändert wurde:

Breitensuche: Erkundet gleichmäßig in alle Richtungen.

Dijkstra Algorithmus: Erkundet Richtungen mit niedrigen Kosten zuerst. Kosten können vorher festgelegt werden, um günstgige Pfade zu priorisieren. Findet Pfade zu allen möglichen Wegpunkten.



A*-Algorithmus: Modifikation des Dijkstra Algorithmus, welcher für das Finden eines einzigen Zieles optimiert ist.

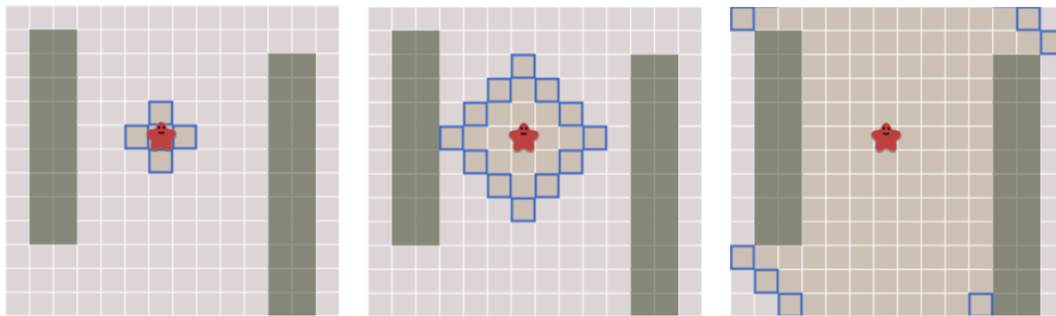
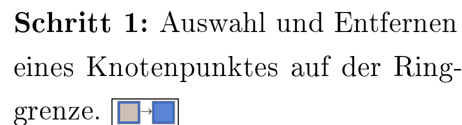


Abbildung 3.9: Expandierung der Ringgrenze beim Breitensuchverfahren [26].

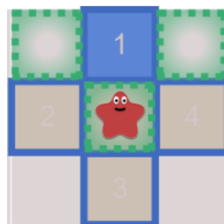
3.2.2 Breitensuche

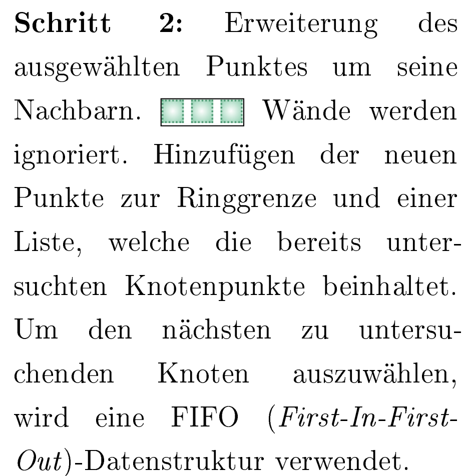
Patel legt auf seiner Website nahe, dass die Grundidee der drei Algorithmen ein expandierender Ring ist, welcher auch Grenze (*Frontier*) genannt wird. Das Prinzip ist Abbildung 3.9 zu entnehmen.

Für das Verfahren sind zwei Schritte notwendig, welche wiederholt werden, bis alle Knoten einer Karte erreicht wurden:



Schritt 1: Auswahl und Entfernen eines Knotenpunktes auf der Ringgrenze.





Schritt 2: Erweiterung des ausgewählten Punktes um seine Nachbarn. Wände werden ignoriert. Hinzufügen der neuen Punkte zur Ringgrenze und einer Liste, welche die bereits untersuchten Knotenpunkte beinhaltet. Um den nächsten zu untersuchenden Knoten auszuwählen, wird eine FIFO (*First-In-First-Out*)-Datenstruktur verwendet.

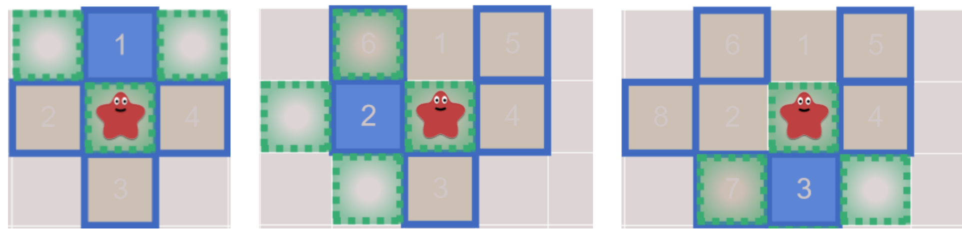


Abbildung 3.10: Expandierung der Ringgrenze bei dem Breitensuchverfahren im Detail [26].

Auf der Internetseite [26] ist der Codeausschnitt aus Listing 3.1 gegeben, um den Algorithmus in Python umzusetzen.

```
frontier = Queue()
frontier.put(start)
reached = set()
reached.add(start)

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in reached:
            frontier.put(next)
            reached.add(next)
```

Listing 3.1: BFS-Algorithmus umgesetzt in Python [26].

Das grundlegende Verfahren der Breitensuche ist noch einmal graphisch in Abbildung 3.10 dargestellt.

Die Schleifen in Listing 3.1 sind bereits die Essenz der Graphensuchalgorithmen (BFS, Dijkstra, A*). Allerdings erzeugen diese keine Pfade, sondern zeigen auf, wie jeder Punkt auf dem gegebenen Graphen erreicht werden kann. Durch das Tracken, von welchem anderen Knoten jeder Punkt erreicht wurde, lassen sich ganze Wege rekonsturieren, somit insbesondere auch vom Ziel- zum Startpunkt (vgl. Abbildung 3.11). Im Code lässt sich dies durch das Ersetzen des „reached“-Sets mit einem „came from“-Dictionary verwirklichen. Durch diese Anpassung zeigt jeder Knotenpunkt auf seinen vorherigen Punkt, über welchen dieser im Algorithmus erreicht wurde. Dies soll über die Pfeile in Abbildung 3.11 verdeutlicht werden. Die Keys des Dictionaries entsprechen dabei dem usprünglichen „reached“-Set. Der angepasste Code in Listing 3.2 ist wieder [26] entnommen.

```
frontier = Queue()
frontier.put(start ★)
came_from = dict() # path A->B is stored as came_from[B] == A
came_from[start] = None

while not frontier.empty():
    current = frontier.get()
    for next in graph.neighbors(current):
        if next not in reached:
            frontier.put(next)
            came_from[next] = current
```

Listing 3.2: Angepasster BFS-Algorithmus umgesetzt in Python [26].

Abbildung 3.11: Pfadfindung mittels Breitensuche und angepasstem Python-Code [26].

Ein Pfad besteht hierbei aus den Verbindungen zwischen den zusammenhängenden Knotenpunkten. Oftmals ist es jedoch einfacher, Knoten zu speichern. Um den Lösungspfad zu rekonsturieren kann der Codeausschnitt aus Listing 3.3 genutzt werden.

```
current = goal

path = []
while current != start:
    path.append(current)
    current = came_from[current]
path.append(start) # optional
path.reverse() # optional
```

Listing 3.3: Lösunsgwegfindung mittel angepasstem BFS-Algorithmus [26].

Dieser hier erläuterte Algorithmus ist laut Patel der einfachste Pfadfindealgorithmus und funktioniert mit jeder beliebigen Graphenstruktur.

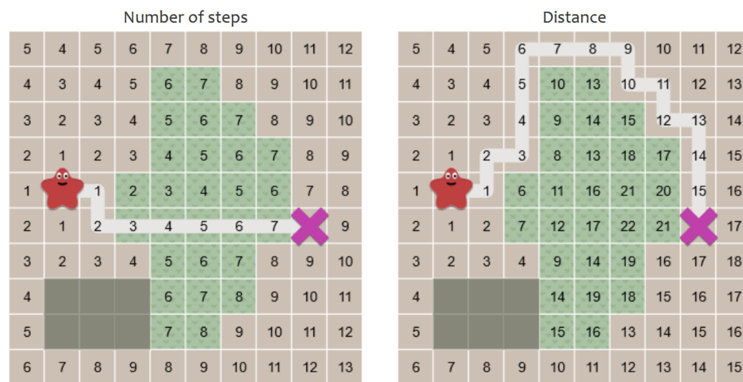


Abbildung 3.12: Anzahl der Schritte (links) im Vergleich zu der Distanz (rechts) [26].

Frühzeitiges Unterbrechen

Um nicht die gesamte Karte mit der Breitensuche zu erkunden, sondern nur bis zu dem gewünschten Zielpunkt, bietet es sich an den Algorithmus frühzeitig zu stoppen (*Early Exit*) (vgl. Listing 3.4). Dies wird durch die einfache Abfrage ermöglicht, ob der aktuell zu untersuchende Knotenpunkt der Zielpunkt ist:

```
if current == goal:
    break
```

Listing 3.4: Einfache if-Abfrage, um einen Suchalgorithmus frühzeitig zu unterbrechen [26].

Bewegungskosten

Bisher kostete jede Bewegung im vorgestellten Verfahren dasselbe. Dies ist jedoch nicht immer der Fall. Bei Veränderung der Umgebung (im nachfolgenden exemplarisch Wald, Ozean und Wüste) müssten die Bewegungskosten den Bedingungen angepasst werden. Das Durchschreiten einer Wüste könnte beispielsweise pro Bewegung nur einen Punkt und das Durchwandern eines Waldes fünf Punkte kosten. Es lässt sich zudem die Art der Bewegung in Betracht ziehen. Eine diagonale Bewegung könnte zum Beispiel das Doppelte von einer axialen Bewegung erfordern. Durch das Einbeziehen dieser Faktoren unterscheidet sich die Anzahl der Schritte zum Ziel von der gewichteten Distanz (vgl. Abbildung 3.12).

Für diese Art von Graphensuchalgorithmen ist der Dijkstra-Algorithmus besonders gut geeignet.

3.2.3 Dijkstra Algorithmus

Der Unterschied zwischen BFS und Dijkstra ist laut Patel das zusätzliche Loggen der Bewegungskosten. Über das Hinzufügen einer zusätzlichen Variable „cost_so_far“ lässt sich dies im Python-Code realisieren. Diese Kosten werden anschließend bei der Bewertung eines Knotenpunktes miteinbezogen. Für dieses Verfahren wird die bisherige Queue zu einer Vorrangwarteschlange (*Priority Queue*) abgeändert, welche ihre Elemente automatisch nach der höchsten Priotität (hier die Wegkosten) sortiert. Um zu verhindern, dass derselbe Punkt mehrfach besucht wird, werden nur noch Knoten hinzugefügt, welche noch nicht besucht wurden oder deren Pfadkosten geringer sind als die des bisherigen Weges. Die Änderungen sind Listing 3.5 zu entnehmen.

```
1  frontier = PriorityQueue()
2  frontier.put(start, 0)
3  came_from = dict()
4  cost_so_far = dict()
5  came_from[start] = None
6  cost_so_far[start] = 0
7
8  while not frontier.empty():
9      current = frontier.get()
10
11     if current == goal:
12         break
13
14     for next in graph.neighbors(current):
15         new_cost = cost_so_far[current] + graph.cost(current, next)
16         if next not in cost_so_far or new_cost < cost_so_far[next]:
17             cost_so_far[next] = new_cost
18             priority = new_cost
19             frontier.put(next, priority)
20             came_from[next] = current
```

Listing 3.5: Implementation des Dijkstra-Algorithmus in Python [26].

Durch die Modifikationen verändert sich die Art, wie sich die Ringgrenze ausbreitet. Das Beispiel in Abbildung 3.13 demonstriert, wie sich das Verfahren verhält, wenn ein Terrain (Wald) andere Bewegungskosten besitzt als beispielsweise eine Ebene.

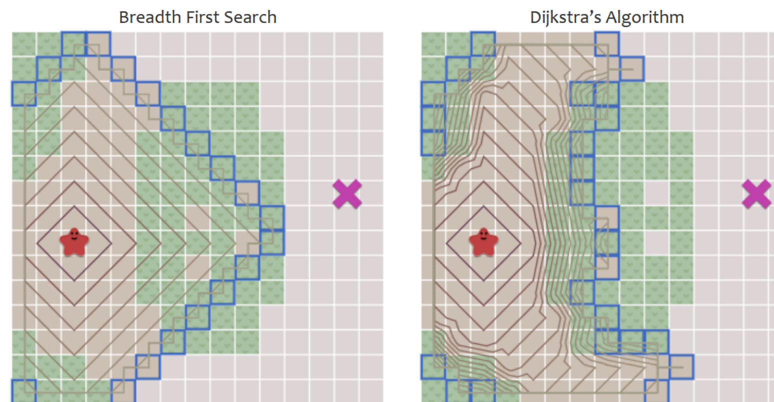


Abbildung 3.13: Ausbreitung der Ringgrenze bei dem Verfahren nach Dijkstra im Vergleich zum Breitensuchverfahren [26].

Patel erläutert, dass sich durch die Verwendung von Bewegungskosten auch Graphen mit interessanteren Strukturen untersuchen lassen als solche mit einer Gitterstruktur. Zudem lassen sich bestimmte Areale vermeiden oder bevorzugen.

3.2.4 Heuristische Suche

Bei den bisher dargestellten Algorithmen breitet sich die Grenze immer gleichzeitig in alle Richtungen aus. Häufig wird jedoch nur der kürzeste Pfad zu einem Punkt gesucht. Damit sich die Grenze mehr in die Richtung des gewünschten Ziels ausbreitet, kann eine heuristische Funktion genutzt werden, welche Aufschluss darüber gibt, wie nah sich ein Punkt an dem Ziel befindet. In Listing 3.6 wird hierfür beispielsweise die Manhatten-Distanz verwendet.

```
1 def heuristic(a, b):
2     # Manhattan distance on a square grid
3     return abs(a.x - b.x) + abs(a.y - b.y)
```

Listing 3.6: Heuristische Funktion zur Entfernungsschätzung mittels Manhatten-Metrik [26].

Statt einer Vorrangwarteschlange, welche die tatsächliche Entfernung nutzt (Dijkstra), wird hier eine implementiert, die die geschätzte Entfernung der heuristischen Funktion verwendet. Diese Art der Suche wird auch *Greedy Best-First Search* (kurz GBFS) genannt, welche die Orte, die sich nah am Ziel befinden, zuerst untersucht. Um dies in

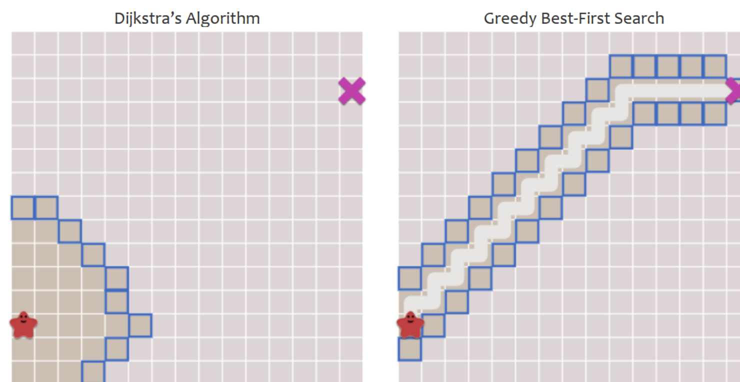


Abbildung 3.14: Vergleich zwischen Dijkstra und Greedy Best-First Search [26].

Python umzusetzen, wird das Gerüst des Dijkstra-Algorithmus (vgl. Listing 3.5) verwendet und die Variable „cost_so_far“, welche die bisherigen Bewegungskosten speichert, entfernt (vgl Listing 3.7).

```
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
came_from[start] = None

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        if next not in cost_so_far or new_cost < cost_so_far[next]:
            priority = heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

Listing 3.7: Implementation von Greedy Best-First Search in Python [26].

Ein Vergleich zwischen Dijkstra und GBFS ist der Abbildung 3.14 zu entnehmen.

Der in Abbildung 3.14 ersichtliche Vorteil von GBFS ist jedoch nur für einfache Graphenstrukturen gegeben. Das Verfahren besitzt Schwierigkeiten, wie in Abbildung 3.15 exemplarisch dargestellt, bei komplexeren Umgebungen mit zusätzlichen Hindernissen. Greedy Best-First Search ist somit in der Regel zwar wesentlich schneller als beispielsweise Dijkstra, aber die Qualität der gefundenen Pfade ist stark abhängig von dem gegebenen Graphen.

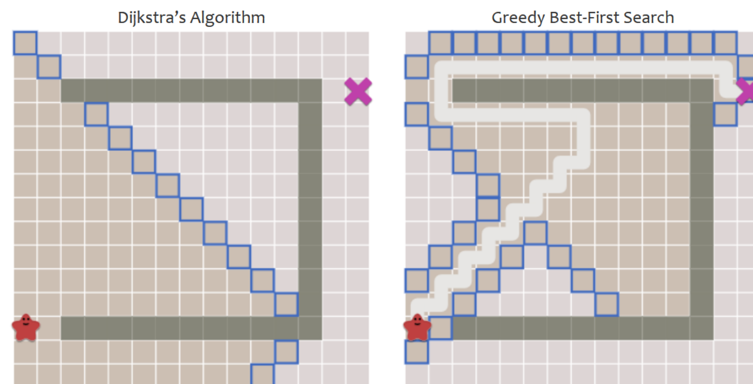


Abbildung 3.15: Vergleich zwischen Dijkstra und Greedy Best-First Search in einer komplexeren Umgebung [26].

3.2.5 A*-Algorithmus

Patel führt aus, dass der Dijkstra-Algorithmus zwar den kürzesten Weg findet, jedoch viel Zeit aufwendet, alle Richtungen zu erkunden. GBFS ist dafür um einiges schneller, findet, wie bereits erwähnt, jedoch nicht immer den kürzesten Pfad. Durch die Kombination der Logik beider Algorithmen lässt sich das Beste aus beiden Bereichen miteinander vereinen und man erhält den A*-Algorithmus. Der Code sieht dem des Dijkstra sehr ähnlich, besitzt aber Variablen für die tatsächlich bisher zurückgelegte Distanz und die geschätzte Entfernung bis zum Ziel. Die nötigen Anpassungen sind Listing 3.8 zu entnehmen. In

```python
frontier = PriorityQueue()
frontier.put(start, 0)
came_from = dict()
cost_so_far = dict()
came_from[start] = None
cost_so_far[start] = 0

while not frontier.empty():
    current = frontier.get()

    if current == goal:
        break

    for next in graph.neighbors(current):
        new_cost = cost_so_far[current] + graph.cost(current, next)
        if next not in cost_so_far or new_cost < cost_so_far[next]:
            cost_so_far[next] = new_cost
            priority = new_cost + heuristic(goal, next)
            frontier.put(next, priority)
            came_from[next] = current
```

Listing 3.8: Implementation des A*-Algorithmus in Python [26].

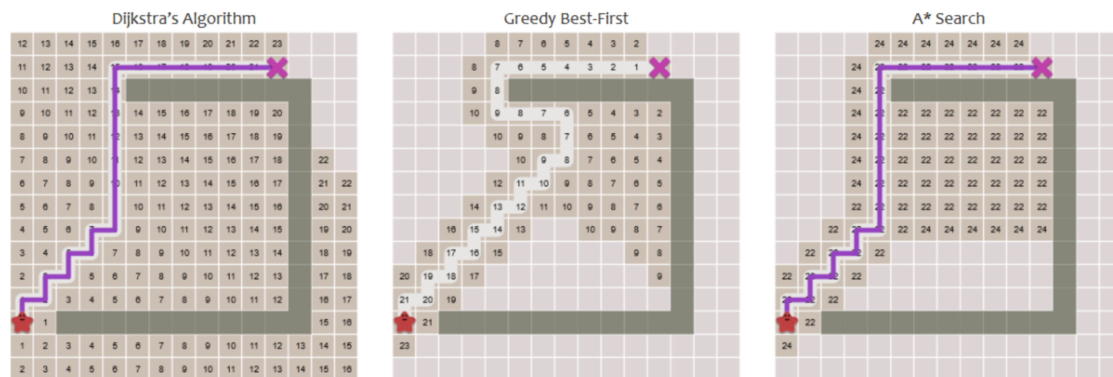


Abbildung 3.16: Vergleich zwischen Dijkstra, Greedy Best-First Search und A* [26].

einem Vergleich zeigt Patel sehr anschaulich die Unterschiede zwischen den Algorithmen (vgl. Abbildung 3.16). Es wird ersichtlich, dass Dijkstra die Distanz vom Start berechnet, GBFS die Entfernung zum Ziel und A* die Summe der beiden Berechnungen nutzt. Um einen optimalen Pfad zu finden, ist es notwendig, dass im A*-Verfahren die heuristische Funktion die Entfernung nicht überschätzt. Dies gelingt vor allem dadurch, dass A* eine Heuristik nutzt, um die gespeicherten Knoten ihren Kosten nach so zu sortieren (von gering nach groß), sodass es wahrscheinlicher wird, den Zielpunkt früher zu erreichen. Überschätzt die Funktion die Distanz, ähnelt A* dem Verhalten des Greedy Best-First Search Algorithmus. Unterschätzt sie die Distanz bei Weitem, ähnelt A* eher dem Dijkstra Algorithmus.

3.2.6 Bestensuche

Ein Beispiel für die Bestensuche ist das Greedy Best-First Search Verfahren, welches bereits zuvor erläutert wurde. Ziel ist es, den kürzesten Weg zwischen zwei Punkten mit Hilfe einer einfachen Heuristik zu ermitteln, welche angibt, wie vielversprechend eine Bewegung im Bezug auf das Erreichen des Zielpunktes ist. Die Greedy Best-First Search bewertet, wie nah ein Knoten vom Ziel entfernt ist und wählt immer denjenigen aus, der diesem am nächsten ist. Dabei werden, wie bereits aufgezeigt, Hindernisse nicht berücksichtigt.

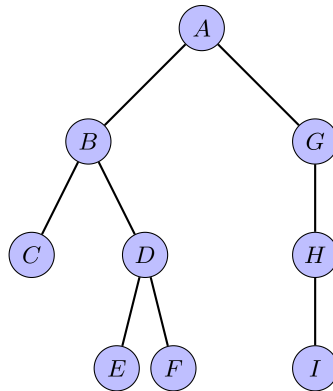


Abbildung 3.17: Ablauf der Tiefensuche. Die Buchstaben zeigen die Reihenfolge der untersuchten Knotenpunkte auf.

3.2.7 Tiefensuchen

Die Tiefensuche ist das Pendant zur Breitensuche. Bei der Breitensuche wird zunächst jeder Knoten einer Ebene untersucht, bevor zur nächsten Ebene übergegangen wird. Dadurch breitet sich das Suchverfahren kreisförmig aus (vgl. Abbildung 3.10 auf Seite 74) und wird wiederholt, bis der Zielpunkt erreicht oder eine Abbruchbedingung erfüllt wurde. Die Tiefensuche untersucht dagegen einen Pfad so weit wie möglich, bevor er diesen zurückverfolgt und einen neuen Pfad auswählt. Auch dies wird solange wiederholt, bis der Zielpunkt erreicht oder eine Abbruchbedingung erfüllt wird. Im Gegensatz zum BFS, wird hierbei eine LIFO (*Last-In-First-Out*)-Datenstruktur verwendet, um den nächsten zu untersuchenden Knoten auszuwählen. Das Verfahren ist in Abbildung 3.17 dargestellt.

3.2.8 Monte-Carlo-Baumsuche

Die Monte-Carlo-Baumsuche (*Monte-Carlo Tree Search*, abgekürzt MCTS) ist der Überbegriff für eine Reihe von Algorithmen, welche nach dem gleichen Prinzip funktionieren und sich für Markow-Entscheidungsprobleme nutzen lassen. Durch Anwendung aller in einem Zustand zulässigen Aktionen wird der Zustandsraum einer Umgebung (beispielsweise eines Spiels) erkundet [18]. MDPs lassen sich auch als Baumgraphen darstellen, welche als *ExpectiMax-Bäume* bekannt sind (vgl. Abbildung 3.18). Diese Art der Darstellung ist für ein besseres Verständnis des MCTS-Algorithmus vorteilhaft.

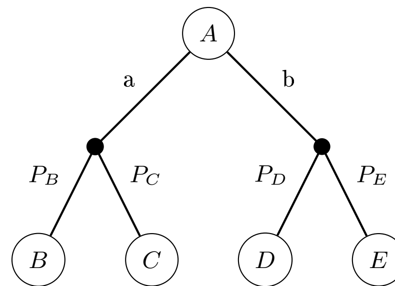


Abbildung 3.18: Darstellung eines MDPs als ExpectiMax-Baum.

In Abbildung 3.18 werden Zustände durch weiße Knotenpunkte mit Großbuchstaben und Aktionen durch Kleinbuchstaben an den Übergängen dargestellt. Schwarze Knotenpunkten repräsentieren die probalistische Unsicherheit: die „Umgebung“, welche auf Basis der Übergangsfunktion die Entscheidung trifft, welches Ergebnis einer Aktion eintrifft [23]. Knoten (A) ist hierbei der Start- und somit der Elternknoten von (B), (C), (D) und (E), welche dementsprechend die Kindknoten von (A) sind. Immer wenn der Agent in einem Markov Descion Process entscheiden muss, welche Aktion er in seinem aktuellen Zustand wählt, wird der Algorithmus erneut ausgeführt. Aktionswahl und -ausführung sind also miteinander verschachtelt.

Grundlegende Merkmale des MCTS sind [23]:

- Approximieren des Q-Values $Q(s, a)$ für State-Action-Paare (s, a) durch zufällige Simulation/Aktionsauswahl.
- Bei einem MDP mit nur einem Agenten wird der zugehörige Suchbaum inkrementell erstellt.
- Die Suche gilt als beendet, wird ein vordefiniertes Rechenbudget erreicht, beispielsweise ein Zeitlimit oder maximale Anzahl an Knotenpunkten.
- Die beste Aktion wird zurückgegeben. Diese gilt als vollständig, wenn keine Sackgassen aufgetreten sind, und als optimal, sofern der ganze Zustandsraum simuliert werden konnte.

Grundidee

Die zugrunde liegende Idee bei den MCTS-Algorithmen ist das Durchführen mehrerer sukzessiver Simulationen ausgehend von demselben Zustand. Darauf aufbauend werden

Folgezustände näher betrachtet, welche durch vorherige Simulationen hohe Bewertungen erhalten haben. In diesen simulierten Verläufen werden die Aktionen durch eine einfache *Rollout-Policy* ausgewählt. Der Begriff *Rollout* stammt hierbei von Tesauro und Galperin [1997], welche Rollout-Algorithmen mit Backgammon kombinierten und den Wert einer Position evaluierten, indem sie diese Position mit zufällig generierten Würfelsequenzen viele Male bis zum Ende des Spiels simulierten (*Rolling Out*) [32]. Bei Monte-Carlo-Baumsuchen wird die Güte von Zustand-Aktions-Paaren durch das Mitteln erhaltener simulierter Belohungen berechnet. Der Suchbaum wird inkrementell erweitert. Dies geschieht durch die nähere Betrachtung vielversprechender Folgezustände, welche durch die Simulationen ersichtlich werden. Sutton erläutert in seinem Buch [32], dass durch die schrittweise Vergrößerung des Baumes effektiv eine Lookup-Tabelle erzeugt wird, die einen Teil einer Action-Value-Funktion speichert. Durch jeden weiteren Durchlauf des Algorithmus wird der Baum durch Hinzufügen neuer Knoten vergrößert, und es werden die geschätzen Werte der Zustands-Aktions-Paare gespeichert.

Jede inkrementelle Erweiterung des Suchbaums unterteilt sich hierbei in vier Schritte, welche laufend wiederholt werden (vgl. Abbildung 3.19) [23]:

1. Selection: Auswahl eines Knotenpunktes, welcher noch nicht vollständig erweitert wurde. (Mindestens ein Kindknoten ist noch nicht näher untersucht worden.)
2. Expansion: Erweitern des Knotenpunktes durch Ausführung einer in diesem Zustand möglichen Aktion.
3. Simulation: Von der neuen Erweiterung aus werden zufällige Aktionen ausgeführt, bis eine Abbruchbedingung erreicht wird (Zielzustand, maximale Zeit oder Knoten überschritten).
4. Backpropagation (*Rückführung*): Die generierte Belohnung innerhalb der simulierten Episode wird genutzt, um die Güte der angetroffenen Knotenpunkte zurück bis zum Startpunkt zu aktualisieren.

Selection Beginnend bei dem Startknoten (*Root Node*) werden nacheinander jeweilige Kindknoten über eine Tree Policy ausgewählt (beispielsweise die weiter unten dargestellten UCB1 oder UCT-Score), bis ein noch nicht vollständig untersuchter Knoten (auch Blattknoten, *Leaf Node*) erreicht wird (vgl. Abbildung 3.20). Dabei ist die richtige Entscheidung nicht immer offensichtlich:

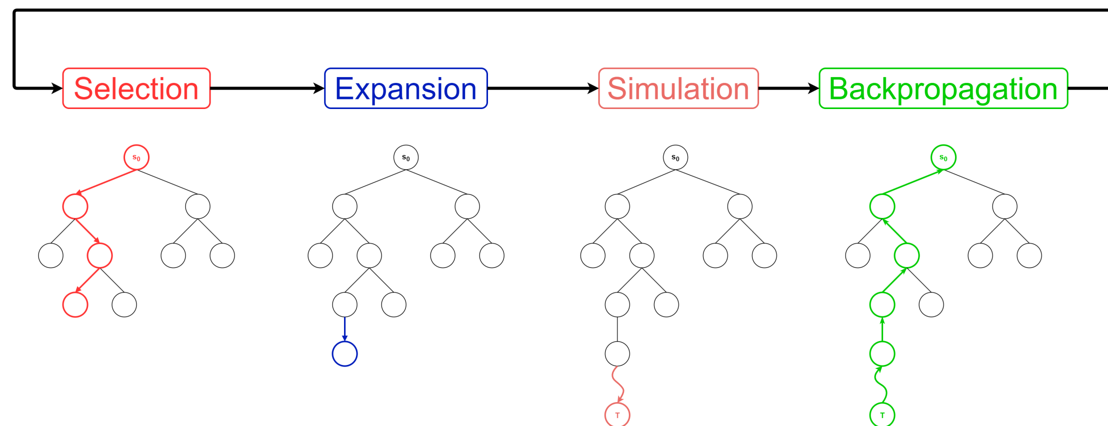


Abbildung 3.19: Ablauf einer Sequenz innerhalb einer Monte-Carlo-Baumsuche [2, 23, modifiziert].

- Auswahl der Abzweigung mit vielversprechenden Werten, welche sich unter Umständen im weiteren Verlauf deutlich verschlechtern.
- Auswahl der Abzweigung mit suboptimalen Werten, welche sich unter Umständen im weiteren Verlauf deutlich verbessern.

Dieses Problem ist auch bekannt als Exploration-Exploitation-Dilemma (vgl. Abschnitt 2.1.5 auf Seite 11). Die Tree Policy sollte also eine gute Balance zwischen Exploitation und Exploration finden, sodass einerseits Züge ausgewählt werden, welche vielversprechend wirken, und anderseits noch nicht erforschte Aktionen ausreichend untersucht werden, falls die bisherige Bewertung Unsicherheiten aufweist [2]. Zu intensives Erkunden des zu untersuchenden Baumes verschlechtert die Rechenleistung und möglicherweise das Ergebnis (beispielsweise längere Lösungswege zum Zielzustand durch Schleifenbildung). Zu wenig Erkunden verpasst möglicherweise einen kürzeren oder gar den einzigen Lösungsweg. Mögliche Tree Policies bei der Auwahl eines Knotenpunktes sind unter anderem:

$$\text{UCB1-Score [32]:} \quad A_t = \underset{a}{\operatorname{argmax}} \left[Q_t(a) + c\sqrt{\frac{\ln\, t}{N_t(a)}} \right] \tag{3.4}$$

$$\text{UCT-Score [23]:} \quad A_t = \underset{a \in A(s)}{\operatorname{argmax}} \left[Q(s,a) + 2c\sqrt{\frac{2\ln\, N(s)}{N(s,a)}} \right] \tag{3.5}$$

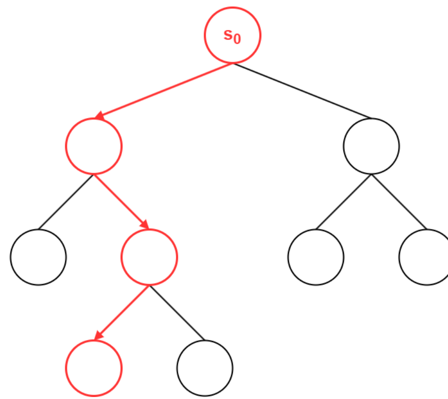


Abbildung 3.20: Selection: Auswahl eines Knotenpunktes [2, 23, modifiziert]. Pfad in rot dargestellt.

mit:

c	– Confidence Value, Balance zwischen Exploration und Exploitation
N_s	– Zähler, wie oft Zustand s_t besucht wurde
$N_{s,a}$	– Zähler, wie oft Aktion a im Zustand s_t ausgeführt wurde
$Q_t(a), Q(s,a)$	– Geschätzte Güte von Aktion a bzw. State-Action-Paar (s, a)

Expansion Der Graph wird durch das Anwenden von noch nicht erforschten Aktionen erweitert, wodurch ein oder mehrere Kindknoten hinzugefügt werden (vgl. Abbildung 3.21). Für jede simulierte Episode (beispielsweise eine Spielrunde) wird so jeweils ein neuer Knoten hinzugefügt.

Der Algorithmus für die Auswahl eines Knotens und dessen Erweiterung verhält sich dabei wie in Abbildung 3.22 dargestellt.

Simulation Bei dem Simulieren wird eine Reihe von zufälligen Aktionen ausgeführt, bis ein Zielzustand erreicht oder eine Abbruchbedingung erfüllt wird (vgl. Abbildung 3.23). Laut Chaslot [2] besitzen die Wahrscheinlicheiten der einzelnen Aktionen einen signifikanten Einfluss auf das simulierte Spielniveau. Chaslot begründet dies darin, dass wenn alle Aktionen dieselbe Wahrscheinlichkeit besitzen, die Spielstrategie schwach wäre und der Algorithmus im allgemeinen darunter leiden würde. Es erweist sich deshalb als sinnvoll heuristisches Vorwissen zu nutzen, um vielversprechenden Aktionen mehr Gewicht zu verleihen.

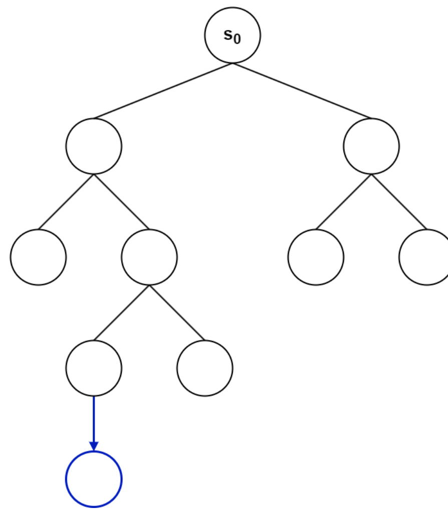


Abbildung 3.21: Expansion: Erweiterung eines Knotenpunktes [2, 23, modifiziert]. Pfad in blau dargestellt.

Backpropagation Der in der simulierten Episode erhaltene Reward stellt die Grundlage für die Berechnung der Güte V_s aller angetroffenen Zustände dar (vgl. Abbildung 3.24). Außerdem werden die Zähler der besuchten Zustände um eins erhöht und das Verhältnis zwischen Gewinn und Verlust angepasst [2]. Zum Schluss wird die Aktion tatsächlich ausgeführt, welche am häufigsten besucht wurde.

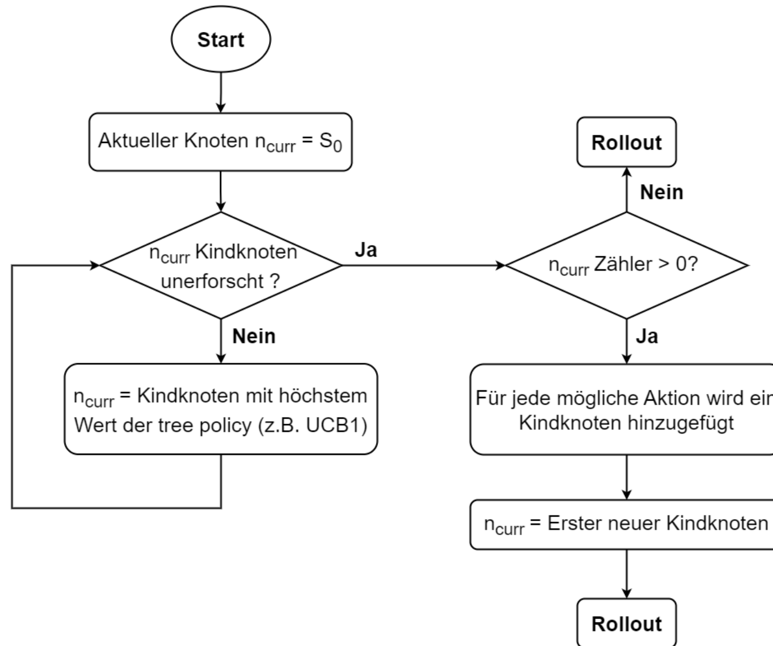


Abbildung 3.22: Algorithmus für die Auswahl und Erweiterung von Knotenpunkten [19, modifiziert].

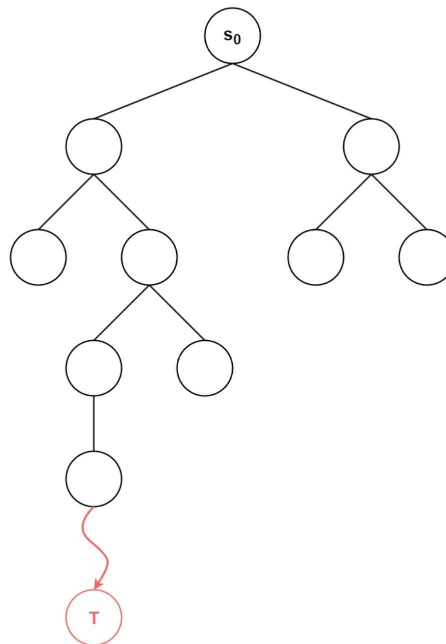


Abbildung 3.23: Simulation: Simulieren des Pfades des neuen Knotenpunktes bis zu einem Endzustand T [2, 23, modifiziert]. Pfad in rosa dargestellt.

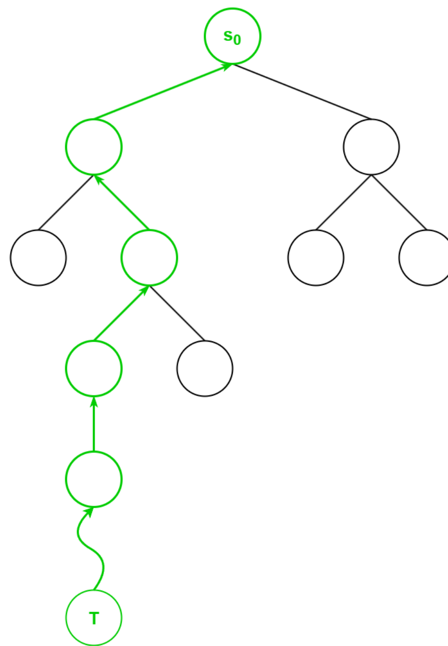


Abbildung 3.24: Backpropagation: Aktualisieren der Statistiken aller angetroffenen Knotenpunkte [2, 23, modifiziert]. Pfad in grün dargestellt.

4 Anforderungsanalyse

Der Fokus dieses Kapitels liegt auf der Analyse der Anforderungen, die für die Umsetzung dieses Projektes notwendig sind. Dabei spielen mehrere Faktoren, die Einfluss auf das fertige System haben, eine tragende Rolle. Einerseits gilt es zu ermitteln, wer die Stakeholder dieses Vorhabens sind, also die Personengruppen, für welche diese Arbeit von Interesse ist. Andererseits liegt ein großes Augenmerk auf der Bewältigbarkeit des Projektes in der vorgegebenen Zeit. Es gibt eine Vielzahl von Möglichkeiten, eine Software zu entwickeln, welche mittels neuronaler Netze einen Zauberwürfel löst, jedoch unterscheiden sich diese vor allem im Zeitaufwand, benötigten Ressourcen und anschließender Qualität. Somit gilt es herauszuarbeiten, welche Lösungsansätze durchführbar und anschließend auch qualitativ zufriendendstellend für die Stakeholder sind. Eine Systemeinheit, welche nach Fertigstellung dieses Projektes sowohl aus Hardware- sowie zugehöriger Software besteht, wäre wünschenswert. Aber auch eine reine Simulationssoftware, mit welcher schlussendlich weitergearbeitet werden könnte, wäre eine wertvolle Bereicherung für nachfolgende Arbeiten.

4.1 Beschreibung

4.1.1 Software

Das Hauptaugenmerk dieser Thesis soll auf dem Softwareteil liegen. Diese Arbeit soll es ermöglichen, Zauberwürfel mit einem neuronalen Netz zu lösen. Dies kann jedoch auch vollständig ohne zugehörige Hardware geschehen, indem der Ist-Zustand des Würfels per Hand in die Software übertragen wird und diese anschließend den Lösungsweg ausgibt. Dafür muss es Anwender:innen möglich sein, den Würfelzustand beispielsweise in ein passendes User Interface einzugeben. Eine geeignete und nachvollziehbare Darstellung des Zauberwürfels ist somit unabdinglich.

4.1.2 Hardware

Eine ergänzende Vorrichtung, in die ein verdrehter Würfel eingelegt werden kann und anschließend gelöst wird, ist erstrebenswert, für dieses Vorhaben jedoch erst einmal zweitrangig. Für die Umsetzung bedarf es vor allem einer Vorrichtung, welche in der Lage ist, die Seiten des Zauberwürfels eigenständig zu drehen, und eines Sensors, welcher den aktuellen Zustand des Würfels vorher aufnimmt. Die Software müsste entsprechend ergänzt werden, um dies zu ermöglichen.

4.2 Stakeholder

Um die Anforderungen des Systems auszuarbeiten, gilt es im Vorfeld die Interessen der Stakeholder zu untersuchen. Diese formen das anschließende Endprodukt dieser Arbeit, indem versucht wird, den Ansprüchen bestmöglich gerecht zu werden. Dafür muss zunächst analysiert werden, welche Personengruppen überhaupt betroffen sind und welchen Nutzen diese von der Fertigstellung dieses Projektes haben.

Anwender:in

Eine wichtige Personengruppe, welche bei der Entwicklung dieses Produktes berücksichtigt werden soll, sind der/die Anwender:innen. Die Software sollte in der Lage sein, den Zauberwürfel zuverlässig zu lösen und bestenfalls den optimalen Lösungsweg zu ermitteln. Dies ist die Grundlage dafür, dass die Software genutzt werden kann, um beispielsweise Personen weiterzuhelfen, die gerade versuchen, die Lösungsalgorithmen für die Zauberwürfel zu erlernen. Sie könnte allerdings auch Grundlage bilden, um sogenannten Speedcubern die Möglichkeit zu bieten, sich zu messen oder möglicherweise neue Algorithmen zu erlernen, die bis dato noch nicht in ihrem Repertoire waren.

Weiterbildung im Bereich künstliche Intelligenz

Für Personen, welche sich im Bereich künstliche Intelligenz und deren Anwendungsbereich weiterbilden möchten, soll diese Arbeit eine ausführliche Einführung in die Thematik sein. Dies betrifft beispielsweise Student:innen, welche diese Arbeit weiterentwi-

ckeln oder aber auch an ähnlichen Projekten arbeiten wollen. Deshalb ist ein detaillierter Grundlagenteil sowie Dokumentation über die entwickelte Software unentbehrlich.

Auftraggeber und Prüfer

Ein weiterer wichtiger Stakeholder ist Auftraggeber und Erstprüfer Prof. Dr. Hensel. Es ist sein eigenes Anliegen, dass am Ende dieses Projektes eine funktionsfähige Software entwickelt wurde, mit Hilfe derer ein verdrehter Zauberwürfel gelöst werden kann. Wünschenswert wäre hierbei auch ein Demonstrator, in welchen ein passender Würfel eingelegt, dessen Zustand erkannt und dieser anschließend automatisch in den Lösungszustand gebracht wird. Zudem besitzt Prof. Dr. Hensel selbst ein großes Interesse an den Methodiken des Reinforcement Learnings, weshalb ein detaillierter Grundlagenteil auch zu seinem Nutze sein könnte. Für nähere Ausarbeitungsdetails erfolgten regelmäßig Absprachen zwischen dem Ersteller dieser Arbeit und dessen Erstprüfer.

Ersteller dieser Arbeit

Für den Ersteller dieser Arbeit steht die persönliche Weiterentwicklung im Bereich künstlicher Intelligenz im Vordergrund. Eine große Ansammlung von Grundlagenwissen und Erlernen neuer Kompetenzen ist in seinem Interesse. Sein Ziel ist allerdings, das bestenfalls am Ende dieser Arbeit ein vorzeigbarer Prototyp entstanden ist, welcher von Student:innen genutzt werden kann, um an diesem oder ähnlichen Projekten weiterzuarbeiten.

4.3 Anwendungsfälle

In diesem Abschnitt soll kurz auf die einzelnen Anwendungsfälle des zu entwerfenden System eingegangen werden. Dafür wurde das in Abbildung 4.1 dargestellte Anwendungsfalldiagramm entworfen, welches eine Übersicht über die einzelnen Komponenten bieten soll.

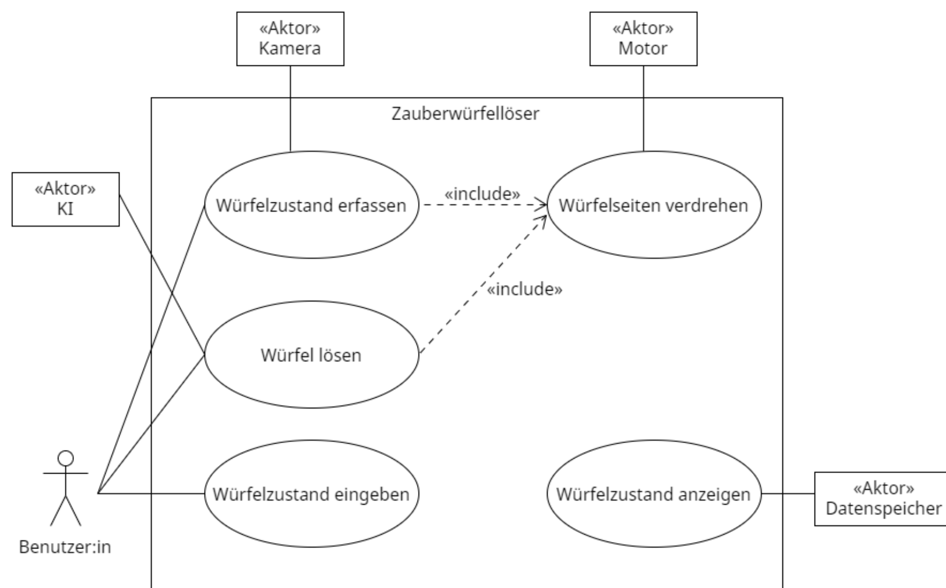


Abbildung 4.1: Anwendungsfalldiagramm des Zauberwürfellösers.

Tabelle 4.1: Anwendungsfall: Würfelzustand erfassen

Name	Würfelzustand erfassen
Kurzbeschreibung	Durch Rotation des Würfels, werden nacheinander alle Seiten des Würfels aufgenommen.
Aktoren	Kamera, Anwender:in
Vorbedingung	Ein Würfel wurde in den Demonstrator eingelegt.
Ergebnis	Würfelzustand in passendem Datenformat (beispielsweise Array).
Beschreibung des Ablaufs	1. Einlegen des Zauberwürfels. 2. Bildaufnahme einer oder mehrerer Seiten. 3. Rotation des Zauberwürfels. 4. Wiederholung der Prozesse, bis der gesamte Würfelzustand erfasst wurde.

Tabelle 4.2: Anwendungsfall: Würfelzustand lösen

Name	Würfel lösen
Kurzbeschreibung	Der Lösungsweg für den aktuellen Zustand wird mittels neuronalem Netzwerk ermittelt.
Aktoren	Künstliche Intelligenz, Motoren
Vorbedingung	Der Würfelzustand wurde zuvor von den entsprechenden Sensoren erfasst oder händisch in die Software eingetragen und wurde in ein für das neuronale Netz verständliches Format transformiert.
Ergebnis	Gelöster Würfel.
Beschreibung des Ablaufs	1. Berechnen des Lösungsweges. 2. Entsprechendes Rotieren der Würfelseiten.

Tabelle 4.3: Anwendungsfall: Würfelzustand anzeigen

Name	Würfelzustand anzeigen
Kurzbeschreibung	Darstellen des aktuellen Würfelzustandes.
Aktoren	Datenspeicher
Vorbedingung	Der Würfelzustand wurde zuvor von den entsprechenden Sensoren erfasst oder händisch in die Software eingetragen.
Ergebnis	2D- oder 3D-Darstellung des aktuellen Zustandes.
Beschreibung des Ablaufs	Umwandeln der gespeicherten Daten in entsprechende Bilddaten.

Tabelle 4.4: Anwendungsfall: Würfelzustand eingeben

Name	Würfelzustand eingeben
Kurzbeschreibung	Eingabe des aktuellen Würfelzustandes in die Software.
Aktoren	Benutzer:in
Vorbedingung	-
Ergebnis	Würfelzustand in passendem Datenformat (beispielsweise Array).
Beschreibung des Ablaufs	Mit Hilfe passender Eingabemöglichkeit überträgt der/die Anwender:in den aktuellen Zustand in die Software.

Tabelle 4.5: Anwendungsfall: Würfelseiten verdrehen

Name	Würfelseiten verdrehen
Kurzbeschreibung	Rotieren der Würfelseiten mittels Motoren.
Aktoren	Motoren
Vorbedingung	Ein Würfel wurde in den Demonstrator eingelegt.
Ergebnis	Zauberwürfel, bei welchem eine oder mehrere Seiten rotiert wurden.
Beschreibung des Ablaufs	Mit Hilfe von Motoren und entsprechenden Konstruktionen werden die Seiten des eingelegten Würfels rotiert.

4.4 Anforderungen

Durch das Analysieren der Interessen von Stakeholdern und das Ermitteln der Anwendungsfälle lassen sich Anforderungen für die zu entwickelnde Einheit formulieren. Hierbei werden funktionale Anforderungen mit (F) und nicht funktionale Anforderungen mit (NF) gekennzeichnet. Aufgrund des zeitlichen Rahmens ist es sinnvoll, den einzelnen An-

forderungen Prioritäten zuzuweisen, falls nicht alle umgesetzt werden können. Solche mit einer hohen Priorität werden hierbei als essentiell betrachtet und sind für das Fertigstellen des Projektes erforderlich. Anforderungen, welche in der Priorität niedrig angesiedelt sind, sind zunächst vernachlässigbar und stellen eine wünschenswerte Erweiterung des Systems dar. Funktional ist die Hardware nicht zwingend erforderlich und wird deshalb separat nach den Ansprüchen an den Softwareteil aufgeführt.

4.4.1 Software

In diesem Abschnitt sollen die Anforderungen für die zu entwickelnde Software näher erläutert werden. In Tabelle 4.6 sind die funktionalen und in Tabelle 4.7 die nicht funktionalen Anforderungen aufgeführt und der Priorität nach sortiert aufzufinden.

Tabelle 4.6: Funktionale Anforderungen an die Software.

ID	**Priorität**	**Anforderung**
S-F1	Hoch	Berechnung des Lösungsweges mittels KI
S-F2	Hoch	Eingabe des Würfelzustandes in einer benutzerbasierten Anwendung
S-F3	Mittel	Anzeige des aktuellen Zustandes in 2D
S-F4	Niedrig	Anzeige des aktuellen Zustandes in 3D
S-F5	Hoch	Erkennen, ob aktueller Zustand der Lösungszustand ist
S-F6	Hoch	Erzeugen zufällig verdrehter Würfelzustände für das Training des neuronalen Netzes
S-F7	Hoch	Automatisches Abbrechen nach einer eingestellten Anzahl von Berechnungsschritten
S-F8	Niedrig	Finden eines optimalen Lösungsweges (≤ 20, *Gottes Zahl*)

Da in dieser Arbeit Zauberwürfel mit Hilfe von künstlicher Intelligenz gelöst werden sollen, ist S-F1 obligatorisch. Um diese Eigenschaft überhaupt zu ermöglichen, muss der Software der Zustand des Würfels übermittelt werden, was in einfachster Form durch händisches Übertragen geschehen kann. Deshalb ist auch S-F2 unerlässlich. Eine geeignete Darstellung des Würfelzustandes in einer 2D-Darstellung (S-F3) wäre hilfreich, ist aus funktionaler Sicht nicht zwingend notwendig. Umso weniger erforderlich ist eine Darstellung in 3D (S-F4), könnte aber dennoch als eine gedankliche Stütze beim Arbeiten mit

der Software dienen. Dass das neuronale Netz erkennen muss, wann der Lösungszustand erreicht wurde (S-F5), ist selbsterklärend. Für das Training der künstlichen Intelligenz ist eine Vielzahl von verschiedenen Würfelzuständen erforderlich, weshalb es zweckmäßig ist, dass die zu entwerfende Einheit in der Lage ist, eigens zufällig verdrehte Würfelzustände zu erzeugen (S-F6), mit welchen das Netzwerk angelernt werden kann. Ein automatisches Abbrechen der Lösungszustandsfindung (S-F7) ist notwendig, damit die KI nicht in eine Endlosschleife gerät und das Programm unaufhörlich weiterläuft. Die Bestimmung des optimalen und somit kürzesten Lösungsweges (S-F8) wird als Anforderung erst einmal hinten angestellt. Dieser Anspruch ist eine wünschenswerte Erweiterung, allerdings für ein lauffähiges System nicht zwingend erforderlich.

Tabelle 4.7: Nicht funktionale Anforderungen an die Software.

ID	**Priorität**	**Anforderung**
S-NF1	Hoch	Maximale Rechenzeit von 5 Minuten
S-NF2	Hoch	Software ist in der Lage mindestens 90 % der Würfel zu lösen
S-NF3	Hoch	Redundanz in Würfelzustandsdarstellung vermeiden [18]
S-NF4	Hoch	Effiziente Speicherung von Würfelzuständen [18]
S-NF5	Hoch	Perfomante Implementation von Würfeltransformationen [18]
S-NF6	Hoch	Geeignete Zustandsrepräsentation für ein neuronales Netz [18]
S-NF7	Niedrig	Eingängige Bedienoberfläche für die Verwendung der Software (GUI)

Die nicht funktionalen Ansprüche fallen deutlich geringer aus. So wäre es wünschenswert, dass das fertige System schlussendlich in der Lage ist, 90 % der ihm zugeführten Würfel zu lösen (S-NF1) und dies innerhalb einer Zeitspanne von maximal 5 Minuten (S-NF2). Diese Werte wurden basierend auf dem Buch von Max Lapan [18] ausgewählt. In Abbildung 4.2 ist zu erkennen, dass das von Lapan verwendete und modifizierte *zero-goal*-Modell in der Lage ist, den Würfel immer mit einer Quote von mindestens 90% erfolgreich zu lösen. Die maximale Rechenzeit ist aus der Zeit abgeleitet, die Lapan für die Lösung des Rubik's Cube verwendet. Dort sind maximal 10 Minuten Rechenzeit pro neuem Würfel, den es zu lösen gilt, eingeplant. Da der Pocketcube im Gegensatz zu dem größeren Zauberwürfel verhältnismäßig einfach ist, wurde die maximale Zeit für diesen auf 5 Mi-

nuten halbiert. Die nachfolgenden Ansprüche (S-NF3 bis S-NF6) sind dem Buch „Deep Reinforcement Learning Hands-On“ [18] von Max Lapan entnommen, welche er dort als zwingend notwendig aufführt. Die Redundanz bei der Zustandsdarstellung des Würfels zu vermeiden (S-NF3), ist deshalb von Wichtigkeit, da der Zustandsraum des Rubik's Cube ohnehin schon gigantisch groß ist und sich durch eine redundante Repräsentation nochmal um ein Vielfaches vergrößert. Auch eine speichereffiziente (kompakte) Implementation von Zuständen ist erforderlich, da bei dem Training und auch der Anwendung des neuronalen Netzwerkes eine Vielzahl verschiedener Zustände im Arbeitsspeicher abgelegt werden und die Geschwindigkeit des Vorgangs beeinflussen. Ein weiterer wichtiger Anspruch an die Implementierung ist die Perfomance der Würfeltransformationen (S-NF4). Damit ist die Umsetzung der Übergänge von einem Würfelzustand in einen Folgezustand gemeint, welche durch die Ausführung einer möglichen Aktion auf den aktuellen Zustand geschieht. Lapan führt hierbei aus, dass eine kompakte Repräsentation alleine nicht ausreicht, wenn diese bei jeder Drehung einer Seite langwierig entpackt werden muss und somit das Training des Netzes wieder verlangsamt. Zuletzt ist eine weiterere unverzichtbare Anforderung, dass die gewählte Darstellung verständlich für das neuronale Netz ist (S-NF6), welches nicht nur für dieses Projekt, sondern für Machine Learning generell gilt [18]. Für eine einfache Handhabung ist eine eingängige Bedienoberfläche (S-NF7) erstrebenswert, aber zunächst entbehrlich. Nachdem die Software fertiggestellt wurde, wird diese dem Erstprüfer Prof. Dr. Hensel übergeben, welcher die Bedienoberfläche eingehend auf Intuitivität testet und bewertet.

4.4.2 Hardware

Diese Sektion beschreibt die Ansprüche an eine mögliche Hardwareumsetzung für einen Demonstrator, welcher in Kombination mit der Software in der Lage wäre, einen Zauberwürfel zu lösen. Solch ein Demonstrator wird zunächst als wünschenswert betrachtet und dient nicht der Zielerfüllung dieser Arbeit. Einen realen Zauberwürfel in solch eine Maschine einzulegen und gelöst zu erhalten, ist dennoch in Anbetracht von Weiterentwicklungsmöglichkeiten dieser Arbeit eine sinnvolle Anforderung, die allerdings hinter der Entwicklung der Software ansteht.

Solch ein Demonstrator umfasst lediglich zwei funktionale Kriterien: Das Erfassen des Würfelzustandes (H-F1) und anschließendes Verdrehen der Würfelseiten (zum Lösungszustand) (H-F2).

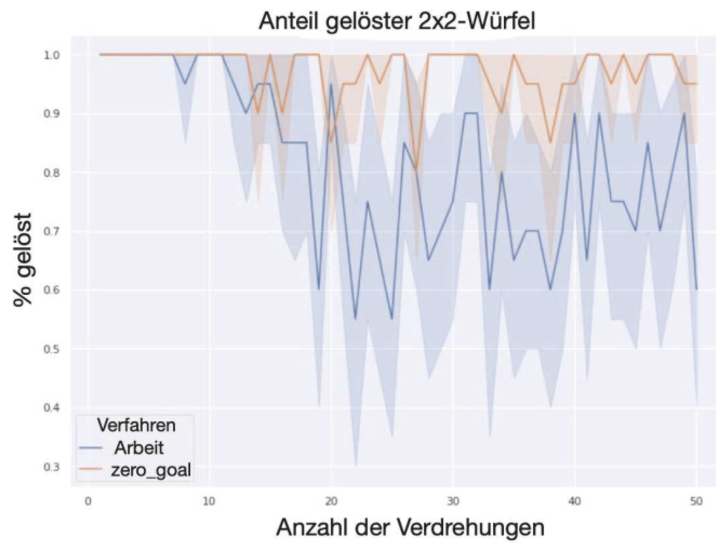


Abbildung 4.2: Anteil der gelösten 2x2x2-Würfel in der Ausarbeitung von Max Lapan [18].

Tabelle 4.8: Funktionale Anforderungen an die Hardware.

ID	Priorität	Anforderung
H-F1	Hoch	Erfassen des aktuellen Würfelzustandes
H-F2	Hoch	Rotieren der Würfelseiten mittels Motoren

5 Konzept

Bevor das System tatsächlich entwickelt wird, gilt es, dieses zunächst konzeptionell auszuarbeiten und die näheren Details zu klären. Die Ansprüche an dieses stehen durch das vorherige Kapitel fest, allerdings fehlt noch, wie diese in die Tat umgesetzt werden sollen. Deshalb werden in diesem Abschnitt Möglichkeiten zur Realisierung der einzelnen Komponenten diskutiert. Dazu gehören vor allem Themen, wie welche neuronalen Netze für diese Anwendung geeignet sind oder wie ein Hardwareaufbau gestaltet werden könnte. Des Weiteren sollen mögliche Schwierigkeiten bei der Umsetzung rechtzeitig erkannt werden, sodass diese nicht erst bei der konkreten Entwicklung auftreten. Durch die Komplexität der Aufgabe erscheint es außerdem als äußerst sinnvoll, Vergleichsarbeiten zu dieser Thematik näher zu untersuchen und Herangehensweisen und konzeptionelle Methodiken zu übernehmen. Im Abschnitt 3.1.2 auf Seite 67 wurden bereits erfolgreiche Werke zu diesem Thema vorgestellt, welche als Grundgerüst für dieses Projekt dienen sollen. Dabei stechen besonders die beiden Arbeiten derselben Forschungsgruppe hervor, welche aus Forest Agostinelli, Stephen McAleer, Alexander Shmakov und Pierre Baldi besteht. In diesen Arbeiten [21, 22] wurde jeweils eine funktionsfähige Software entwickelt, welche in der Lage ist, einen Rubik's Cube zu lösen. Die beiden Arbeiten unterscheiden sich jedoch in dem Aufbau der verwendeten neuronalen Netze und zusätzlichen Algorithmen, welche scheinbar notwendig sind, um zuverlässig zum Lösungszustand zu gelangen. Auf die erste Veröffentlichung aufbauend hat Max Lapan sich in seinem Buch „Deep Reinforcement Learning Hands-On“ [18] näher mit dieser auseinandergesetzt und versucht die dort erläuterte Methode nachzuimplementieren. Den zugehörigen Programmcode stellt Lapan auf seiner GitHub-Seite[1] zur Verfügung. Aufgrund der somit ausreichend zur Verfügung stehenden Dokumentation zur ersten Arbeit der Forschungsgruppe bietet es sich an, diese zunächst als Grundlage heranzuziehen, um einen Einstieg in die Komplexität dieser Thematik zu bekommen. Wird dies erfolgreich umgesetzt, soll auch versucht werden, den zweiten Ansatz der Forschergruppe um McAleer zu verwirklichen.

[1] https://github.com/Shmuma/rl/tree/master/articles/01_rubic
- Zugriffsdatum: 07.07.23

5.1 Software

Da das Hauptaugenmerk dieser Thesis auf dem Softwareteil liegt, wird zunächst das zugehörige Konzept, welches sich aus den Anforderungen aus Abschnitt 4.4.1 ergibt, ausgearbeitet. Dafür gilt es, zuerst eine Entwicklungsumgebung festzulegen, in der die Anwendung entworfen werden soll. Der Kern ist hierbei die Arbeit mit neuronalen Netzen, weshalb primäres Kriterium ein einfacher Einstieg beziehungsweise gute Verständlichkeit der gewählten Progammiersprache im Bereich Machine Learning sein wird.

5.1.1 Entwicklungsumgebung

Für das in diesem Projekt geplante Vorhaben bieten sich einige Programmiersprachen für die Umsetzung an. Idealerweise sollte sich die ausgewählte Codiersprache im Repertoire des Erstellers dieser Arbeit befinden, doch können Kompromisse eingegangen werden, findet sich eine Sprache mit einfacher Erlernbarkeit. Anbieten würden sich deshalb Java, Matlab, C oder C++. Der PYPL-Index (PopularitY of Programming Language), welcher angibt, wie oft eine Programmiersprache in der Suchmaschiene „Google“ eingegeben wurde, zeigt hingegen einen eindeutigen Trend in Richtung der Codiersprache Python. Im Jahre 2018 hat Python Java als populärste Progammiersprache abgelöst und befindet sich seitdem auf dem ersten Platz und besitzt mittlerweile einen Anteil von mehr als einem Viertel (28%) [5]. Der Grund dafür liegt vermutlich in der einfachen und übersichtlichen Syntax und der Vielzahl an kostenfreien Bibliotheken und Tools, die den Nutzer:innen zur Verfügung stehen. Des Weiteren ist Python dafür bekannt, dass es leicht zu erlernen und dennoch in der Lage ist, die Leistungsfähigkeit der Systemebene zu nutzen [28]. Aufgrund dessen wird der gesamte Softwareteil in der Codiersprache Python mit den entsprechenden benötigten Bibliotheken implementiert, welche in den nachfolgenden Abschnitten unter anderem erörtert werden sollen.

Machine Learning Framework

Ein weiteres Kriterium für die Auswahl von Python sind die zur Verfügung stehenden Frameworks im Bereich des Deep Learning. Denn dort stehen die zur Zeit beliebtesten und am meisten genutzten Frameworks Pytorch und Tensorflow zur Verfügung. Seit 2017 gilt Pytorch als die populärste Deep Learning Bibliothek in der Forschung [28]. Und in 2019 war Pytorch die meistbenutzte Deep Learning Bibliothek auf allen großen Deep Learning

Konferenzen [28, 11]. Tensorflow besitzt dafür laut Quelle [28] eine noch immer größere Beliebtheit im Industriebereich. Da sich zunächst in den Bereich des Deep Learnings eingearbeitet werden soll und Pytorch einen starken Fokus auf Benutzerfreundlichkeit legt [28] und zudem einfacher für Neueinsteiger zu verstehen sein soll [8], wird dieses Framework auch in dieser Arbeit eingesetzt werden.

5.1.2 Trainingsumgebung (Simulation)

Um das neuronale Netzwerk zu trainieren, müssen diesem Daten zugeführt werden, aus welchen das Netz „Schlüsse" ziehen kann. In diesem Fall sind das Würfelzustände und mögliche zusätzliche Informationen, wie beispielsweise, ob der aktuelle Zustand der Lösungszustand ist oder wie weit der aktuelle Zustand von diesem entfernt ist. Diese Daten müssen allerdings vor oder während des Trainings zuerst generiert werden. In Ivan Gridins Buch „Practical Deep Reinforcement Learning with Python" [8] werden neuronale Netze in bekannten Spielklassikern trainiert, welche in einer kostenfreien Bibliothek namens *OpenAI Gym* zu Verfügung gestellt werden. Die in der Bibliothek angebotenen Spieleumgebungen sind für das Training von KIs ausgelegt und besitzen immer denselben Aufbau. Primär kommt dort eine Funktion zum Einsatz, mit welcher der nächste Schritt abhängig von einer ausgewählten Aktion durchgeführt werden kann und welche anschließend den Folgezustand, den zugehörigen Reward und die Information, ob das Spiel abgeschlossen wurde, ausgibt. Dies ermöglicht eine einfache Arbeit mit den angebotenen Spielen und erfordert kein Einarbeiten in einen Programmcode. Um eine Weiternutzung zu vereinfachen und für Thesen, welche auf dieser aufbauen könnten, soll auch im Rahmen dieser Arbeit eine Umgebung des Zauberwürfels mittels OpenAI Gym entworfen werden.

5.1.3 Zustandsrepräsentation

Ein wichtiger Aspekt, den es im Vorfeld festzulegen gilt, ist die Art und Weise, wie die Würfelzustände im Programmcode umgesetzt werden, sodass diese den Anforderungen S-NF3 (Redundanz in Würfelzustandsdarstellung vermeiden), S-NF4 (Effiziente Speicherung von Würfelzuständen), S-NF5 (Perfomante Implementation von Würfeltransformationen) und S-NF6 (Geeignete Zustandsrepräsentation für ein neuronales Netz) gerecht werden. Der Zustand lässt sich auf unterschiedliche Arten repräsentieren, von denen die interessantesten kurz erläutert und in Bezug auf Umsetzbarkeit miteinander verglichen werden sollen.

Facelet Repräsentation

Die zunächst offensichtlichste und eingängigste Art, den Würfelzustand darzustellen, ist die über seine farbigen Sticker. Mit Hilfe eines 6x3x3-Arrays könnte der Zustand über Zahlenwerte (Integer) oder Buchstaben (Char) wiedergegeben werden (vgl. Listing 5.1). Der große Vorteil hierbei ist die leichte Nachvollziehbarkeit für Nutzer:innen. Durch eine One-Hot-Kodierung [15] lässt sich der Würfelzustand auch in ein für das neuronale Netz verständliches Format übertragen. Allerdings treten große Schwierigkeiten bei der Erfüllung der Anforderungen an die Redundanz und die Speichereffizienz auf. Durch diese Art der Darstellung wäre es theoretisch möglich, dass alle Facelets bis auf die Mittelsteine dieselbe Farbe besitzen [18]. Somit ergibt sich eine neue Zustandsraumgröße: $6^{6 \cdot 8} \approx 2.25 \cdot 10^{37}$ (48 Sticker, welche sechs Farben annehmen können). Wird von vier Byte pro Integer beziehungsweise Char ausgegangen, so ergibt sich für 54 Sticker eine Größe von: $4\,\text{Bytes} \cdot 54 = 216$ Bytes. Für die Berechnung des Speicherplatzes werden 54 statt der 48 angegebenen Sticker berücksichtigt, da der Mittelstein sich zwar nicht ändert, aber trotzdem in einem 6x3x3-Array mitaufgeführt werden würde. Jedoch werden hier die zuätzlich benötigten Speicheradressen nicht berücktsichtigt, die für die Repräsentation über ein dreidimensionales Array notwendig wären. Ein weiterer Aspekt, der gegen diese Darstellungsart spricht, ist, dass die Transformation der Zustände nicht qualitativ implementiert werden kann. Dies liegt vor allem daran, dass zuerst die Cubies beziehungsweise Facelets, welche von der Aktion bewegt werden, ermittelt und anschließend die entsprechenden Felder des Arrays aufwendig angepasst werden müssen.

Ein wenig Besserung könnte die Implementation über ein eindimensionales Array mit 48 Einträgen bewirken, statt eines 6x3x3-Arrays. Durch weniger Dimensionen sind weniger Speicheradressen notwendig, wodurch die Speichereffizienz verbessert wird. Des Weiteren lassen sich die Transformationen einfacher implementieren, da es die Notwendigkeit vermeidet, sich mit mehrdimensionalen Arrays oder komplexen Datenstrukturen befassen zu müssen. Der Zauberwürfel ist ein komplexes mathematisches Objekt, bei dem die Beziehung seiner Komponenten im Vordergrund steht. Durch die Zustandsrepräsentation über die farbigen Facelets geht diese Beziehung nicht eindeutig hervor, weshalb in dieser Arbeit ein anderer Ansatz verwirklicht werden soll.

```
1                ['w' 'w' 'w']
2                ['w' 'w' 'w']
3                ['w' 'w' 'w']
4  ['o' 'o' 'o'] ['g' 'g' 'g'] ['r' 'r' 'r'] ['b' 'b' 'b']
5  ['o' 'o' 'o'] ['g' 'g' 'g'] ['r' 'r' 'r'] ['b' 'b' 'b']
6  ['o' 'o' 'o'] ['g' 'g' 'g'] ['r' 'r' 'r'] ['b' 'b' 'b']
7                ['y' 'y' 'y']
8                ['y' 'y' 'y']
9                ['y' 'y' 'y']
```

Listing 5.1: Würfelzustandsrepräsentation mittels 6x3x3 Char-Array.

Cubie Repräsentation

Die Würfelzustandsdarstellung über die einzelnen Cubies, aus denen der Zauberwürfel besteht, ist die Variante, welche Max Lapan [18] in seinem Buch gewählt hat. Hierbei werden nicht die farbigen Sticker berücksichtigt, sondern die Position der einzelnen Würfelsteine und deren Orientierung. Aus dem Grundlagenteil 3.1 ist bekannt, dass der Rubik's Cube aus zwölf Kantensteinen, welche zwei Orientierungen besitzen, und den acht Ecksteinen mit drei Anordungsmöglichkeiten besteht. Dies lässt sich über vier Arrays, Listen oder Tuple in Python implementieren (vgl. Listing 5.2) und verringert die benötigte Datenmenge auf 160 Bytes pro Zustand. Der große Vorteil dieser Repräsentation ist, dass sich die Beziehung der Cubies untereinander basierend auf deren Position und Orientierung einfacher umsetzen lässt, was die Manipulation des Zustandes erleichtert. Es können direkt Ort und Orientierung einzelner Cubies verändert werden, um die gewünschte Rotation zu erwirken. Bei der Drehung einer Würfelseite können einfach die Positionen der entsprechenden Cubies getauscht werden, während die Beziehungen beibehalten werden.

```
1  State = collections.namedtuple("State", field_names=['corner_pos',
2                                                        'side_pos',
3                                                        'corner_ort',
4                                                        'side_ort'])
5  ...
6  initial_state = State(corner_pos= (0, 1, 2, 3, 4, 5, 6, 7),
7                        side_pos   = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11),
8                        corner_ort= (0, 0, 0, 0, 0, 0, 0, 0),
9                        side_ort   = (0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0))
```

Listing 5.2: Kompakte Darstellung des Urzustandes des Zauberwürfels [18].

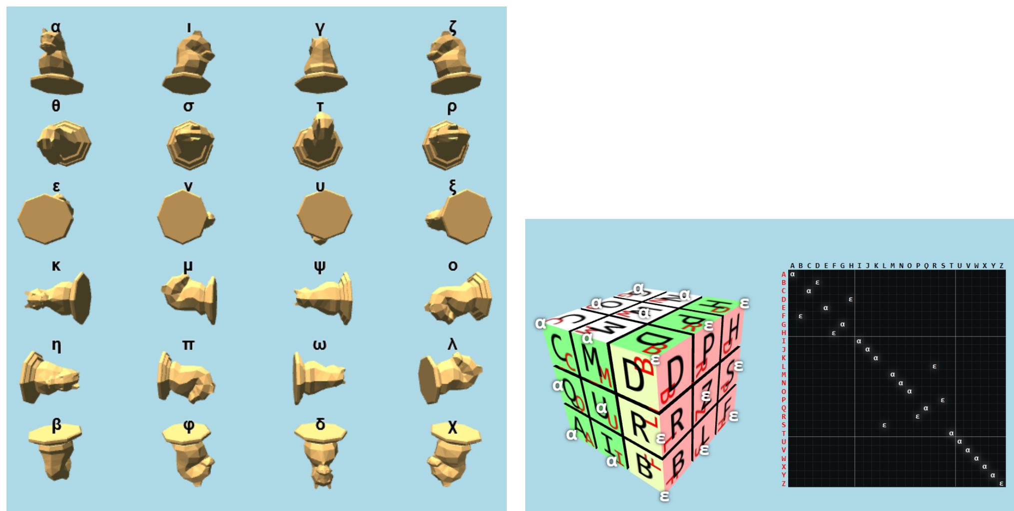
Mit Hilfe von One-Hot-Kodierung kann auch diese Darstellung wieder für ein neuronales Netz in eine verständliche Form gebracht werden. Dadurch ergibt sich ein 20x24

Tensor, welcher einen größeren Speicherplatzbedarf ($4\,\text{Bytes} \cdot 20 \cdot 24 = 1920$ Bytes pro Zustand) besitzt als die kompakte Repräsentation, aus welcher dieser hervorgeht. Der Tensor dient jedoch nur als Netzwerkeingabe und die kompakte Darstellung wird für längere Speicherung der Zustände im Arbeitsspeicher verwendet, wodurch die Effizienz wieder gewährleistet werden kann. Anzumerken ist, dass auch die Zustandsdarstellung mit einem Wert von $24^{20} = 4,02 \cdot 10^{27}$ möglichen Kombinantionen noch immer redundant ist. Dies liegt vor allem an den besonderen Eigenschaften der Würfeltransformationen, da nicht nur ein Stein allein bewegt werden kann, ohne dass alle anderen unberührt bleiben [18]. Für die Erfüllung aller an die Zustandsraumdarstellung gestellten Anforderungen (S-NF3 bis S-NF6) wird auch hier die Cubie Repräsentation verwendet, dem Vorbild von Lapans Implementation entsprechend. Dennoch gibt es weitere interessante Varianten den Zustand des Rubik's Cube abzubilden.

Repräsentation in berechenbarer Form

Auf der Website[1] verfolgt der Autor einen anderen Ansatz, um den Zustand eines Zauberwürfels auszudrücken. Ähnlich wie in der Darstellung zuvor werden auch hier die Positionen der einzelnen Cubies getrackt. Statt der Orientierungen im Bezug zu den Cubicles werden hier die orthogonalen Anordungsmöglichkeiten (im 3D-Raum sind dies $6 \cdot 4 = 24$) berücksichtigt (vgl. Abbildung 5.1). Mit Hilfe von Würfelpermutationen, welche in einem 26x26 Array festgehalten werden, verändert sich die orthogonale Anordnung der Cubies. Über das Wissen, welche Rotation welche Anordnung hervorruft mit Bezug auf die aktuelle Orientierung, kann durch eine einfache Multiplikationstabelle ermittelt werden, welche Aktion die betroffenen Cubies in welche orthogonale Orientierung versetzt. Somit kann nach und nach der Lösungszustand des Würfels hergestellt werden. Dafür ist es allerdings notwendig zu wissen, welcher Prozess zum aktuellen Zustand geführt hat. Andernfalls ist die aktuelle Orientierung der Cubies nicht bekannt und es kann nicht berechnet werden, welche Transformationen nötig sind, um den Lösungszustand wiederherzustellen. Es wäre demnach zwar möglich, ein neuronales Netz mittels entsprechend umgesetztem Tensor (beispielsweise wieder mittels One-Hot-Kodierung) zu trainieren, allerdings fehlt anschließend die Möglichkeit, diesem neue unbekannte Zustände zum Lösen zu übergeben, da die Aktionen, die dorthin geführt haben, nicht bekannt sind. Die Berechenbarkeit der Würfelzustände macht diese Art der Darstellung aus mathematischer Sicht sehr interessant, ist jedoch für das Ziel dieser Arbeit nicht praktikabel.

[1] https://k-l-lambda.github.io/2020/12/14/rubik-cube-notation/ - Zugriffsdatum: 07.07.23



(a) Orthogonale Anordnungsmöglichkeiten im 3D-Raum.

(b) Rubik's Cube in berechenbarer Form nach der Transformation R.

Abbildung 5.1: Rubik's Cube in berechenbarer Form. Quelle[1].

Hyperwürfel Repräsentation

Der Zustand des Rubik's Cube kann auch als Hyperwürfel[2] präsentiert werden. Dieser verallgemeinert einen Würfel auf höhere Dimensionen, sodass ein dreidimensionaler Zauberwürfel beispielsweise als vier- oder höherdimensionaler Hyperwürfel dargestellt werden kann (vgl. Abbildung 5.2). Dies kann die Visualisierung des Zustandes vereinfachen und das Verständnis der Struktur und der Beziehungen zwischen den Cubies fördern. Der Zustand könnte somit zum Beispiel über vier Koordinanten für die Position (x, y, z, w) und zwei Variablen für die Orientierung (u,v) abgebildet werden. Bei dieser Repräsentationsweise treten ähnliche Schwierigkeiten auf wie bei der über ein 6x3x3-Array mit den farbigen Stickern des Würfels. Die Arbeit mit höherdimensionalen Darstellungsarten kann wesentlich komplexer sein und erschwert eine effiziente Speicherung und performante Würfeltransformationen, weshalb diese hier nicht weiter berücksichtigt werden.

[1] https://k-l-lambda.github.io/2020/12/14/rubik-cube-notation/ - Zugriffsdatum: 07.07.23
[2] https://mathworld.wolfram.com/HypercubeGraph.html - Zugriffsdatum: 07.07.23

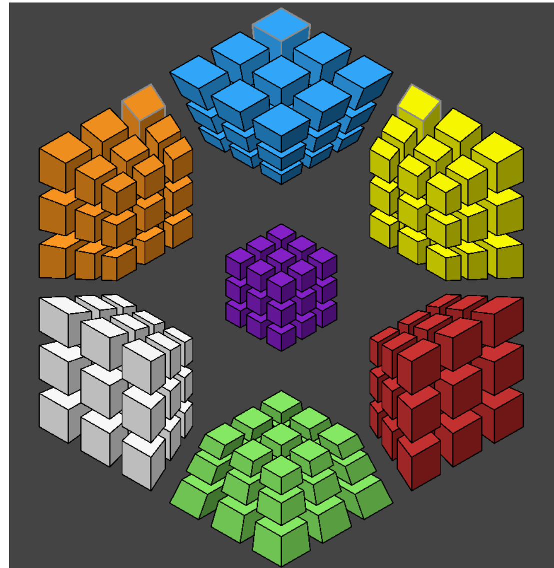


Abbildung 5.2: Darstellung des Zauberwürfels in 4D. Quelle[1].

Vergleich Cubie und Facelet Repräsentation

Ein kurzer Überblick zwischen den Größenunterschieden der beiden relevanten Darstellungsarten ist der Tabelle 5.1 zu entnehmen.

Tabelle 5.1: Größenvergleich der Zustandsräume des Zauberwürfels nach Darstellungsart

Darstellungsart	Größe Zustandsraum
Zauberwürfel	$4,33 \cdot 10^{19}$
Sticker-Repräsentation	$2,5 \cdot 10^{37}$
Cubelet-Repräsentation	$4,02 \cdot 10^{27}$

Darstellung

Umgebungen in OpenAI Gym besitzen des Weiteren oftmals eine Render-Funktion, mit Hilfe welcher sich der aktuelle Spielzustand grafisch ausgegeben lässt. Dafür kommt üblicherweise die Bibliothek *Pygame* zum Einsatz, mit derer sich einfach simple Grafiken

[1] https://hypercubing.xyz/hyperspeedcube/ - Zugriffsdatum: 07.07.23

erzeugen lassen. Im Gegensatz zu dem in Ivan Gridins Buch „Practical Deep Reinforcement Learning with Python“ [8] vorgenommen Training mittels Bilddaten als Eingabe, ist diese Art des Trainings hier nicht unbedingt geeignet. Denn in diesem Fall reicht es aus, wenn der Würfelzustand über Python Datenstrukturen und mittels entsprechender Zahlenwerte oder Ziffern dargestellt wird. Dafür sind weder Kameraaufnahmen noch erzeugte Grafiken über Pygame notwendig. Nichtsdestotrotz soll ein Klassenobjekt des Würfels in OpenAI Gym erzeugt werden, welches eine 2D- oder unter Umständen sogar 3D-Darstellung der Würfelzustände ermöglichen soll. Dies liegt darin begründet, dass es Anwender:innen einfacher fallen wird, den Geschehnissen zu folgen, werden diese visuell zur Verfügung gestellt. Für die Darstellung soll auch hier die angesprochene Python Bibliothek Pygame verwendet, welche durch ihre Einfachheit und die ausgiebige Dokumentation auf der zugehörigen Website[1] hervorsticht.

5.1.4 Neuronales Netz

Die richtige Auswahl der Art des neuronalen Netzes ist ausschlaggebend für den Trainingserfolg eines Projektes mit künstlicher Intelligenz. In dem Grundlagenteil 2.2.1 auf Seite 21 sind verschiedene Typen von Netzwerken vorgestellt worden, aus welchen es das für diese Anwendung passende auszuwählen gilt. Es stehen bei Weitem mehr mögliche Architekturarten zur Auswahl, dennoch soll sich hier auf die grundlegenden beschränkt werden, deren Einsatzgebiete noch einmal knapp in Tabelle 5.2 festgehalten werden.

Tabelle 5.2: Übersicht über die neuronalen Netzwerkarten aus Abschnitt 2.2.1.

Art	**Aufbau**	**Einsatzgebiet**
Feed Forward	Standard Netzwerk	Einfache Aufgaben
Convolutional	Pooling- und Filter-Schichten zum Finden von Charakteristiken	Bildverarbeitung
Recurrent	Rückkopplung der gleichen oder unterschiedlicher Schichten	Spracherkennung, Übersetzung

In der Tabelle sind in der rechten Spalte die typischen Anwendungsgebiete der neuronalen Netzwerkarchitekturen festgehalten. Aus diesen wird bereits ersichtlich, dass CNN oder

[1] https://www.pygame.org/docs/ - Zugriffsdatum: 07.07.23

RNN vermutlich nicht die optimale Wahl für dieses Vorhaben sind. Dabei ist anzumerken, dass CNNs besonders gut geeignet sind, um Charakteristiken oder auch Muster in Bildern zu erkennen. Dies könnte genutzt werden, um Zusammenhänge zwischen Aktionen und den daraus resultierenden Würfelzuständen zu erkennen. So könnte man einem Convolutional Neural Network beispielsweise nicht nur ein Bild als Eingabe zuführen, sondern gegebenenfalls gleich mehrere, in welchen der Zusammenhang zwischen den Zuständen deutlich wird durch aufeinanderaufbauende Würfeltransformationen. Auch Max Lapan führt in seinem Buch auf, dass ein Convolutional Neural Network zielführend sein könnte, da das Netz möglicherweise von Faltungen profitiert.

Ein Recurrent Neural Network besitzt den Vorteil, dass Daten über einen kurzen Zeitraum gespeichert werden. Dies ist unter Umständen hilfreich, um die Beziehung zwischen Würfelaktionen zu erkennen. Lösungsalgorithmen arbeiten oft mit einer dreistufigen Struktur, bestehend aus einem Setup-Schritt[1], einer eigentlichen Aktion und einem invertierten Setup-Schritt (Bsp: U R U'). Durch Anwendung einer solch dreistufigen Struktur werden bestimmte Teile des Würfels gezielter beeinflusst. Soll beispielsweise nur die Position eines Ecksteins verändert werden, hilft solch eine Bewegungsabfolge dabei, dass möglichst wenig andere Steine durch diese mitverändert werden. Dem RNN ist somit eventuell möglich, solche Verbindungen zu erkennen und auf diesen aufzubauen. Aufgrund dessen, dass Lapan sowie McAleer et al. auf einfache Feedforward Netzwerke in ihren Ausarbeitungen [18, 21, 22] setzen und sich hier zunächst in die komplexe Thematik eingearbeitet werden soll, wird auch hier die vorwärtsgerichtete Netzwerkarchitektur verwendet werden. In dem zweiten Artikel [22] des Forscherteams wird eine abgewandelte Form des Feed Forward Netzwerkes genutzt, das Residual Neural Network. Durch mehr Schichten und einem somit tieferen Aufbau ist es möglich, komplexere Funktionen zu modellieren, was sich in diesem Einsatzgebiet als zweckmäßig erweisen könnte. In einem zweiten Teil soll deshalb auch mit einem Residual Netzwerk gearbeitet werden, dem Vorbild von McAleer et al. entsprechend.

5.1.5 Training des neuronalen Netzwerkes

Im Abschnitt 5.1.2 auf Seite 102 wurde bereits artikuliert, dass die künstliche Intelligenz in einer OpenAI Gym Umgebung trainiert wird. Es fehlt noch, wie das Ganze geschehen

[1] Zugfolge oder Aktion, die vor der Anwendung eines bestimmten Algorithmus verwendet wird. Dadurch wird der nachfolgende Algorithmus effektiver oder einfacher anzuwenden.

soll. Q-Learning, welches in der zugehörigen Sektion 2.1.5 vorgestellt wurde, ist hier aufgrund des riesigen Zustandsraumes des Rubik's Cube nicht durchführbar. Es ist schlichtweg nicht möglich, eine Tabelle für jedes mögliche Zustands-Aktions-Pärchen zu erzeugen, welche für jeden Zustand die bestmögliche Aktion beinhaltet. Des Weiteren gilt es immernoch festzulegen, wie das Netzwerk trainiert werden soll. Es wäre nicht zielführend, dem Netzwerk einen unzusammenhängenden Würfelzustand mit dem entsprechenden Reward für diesen zu übergeben. Denn dieser ist für alle außer dem Lösungszustand null oder negativ, je nach Festlegung.

Verbesserungen könnten durch Hinzufügen von zusätzlichen Rewards und Punishments erzielt werden. Der Lösungsalgorithmus nach Fridrich beginnt beispielsweise damit, zunächst ein weißes Kreuz auf der entsprechenden Würfelseite zu erzeugen und anschließend die fehlenden Ecken mit einem weiteren Algorithmus anzupassen. Es könnten Rewards für solche Zwischenschritte hinzugefügt werden. Dadurch wäre das Training der KI allerdings nicht mehr autodidaktisch, und unter Umständen lernt die KI den Lösungszustand niemals kennen und versucht immer nur die belohnten Zwischenschritte zu erreichen. Alternativ könnten (zusätzlich) Bestrafungen für das Überschreiten einer maximalen Anzahl von Schritten oder für Würfelaktionen, welche die vorherige Aktion durch die Anwendung der invertierten Transformation negiert, implementiert werden. Doch auch hier wirft das dieselben Probleme auf wie bei zusätzlichen Belohnungen.

Eine in Papern zu diesem Thema weit verbreitete Trainingsmethode ist das von dem Forschungsteam um McAleer entwickelte autodidaktische Iterationsverfahren (kurz ADI). In diesem werden dem Netwerk zusammenhängende Würfelzustände präsentiert, indem immer im Lösungszustand begonnen wird und dieser schrittweise durch eine zufällige Aktion verändert wird. Dabei werden immer die Folgezustände ausgewertet, und der beste Wert eines der Folgezustände wird als Label für den aktuellen Zustand ausgewählt. Dieser Zustand wird wieder durch eine zufällige Transformation verändert, dann wieder ausgewertet et cetera. Aufgrund der Menge an Artikeln, die diese Technik nutzen, und der Einfachheit von der Methode, soll auch hier das autodidaktische Iterationsverfahren Anwendung finden.

5.1.6 Anwendung des neuronalen Netzes

Üblicherweise stellt ein trainiertes neuronales Netz die Kernkomponente eines Systems dar und wäre alleinstehend vollkommen ausreichend, um die nötigen Berechnungen an-

zustellen. Max Lapan schildert in seinem Buch [18], dass es bei Anwendungen mit solch einem großen Zustandsraum indessen zu Komplikationen kommen kann. Dort führt er aus, dass folgende Schritte theoretisch funktionieren sollten, dies in der Praxis jedoch leider nicht tun:

1. Aktuellen Zustand an das neuronale Netz übergeben.
2. Auswahl der Aktion, welche den besten Wert vom neuronalen Netz zugewiesen bekommen hat.
3. Anwenden der Aktion.
4. Wiederholen der vorherigen Schritte bis der Lösungszustand gefunden wurde.

Diese Herangehensweise ist laut Lapan stark abhängig von der Qualität des Modells. Durch die Größe des Zustandsraumes und Art des neuronalen Netzes sei es nicht möglich, die KI so zu trainieren, dass diese immer die optimale Aktion für jeden Zustand ermittelt. Das Netzwerk weise zwar in vielversprechende Richtungen, könne sich aber genauso gut verirren. Dies äußert sich beispielsweise durch Endlosschleifenbildung von Aktionen, welche immer wieder nacheinander ausgeführt werden (beispielsweise U R L U R L et cetera). Es können schlichtweg nicht alle möglichen Zustände im Training vorkommen. Nichtsdestotrotz können die Hinweise der KI sinnvoll mit zusätzlichen Algorithmen kombiniert werden, um den Zielzustand zuverlässig zu finden. McAleer et al. setzen dabei in ihrem früheren Paper [21] auf eine Kombination einer KI mit einer Monte-Carlo-Baumsuche, welche Lapan in seinem Buch versucht nachzuimplementieren. Auch hier soll deshalb in einem ersten Teil ein neuronales Netz trainiert und anschließend mit einer MCTS, mit den beiden Quellen als Vorbild, vereinigt werden. In dem zweiten Artikel McAleers [22] wird ein Netzwerk mit einer tieferen Architektur verwendet, welches anschließend einem A*-Algorithmus die richtige Richtung weisen soll. Diese Thesis wird deshalb auch einen zweiten Teil aufweisen, der sich damit beschäftigt, diese Herangehensweise nachzubilden.

Ein wichtiger Punkt, welchen Max Lapan in Quelle [18] anspricht, ist die benötigte Trainingszeit eines neuronalen Netzwerkes, welche vor allem von der Rechenleistung der zur Verfügung stehenden Hardware abhängig ist. In dem Artikel [21] arbeitet das Forscherteam mit drei Nvidia GPUs des Typs Titan Xp mit jeweils 12 Gigabyte RAM. Mit Hilfe dieses Setups trainierte das Team ein neuronales Netz für 44 Stunden, bei welchem dieses ungefähr 8 Milliarden Würfelzustände ingesamt und etwa 50.000 Zustände pro Sekunde

gesehen hat [21]. Lapan stand im Gegensatz dazu eine Nvidia GTX 1080 Ti mit 11 Gigabyte RAM zur Verfügung, mit welcher dasselbe Training sechs Tage lang dauern würde [18]. Für diese Arbeit wird eine Grafikkarte GTX 1050 Ti von Nvidia zum Einsatz kommen, welche mit 4 GB RAM nicht einmal die Hälfte an Ressourcen aufbieten kann wie die GPU von Lapan. Ursprünglich sollte mehr Rechenleistung von der Universität des Erstellers bereit gestellt werden, aufgrund eines Cyberangriffs war dies jedoch leider nicht mehr möglich und es bleibt bei der einen Grafikkarte. Um das Problem mit den fehlenden Rechenkapazitäten zu umgehen, greift Lapan auf die kleine Version des Zauberwürfels zurück, den Pocketcube, welcher auch sechs Seiten besitzt, allerdings nur mit jeweils vier farbigen Flächen. Lapan führt aus, dass dafür eine neue angepasste Trainingsumgebung notwendig ist, welche das Verhalten des kleineren Würfels simuliert. Mit dieser Änderung dauere das Training seines Netzes nur noch ungefähr eine Stunde.

Da die hier zur Verfügung stehenden Ressourcen noch weniger leistungsfähig sind, wird diese gefundene Lösung aufgegriffen und auch eine Umgebung für den Pocketcube implementiert, mit welcher die neuronalen Netze in den respektiven Teilen dieser Arbeit trainiert werden. Dies wird im groben Vergleich der Hardware nicht eine, sondern vermutlich eher zwei bis drei Stunden in Anspruch nehmen. Außerdem sollen die Hyperparameter des zu trainierenden Netzwerkes manuell gefunden und angepasst werden, da hierdurch ein größerer Lerneffekt bezüglich deren Einfluss auf den Lernerfolg einer KI erzielt wird. Dies erfordert mehrere Durchläufe einer Trainingseinheit, um möglichst optimale Parameter für das Netzwerk zu finden. Ein Training mit einer KI für den Rubik's Cube, für welches nicht die notwendigen Ressourcen zur Verfügung stehen, würde somit einfach zu viel Zeit kosten. Dies ist auch der Grund, warum eine Grid Search hier vermutlich nicht in Frage kommt, da mit steigender Anzahl von Parametern auch die benötige Zeit exponentiell steigen würde. Sollte nach Abschluss der konzeptionell ausgearbeiteten Punkte noch genügend Zeit zur Verfügung stehen, ist es möglich, noch Versuche mit dem Rubik's Cube anzustellen. Bis dahin besitzt die Ausarbeitung der Arbeit hinsichtlich des Pocketcubes mehr Priorität. Dadurch, dass der Fokus auf den kleineren Zauberwürfel gelegt wird, ändert sich auch die funktionale Anforderung an den optimalen Lösungsweg (≤ 20, *Gottes Zahl*) (S-F8, in Tabelle 4.6 auf Seite 96). Die *Gottes Zahl* beträgt für den Pocketcube nämlich 14 Rotationen, werden halbe Drehungen der Würfelseiten als zwei einzelne Vierteldrehungen gewertet. Die Anpassung ist der Übersicht halber in Tabelle 5.3 dargestellt.

Tabelle 5.3: Angepasste funktionale Anforderung an die Software.

ID	Priorität	Anforderung
...	...	...
S-F8	Niedrig	Finden eines optimalen Lösungsweges ($\leq$ 14, *Gottes Zahl*)

5.1.7 Bedienoberfläche

Die zu entwerfende Bedienoberfläche soll einerseits die Handhabung der Software für Anwender:innen vereinfachen und andererseits die Einzelkomponenten des Systems (KI, Würfelerkennung, Algorithmen) miteinander vereinen. So sollen Benutzer:innen grundsätzlich zwei Möglichkeiten zur Zustandsaufnahme des Würfels geboten werden. Entweder soll die Erfassung des Würfels automatisiert passieren, indem der Würfel in eine passende Hardware eingelegt wird und anschließend mit einer Kamera die Farben erkannt werden. Anschließend werden seine Seitenflächen mit Hilfe von Schrittmotoren rotiert. Alternativ sollen die Farben des Würfels händisch eingetragen werden können. Wird eine Hardware angeschlossen, sind Bedienelemente notwendig, welche das Verbinden (und Trennen) zu dieser ermöglichen und dessen Verbindungsstatus anzeigen. Im selben Zuge ist es auch zweckmäßig, das aufgenommene Bild der Kamera und gegebenenfalls nur die aktuelle Würfelseite, die es zu erfassen gilt, separat anzuzeigen. Andernfalls wäre ein Element hilfreich, welches den Aufbau eines aufgeklappten Würfels in 2D enthält und in dem Farben händisch zugeordnet werden können. Sollte das System gänzlich ohne Hardware genutzt werden, ist eine ansprechende Ausgabe des Lösungsweges erforderlich, mit welcher Anwender:innen den Würfel eigenständig in den Zielzustand bringen können. Im vorherigen Abschnitt wurde bereits diskutiert, dass zwei unterschiedliche Algorithmen in Verbindung mit einem neuronalen Netz zur Lösungsfindung genutzt werden sollen, weshalb es wünschenswert ist, eine Auswahlmöglichkeit des bevorzugten Lösungsverfahrens einzurichten. Das zur Bedienoberfläche zugehörige Ablaufdiagramm ist der Abbildung 5.3 zu entnehmen.

5.2 Hardware

Prof. Dr. Hensel hat sich bereit erklärt, die Entwicklung eines Demonstrators zu übernehmen, welches auch die konzeptionelle Ausarbeitung beinhaltet. Der Demonstrator

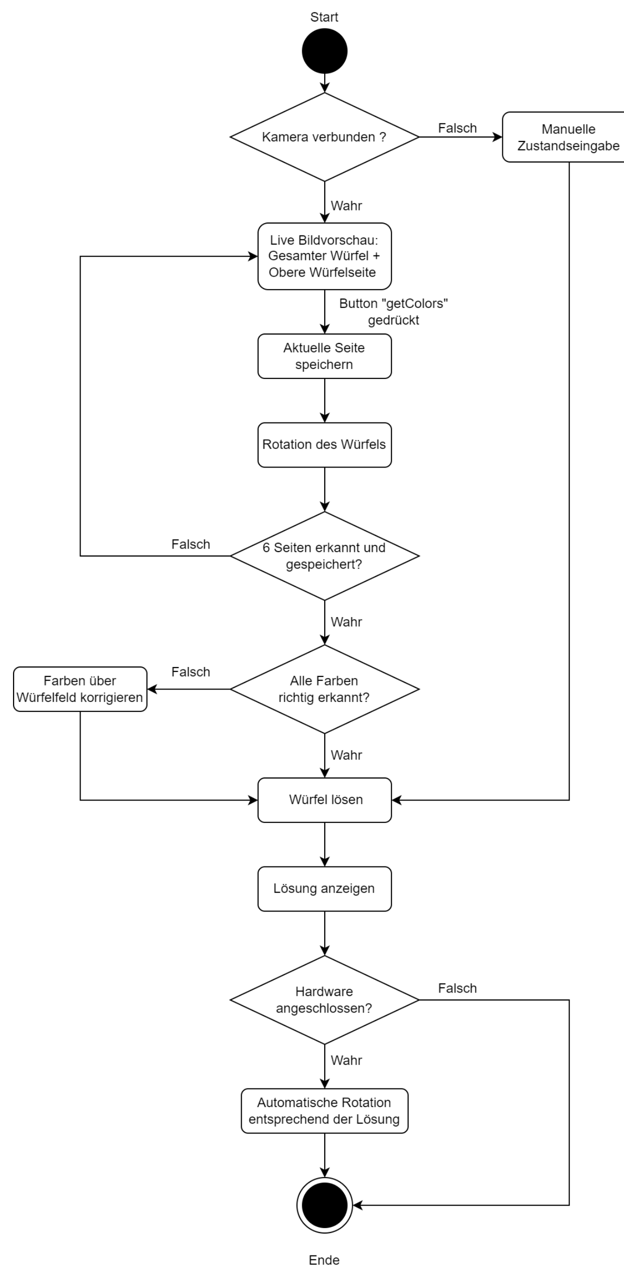
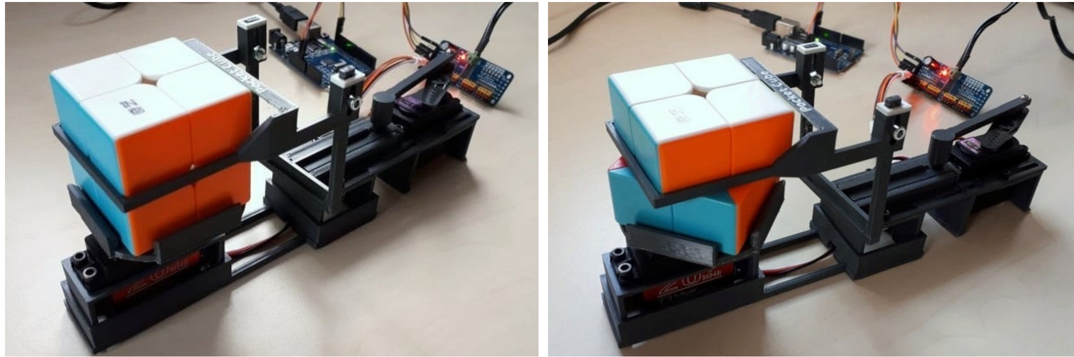
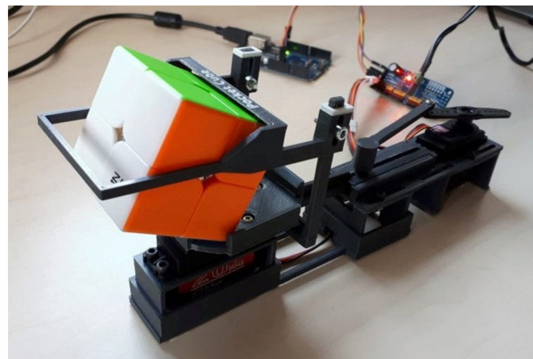


Abbildung 5.3: Ablaufdiagramm der Bedienoberfläche.

besteht hierbei hauptsächlich aus 3D-gedruckten Teilen. Durch die Integration von zwei Servomotoren können bestimmte Komponenten der Konstruktion über eine zugehörige Python-API (Application Programming Interface, *Programmierschnittstelle*) bewegt werden, wodurch eine Rotation der Würfelseiten ermöglicht wird. Die Python-Softwa-



(a) Ruhezustand. (b) Rotation der unteren Seite.



(c) Kippen des gesamten Würfels.

Abbildung 5.4: 3D-gedruckter 2x2x2-Zauberwürfellöser [14].

re-Schnittstelle sendet Befehle an einen angeschlossenen Arduino, welcher wiederum die Ansteuerung der Motoren übernimmt. Durch eine zusätzlich gedruckte Kamerahalterung kann eine Webcam auf dem Demonstrator angebracht werden, sodass diese für eine automatisierte Würfelzustandsaufnahme genutzt werden kann. In Abbildung 5.4 ist der entworfene Demonstrator dargestellt. Weitere Informationen zu der entworfenen Hardware können in [14] nachgelesen werden.

6 Softwareentwicklung: Custom Environment für OpenAI Gym

Die Entwicklung eines Custom Environments (*Benutzerdefinierte Umgebung*) bringt einige Vorteile mit sich, wenn mit Reinforcement Learning Algorithmen gearbeitet wird. Durch den immer gleichen Aufbau der Umgebungen in OpenAI Gym können Bibliotheken wie OpenAI Baselines[1] oder TensorFlow Agents[2] genutzt werden, welche mit der OpenAI Gym API kompatibel sind und bereits Reinforcement Learning Algorithmen zur Verfügung stellen. Somit müssen diese nicht selbst implementiert werden und bringen eine Zeitersparnis mit sich. Durch den standardisierten Aufbau lassen sich zugleich die Ergebnisse von verschiedenen Umgebungen und Algorithmen besser miteinander vergleichen. Mittels integrierter Benchmarking-Tools können die Resultate auch mit Forschungsergebnissen von anderen Personen auf Scoreboards verglichen werden. Der große Vorteil liegt jedoch in der Zugänglichkeit, welche es anderen Forschern ermöglicht mit der entworfenen Umgebung weiterzuarbeiten und eigene Algorithmen mit Hilfe des Custom Environments weiterzuentwickeln. Für diese Arbeit soll zunächst eine eigene Umgebung für den Pocketcube entwickelt werden, wobei sich auf den Programmcode[3] von Max Lapan [18] gestützt wird.

6.1 Bibliotheken

Für die Implementation der eigenen Umgebung sind üblicherweise zwei Bibliotheken notwendig: OpenAI Gym und Pygame.

[1] https://openai.com/research/openai-baselines-dqn - Zugriffsdatum: 07.07.23
[2] https://www.tensorflow.org/agents - Zugriffsdatum: 07.07.23
[3] https://github.com/Shmuma/rl/tree/master/articles/01_rubic/libcube/cubes - Zugriffsdatum: 07.07.23

6.1.1 OpenAI Gym

Die Bibliothek OpenAI Gym stell die Klasse „gym.Env“ zur Verfügung, von welcher das selbst entworfene Custom Environment erben kann. Diese enthält bereits Methoden und Attribute, welche die Interaktion mit der Umgebung ermöglichen. Dazu gehören vor allem folgende Funktionen:

- step(): Führt eine mögliche Aktion aus (beispielsweise durch den Agenten). Zurückgegeben werden die entsprechende Belohung, der neue Zustand und ob das Spiel beendet wurde.
- reset(): Setzt die Umgebung und die zugehörigen Varibalen in den Ausgangszustand zurück.
- render(): Zeigt den aktuellen Zustand der Umgebung mittels Grafik- oder Textausgabe an.

Diese Methoden sind mindestens notwendig, damit die Umgebung nahtlos mit anderen Gym-Tools und -Funktionen zusammenarbeiten kann.

Um OpenAI Gym zu installieren, kann der Paketmanager *pip* genutzt werden. Es ist folgender Befehl in die Kommandozeile oder das Terminal einzugeben:

```
1 pip install gym
```

Anschließend kann die Bibliothek in Python über nachfolgenden Code importiert werden:

```
1 import gym
```

6.1.2 Pygame

Pygame ist für die Entwicklung eines Custom Environments nicht zwingend erforderlich und ermöglicht lediglich die grafische Zustandsausgabe der Umgebung bei Nutzung der render()-Funktion. Die Zustandsausgabe kann allerdings auch über eine Textausgabe in der Kommandozeile der verwendeten IDE erfolgen. Eine grafische Ausgabe bietet sich jedoch an, da diese eine natürlichere Möglichkeit für den Menschen ist, komplexe Informationen wahrzunehmen und zu verstehen. Mit einer grafischen render()-Funktion

können Benutzer:innen den Zustand der Umgebung und die durchgeführten Aktionen intuitiver nachvollziehen. Deshalb soll auch hier Pygame verwendet werden, um eine grafische Darstellung umzusetzen.

Pygame kann wieder über den Paketmanager von Python mittels folgendem Befehl installiert werden,

```
1 pip install pygame
```

und wird über folgende Zeile im Programmcode eingebunden:

```
1 import pygame
```

Eine ausgiebige Dokumentation zu den zugehörigen Funktionen ist auf der Homepage[1] von Pygame nachzulesen.

6.2 Würfelumgebung

Bevor die zu OpenAI Gym Environments typischen Funktionen implementiert werden, gilt es die Darstellung der Würfelzustände und -aktionen zu klären.

6.2.1 Zustände

Die zum Zauberwürfel zugehörige Terminologie und Notation wurde bereits im Abschnitt 3.1 auf Seite 60 erläutert und ändert sich für den Pocketcube nicht. Da der 2x2x2-Würfel quasi nur aus den Ecksteinen eines Rubik's Cube besteht, fällt die Observierung der Position und Orientierung der Kantensteine weg. In der Konzeptentwicklung wurde bereits festgelegt, dass die Darstellung des Zauberwürfelzustandes am besten über die einzelnen Cubies geschieht, da diese Repräsentation weniger redundant ist als beispielsweise über die Darstellung mittels der unterschiedlich farbigen Facelets des Würfels. Des Weiteren geht die Beziehung der Cubies untereinander deutlicher hervor, was sich als hilfreich bei dem Training des neuronalen Netzes herausstellen könnte. In der Konzeptausarbeitung befindet sich Listing 5.2, welches die Implementation von Max Lapan in seinem Buch [18] aufzeigt. Auch hier soll diese Repräsentationsweise aufgegriffen werden, jedoch fallen

[1] https://www.pygame.org/docs/ - Zugriffsdatum: 07.07.23

die Komponenten für die Kantensteine weg (vgl. Abbildung 6.1).

```
State = collections.namedtuple("State", field_names=['corner_pos',
                                                     'corner_ort'])
...
initial_state = State(corner_pos=(0, 1, 2, 3, 4, 5, 6, 7),
                      corner_ort=(0, 0, 0, 0, 0, 0, 0, 0))
```

Listing 6.1: Kompakte Darstellung des Urzustandes des Pocketcubes [18].

6.2.2 Aktionen

Der Aktionsraum wird in Python mit Hilfe einer enum-Klasse[1] erstellt, wodurch jedem Buchstaben ein eindeutiger Integer-Wert zugeordnet wird (vgl. Listing 6.2).

```
class Action(enum.Enum):
    R = 0
    L = 1
    U = 2
    D = 3
    F = 4
    B = 5
    r = 6
    l = 7
    u = 8
    d = 9
    f = 10
    b = 11
```

Listing 6.2: Zuweisung von Zahlenwerten zu den Zauberwürfelaktionen [18].

Durch zusätzliche Implementation eines Dictionaries[2] wird außerdem jeder Aktion die entsprechende inverse Aktion zugeordnet. Ein weiteres Dictionary enthält die Informationen über die Positionswechsel der Cubies und den zugehörigen Orientierungsveränderungen (vgl. Listing 6.3). Die erste Zeile jeder Aktion sagt aus, welche Cubies es zu tauschen gilt. Bei (x, y) wird Cubie x an den Ort von y bewegt. Die zweite Zeile gibt Aufschluss über die zugehörige Orientierung, die der Cubie anschließend besitzt. Hierbei steht eine Null für 0°, eine Eins für 120° und eine Zwei für 240° beziehungsweise −120°. Für die Transformationen, welche aus Permutation (Position) und Rotation (Orientierung) der jeweiligen Cubies bestehen, wurden entsprechende Funktionen erstellt, welche im Anhang

[1] https://docs.python.org/3/library/enum.html - Zugriffsdatum: 07.07.23
[2] https://docs.python.org/3/tutorial/datastructures.html#dictionaries - Zugriffsdatum: 07.07.23

nachgelesen werden können.

```
1     _transform_map = {
2         Action.R: [
3             ((1, 2), (2, 6), (6, 5), (5, 1)),    # corner map
4             ((1, 2), (2, 1), (5, 1), (6, 2))],   # corner rotate
5         Action.L: [
6             ((3, 0), (7, 3), (0, 4), (4, 7)),
7             ((0, 1), (3, 2), (4, 2), (7, 1))],
8         Action.U: [
9             ((0, 3), (1, 0), (2, 1), (3, 2)),
10            ()],
11        Action.D: [
12            ((4, 5), (5,  6), (6, 7), (7, 4)),
13            ()],
14        Action.F: [
15            ((0, 1), (1, 5), (5, 4), (4, 0)),
16            ((0, 2), (1, 1), (4, 1), (5, 2))],
17        Action.B: [
18            ((2, 3), (3, 7), (7, 6), (6, 2)),
19            ((2, 2), (3, 1), (6, 1), (7, 2))]
20    }
21
```

Listing 6.3: Informationen für die Transformation der Würfelzustände [18].

6.2.3 OpenAI Gym Funktionen

Nicht nur die drei grundlegenden Funktionen müssen bei der Erstellung eines eigenen Environments implementiert, sondern auch die Attribute festgelegt werden, welche dieses beschreiben. Dazu gehören vor allem der Aktions- und Beobachtungsraum, welche sich hier aus der Anzahl der möglichen Transformationen und der Art der Zustandsrepräsentation ergeben (vgl. Listing 6.4).

```
1 class Cube_2x2(gym.Env):
2     def __init__(self):
3         self.action_space = spaces.Discrete(len(Action))
4         self.observation_space = State(corner_pos=tuple(range(8)),
5                                        corner_ort=tuple([0]*8))
6 ...
```

Listing 6.4: Definition des Aktions- und Beobachtungsraumes bei der Initialisierung einer Pocketcube-Umgebung.

Hierbei beinhaltet der Beobachtungsraum den aktuellen Würfelzustand, welcher sich mit Hilfe der step()-Funktion verändern lässt.

step()

Diese Funktion wird in den Gym-Umgebungen aufgerufen, wenn der Agent einen Schritt in diesen ausführt. Sie nimmt als Eingabe die gewählte Aktion und gibt dafür den nächsten Zustand und die entsprechende Belohnung aus. Des Weiteren zeigt diese an, ob das Spiel nach Durchführung der Aktion beendet ist oder nicht. Für den Pocketcube ist dies im Prinzip die Anwendung einer der möglichen zwölf Transformationen auf den aktuellen Zustand. Ist der Würfel gelöst, wird 1 als Reward ausgegeben, andernfalls -1. Für das Training des neuronalen Netzes mittels autodidaktischem Iterationsverfahren wird es nötig sein, dass eine Aktion angewendet werden kann, ohne dass der aktuelle Zustand verändert wird beziehungsweise ein beliebiger Zustand übergeben werden kann, auf den die Transformation angewendet werden soll. Es sollen die nächstmöglichen Würfelzustände unabhängig voneinander untersucht werden können, damit der mit den besten Werten und die Aktion die zu diesem führt, ausgewählt werden kann. Aus diesem Grund wird der Funktion hier eine weitere Variable übergeben, welche den Zustand, den es zu verändern gilt, beinhaltet. Soll der aktuelle Zustand der Umgebung genutzt werden, bleibt die Übergabe des Zustandes einfach aus (vgl. Listing 6.5).

```python
def step(self, action, state = None):
    if state is None:
        current_state = self.observation_space
    else:
        current_state = state
    new_state = transform(current_state, action)
    if self.is_goal(new_state) == 1:
        reward = 1
        done = True
        self.is_solved = True
    else:
        reward = -1
        done = False
        self.is_solved = False
    info = {}
    if state is None:
        self.observation_space = new_state
    return new_state, reward, done, info
```

Listing 6.5: Realisierung der step-Funktion in der Pocketcube-Umgebung.

reset()

Die reset()-Funktion wird verwendet, um die Umgebung wieder in ihren initialen Zustand zu versetzen (vgl. Listing 6.6). Dazu gehört nicht nur der aktuelle Umgebungszustand, sondern auch alle zugehörigen Variablen wie zum Beispiel die in einer Episode erhaltenen Belohnungen oder die letzte getätigte Aktion. Bevor eine neue Episode initiiert werden soll, ist es üblich, reset() aufzurufen.

```
def reset(self):
    self.observation_space = self.initial_state
..
```

Listing 6.6: Realisierung der reset-Funktion in der Pocketcube-Umgebung.

render()

Die Funktion render() ist optional und ermöglicht es den Zustand eines Environments zu visualisieren. Üblicherweise wird dafür die Bibliothek Pygame verwendet. Mit Hilfe der in Pygame zur Verfügung gestellten Funktionen und Klassen lassen sich einfache Formen für die Grafikausgabe umsetzten. Durch die Nutzung von Rotationsmatrizen [4] ist es auch möglich 3D-Objekte zweidimensional abzubilden. Dies ist allerdings mit erhöhtem Arbeits- und Rechenaufwand verbunden.

2D-Darstellung Um den aktuellen Würfelzustand in 2D darzustellen, muss zunächst die kompakte Repräsentation in eine verständliche Form übersetzt werden, da aus dieser nicht direkt die Anordnung der farbigen Sticker hervorgeht. Dafür wird zuvor in einem Dictionary festgehalten, welcher Cubie welche Farben besitzt:
Durch Veränderung der Position ändern sich die zu den Cubies zugehörigen Farben nicht.

```
corner_colors = (
    ('W', 'R', 'G'), ('W', 'B', 'R'), ('W', 'O', 'B'), ('W', 'G', 'O'),
    ('Y', 'G', 'R'), ('Y', 'R', 'B'), ('Y', 'B', 'O'), ('Y', 'O', 'G')
    )
```

Listing 6.7: Dictionary für die Farben, welche sich auf den zugehörigen Cubies befinden [18].

Die Änderung der Orientierung eines Steines ändert jedoch die Reihenfolge der Farben

innerhalb eines Tupels in Listing 6.7. Dadurch kann mit Hilfe einer entsprechenden Funktion, welche eine Umsetzungstabelle inkorporiert (vgl. Listing 6.8) der aktuelle Zustand in Kompaktdarstellung in eine Form gebracht werden, die für den/die Nutzer:in besser verständlich ist und sich für eine anschließende 2D-Umsetzung besser eignet.

```
corner_maps = (
    # top layer
    ((0, 2), (3, 0), (1, 1)),
    ((0, 3), (4, 0), (3, 1)),
    ((0, 1), (2, 0), (4, 1)),
    ((0, 0), (1, 0), (2, 1)),
    # bottom layer
    ((5, 0), (1, 3), (3, 2)),
    ((5, 1), (3, 3), (4, 2)),
    ((5, 3), (4, 3), (2, 2)),
    ((5, 2), (2, 3), (1, 2))
)
```

Listing 6.8: Umsetzungstabelle für die Darstellung über Cubies nach Facelets unter Berücksichtung der Position und Orientierung der Würfelsteine [18].

Die aufgeklappte Darstellung des Pocketcubes (vgl. Abbildung 6.1) ist für eine 2D-Repräsentation und eine leichtere Nachvollziehbarkeit besser geeignet, da direkt jede Auswirkung einer getätigten Aktion ersichtlich wird. Würde der Würfel dreidimensional abgebildet werden, wären immer mindestens drei Seiten für den/die Anwender:in nicht sichtbar, weshalb sich die Umsetzung einer zweidimensionalen Darstellung eher anbietet. Dabei wird für jeden farblichen Sticker auf dem Pocketcube ein Quadrat erzeugt, welches mit der Farbe des ihm zugewiesenen Facelets ausgefüllt wird. Durch Hinzufügen von Textfeldern lassen sich zusätzlich relevante Informationen anzeigen, wie zum Beispiel die letzte getätigte Transformation oder die Anzahl der getätigten Rotationen. In Abbildung 6.1 ist die Ausgabe der entworfenen render()-Funktion für den Pocketcube exemplarisch dargestellt.

3D-Darstellung Eine dreidimensionale Abbildung des Würfels gestaltet sich als wesentlich komplexere Aufgabe als das zweidimensionale Pendant. Um ein 3D-Objekt auf einem Bildschirm abzubilden, müssen dessen neue Koordinatenpunkte mit Hilfe von Drehmatrizen berechnet werden. Damit die farbigen Flächen jedoch auch korrekt abgebildet werden und die gezeichneten Objekte sich nicht überschneiden, gilt es den Maleralgorithmus [1] zu berücksichtigen. Schaut der/die Anwender:in von vorne auf den Würfel und die Rückseite von diesem würde zuletzt gezeichnet werden, wäre die Vorderseite überdeckt

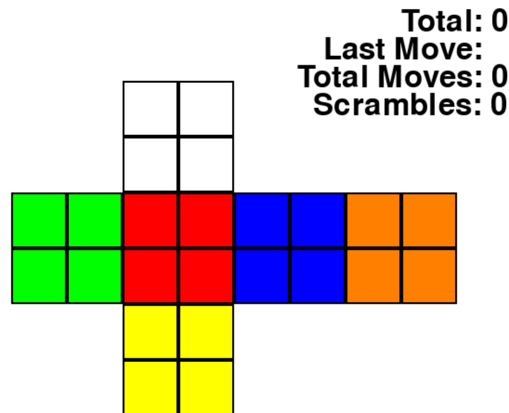


Abbildung 6.1: 2D-Darstellung des aktuellen Zustands in der Pocketcube-Umgebung.

und es kann keine korrekte Abbildung des Pocketcubes stattfinden. Deshalb muss die z-Koordinate der zu zeichnenden Polygone bei der Darstellungsreihenfolge in Betracht gezogen werden. Da bei einer Repräsentation des Würfels als dreidimensionales Objekt mindestens drei Seiten für den/die Anwender:in verdeckt werden, bietet es sich an, den Würfel rotierbar zu gestalten, sodass durch entsprechende Orientierungsveränderungen der ganze Würfel betrachtet werden kann. Dadurch ist es aber nicht oder nur sehr schwer möglich, erstellten Quadraten direkt farbige Sticker des Würfels zuzuweisen, da sich die Koordinaten durch die Rotationen verändern. Dies hat wiederum Einfluss auf die Reihenfolge, in der die Polygone gezeichnet werden, sodass es nicht zu Überschneidungen bei der Darstellung kommt. Damit der Würfel rotiert werden kann, gilt es zudem, entsprechende Tastenzuweisungen zu implementieren, die dies ermöglichen. In diesem Fall wird der Nummernblock der Tastatur verwendet, um den Pocketcube um seine eigenen Achsen zu rotieren. Die Würfeltransformationen werden durch die Bewegung der entsprechenden Koordinatenpunkte der Cubies auf einer Kreisbahn umgesetzt, welche von der jeweiligen Aktion betroffen sind. Die Drehungen werden über zugewiesene Tasten (R, L, U, D, F, B) durchgeführt. Durch zusätzliches Halten der Shift-Taste kann die zugehörige inverse Bewegung realisiert werden. Auch hierbei muss die aktuelle Orientierung des Würfels berücksichtigt werden. Der Abbildung 6.2 ist die implementierte 3D-Darstellung des Pocketcubes zu entnehmen.

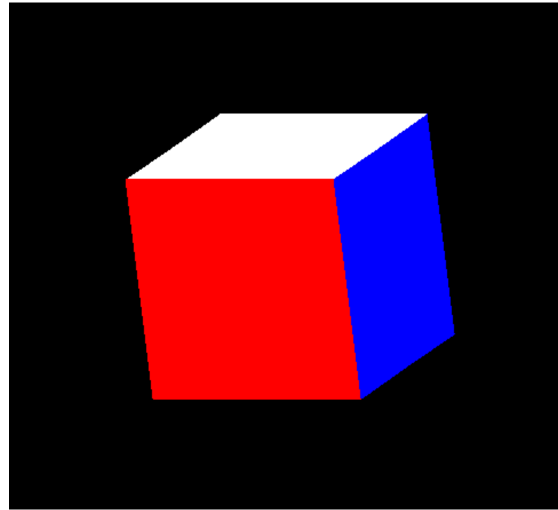


Abbildung 6.2: 3D-Darstellung des Pocketcubes.

6.2.4 Weitere Funktionen

Zu guter Letzt werden hier noch weitere Funktionen der entwickelten Umgebung aufgeführt, die nicht zum Standardrepertoire der Gym API gehören, allerdings eine wichtige Rolle bei dem Training des neuronalen Netzes erfüllen. Dabei soll nicht auf alle Funktionen eingegangen werden, sondern nur auf wenige relevante.

scramble()

Um das neuronale Netz später mit zufällig generierten Zuständen zu speisen, ist es sinnvoll, eine Funktion zu generieren die diesen Zweck erfüllt. Dafür wurde scramble() entworfen, welche eine vorgegebene Anzahl von zufälligen Transformationen auf den aktuellen oder übergebenen Zustand ausführt. Dabei ist berücksichtigt worden, dass eine Aktion nicht direkt durch ihre Inverse zunichte gemacht wird und somit die eigentliche Anzahl an Verdrehungen nicht mehr der Vorgabe entspräche.

is_goal()

Die is_goal()-Funktion überprüft, ob der übergebene Zustand der Zielzustand ist und gibt dementsprechend eine Eins aus, wenn dies der Fall ist. Andernfalls ist die Ausgabe

null. Dies ist relevant für die nachfolgende Funktion, welche die Folgezustände überprüft.

explore_state()

Diese Funktion ist Kernbestandteil der autodidaktischen Iteration und prüft mittels is_goal(), ob einer der Folgezustände eines übergebenen Zustandes der Zielzustand ist. Durch schrittweise Veränderung des aktuellen Würfelzustandes, beginnend im Lösungszustand, werden dem neuronalen Netz die Zusammenhänge zwischen den aufeinanderfolgenden Zuständen besser näher gebracht. Bereits nach der ersten Rotation aus dem Zielzustand heraus erkennt das Netzwerk durch das „Zurückblicken“ mit Hilfe der explore_state()-Funktion, dass einer der überprüften Zustände der Zielzustand ist und welche Aktion zu diesem führt. Bei der nächsten Verdrehung besitzt der vorherige Zustand beziehungsweise die Aktion, die zu diesem führt, eine höhere Güte als die anderen untersuchten Folgezustände beziehungsweise den jeweiligen Aktionen, die zu jenen führen, und es zeichnet sich ein deutlicher Pfad für das Netzwerk zum Lösungszustand ab.

7 Softwareentwicklung: Bedienoberfläche

Um Anwender:innen zu ermöglichen, die entwickelten Lösungsmethodiken zu nutzen, ist es zweckmäßig, eine eingängige Bedienoberfläche (*Graphic User Interface*, kurz GUI) anzufertigen. Dabei soll weniger darauf eingegangen werden, wie die GUI designt wurde, sondern vielmehr welche Komponenten implementiert wurden und deren Funktionsweise. Das Hauptaugenmerk liegt hierbei auf der Integration der von Prof. Dr. Hensel entworfenen Hardware mit den hier untersuchten Suchalgorithmen. Damit die beiden Teilbereiche sinnvoll miteinander vereinigt werden können, muss eine Option eingebunden werden, die es ermöglicht, den Zustand eines realen Würfels in die Software zu übertragen. Entweder geschieht dies über die Einbindung einer Kamera oder mittels manueller Eingabe über ein entsprechendes Bedienfeld. Es sollen beide Optionen verwirklicht werden, falls Anwender:innen keine Kamera zur Verfügung stehen sollte.

7.1 Bibliothek

Für das Design der Benutzeroberfläche wurde die Bibliothek *CustomTkinter*[1] verwendet, welche eine Weiterentwicklung der Standard Python Bibliothek *Tkinter* darstellt. CustomTkinter bietet eine modernere und anpassbarere Bedienoberfläche an, mit zusätzlichen Widgets und Styling-Optionen. Zu den vorgefertigten Widgets gehören vor allem Schaltflächen, Beschriftungen und Textfelder, aber auch Objekte wie Fortschrittsbalken und scrollbare Rahmen. Des Weiteren stellt die Bibliothek weitere Funktionen zur Verfügung, wie zum Beispiel Drag & Drop und anpassbare Menüs.

Um CustomTkinter zu installieren kann der Paketmanager pip von Python genutzt werden. Es ist folgender Befehl in die Kommandozeile oder ein Terminal des Betriebssystems einzugeben:

[1] `https://github.com/TomSchimansky/CustomTkinter` - Zugriffsdatum: 07.07.23

```
1    pip install customtkinter
```

Listing 7.1: Befehl um CustomTkinter zu installieren.

Wie bereits erwähnt, soll nicht darauf eingegangen werden, welche Komponenten verwendet und wie diese erstellt wurden. Eine ausreichende Dokumentation zur Bibliothek befindet sich unter folgender Adresse[1]. Der integrierte Gridmanager ist für die Platzierung der Objekte auf der Benutzeroberfläche zuständig und dessen Funktionsweise ist unter Umständen nicht direkt ersichtlich. Daher empfiehlt es sich, weitere Informationen zu diesem Thema einzuholen. Eine Möglichkeit dafür bietet folgender Link[2].

7.2 Aufbau

Die entworfene Benutzeroberfläche teilt sich grob in zwei Bereiche auf (vgl. Abbildung 7.1), die nachfolgend näher erläutert werden.

7.2.1 Bereich: Darstellungen

Im linken Bereich der GUI befinden sind alle für Benutzer:innen relevanten Informationen bezüglich des Würfelzustandes.

Kamera

Oben links befindet sich das Feld für die Kamera, mittels welcher sich der aktuelle Zustand des Pocketcubes erfassen lässt. Für die Umsetzung wurde ein fester Bereich eingestellt, der nur die obere Würfelseite des Pocketcubes erfasst und nachfolgend in der oberen rechten Ecke des Kamera-Feldes angezeigt wird. Durch den Button *Get Colors*, welcher sich links in der unteren Sektion des Einstellungsbereiches befindet, können die Farben, die sich im eingestellten Feld befinden, aufgenommen werden.

[1] https://customtkinter.tomschimansky.com/documentation - Zugriffsdatum: 07.07.23
[2] https://tkdocs.com/tutorial/grid.html - Zugriffsdatum: 07.07.23

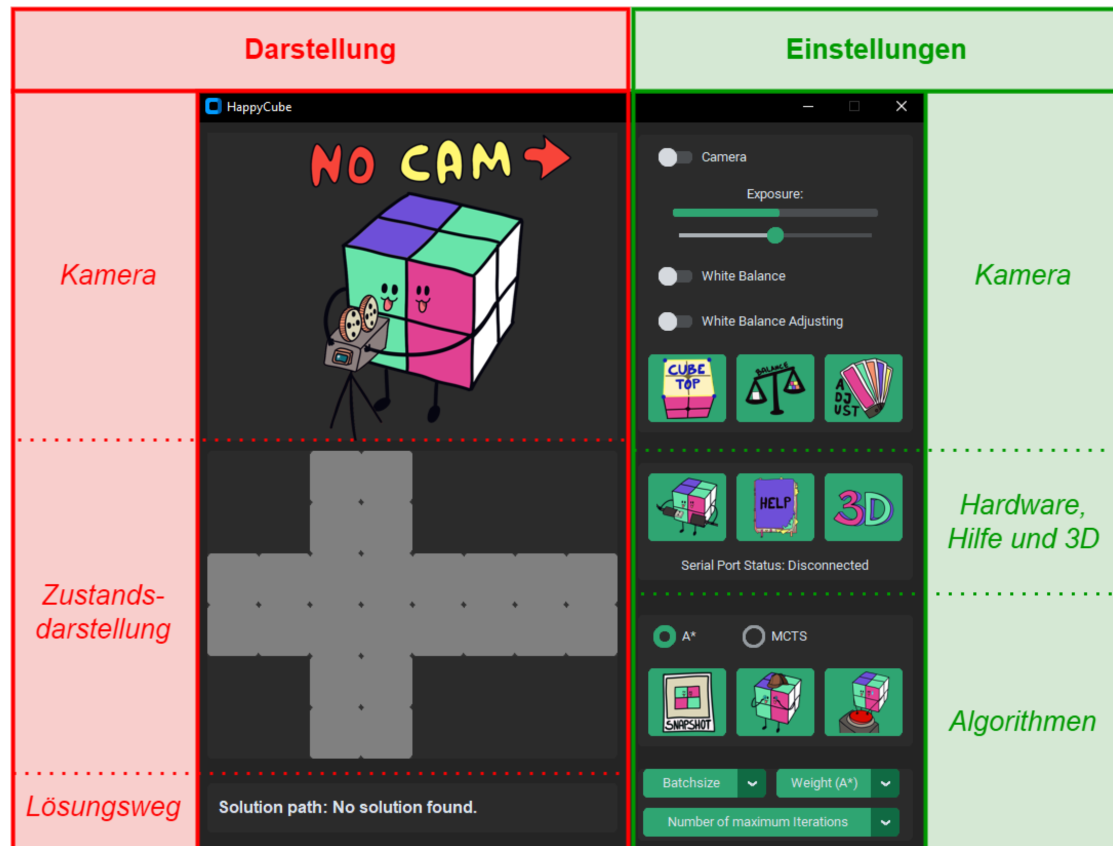


Abbildung 7.1: Aufbau der grafischen Benutzeroberfläche.

Würfelzustand

Die durch die Kamera erfassten Farben werden automatisch in diesem Feld auf der zugehörigen Würfelseite angezeigt. Dieses dient nicht nur Anschauungszwecken, sondern kann auch zur manuellen Eingabe des Würfelzustandes genutzt werden, indem auf die entsprechenden Felder geklickt wird. Dabei entspricht die Anzahl der getätigten Mausklicks einer der sechs Farben des Würfels.

Lösungsweg

Mit Hilfe eines Textfeldes wird die Ausgabe des Lösungsweges umgesetzt, den der eingestellte Algorithmus für den erfassten Zustand gefunden hat.

7.2.2 Bereich: Einstellungen

In dieser Sektion lassen sich alle Einstellungen treffen, die unter Umständen nötig sind, um die korrekte Funktionsweise zu gewährleisten.

7.2.3 Kamera

Um eine optimale Erfassung der Farben zu gewährleisten, ist es in bestimmten Umgebungen notwendig die Belichtungszeit und die Farbbalance des Kamerabildes anzupassen. Über den *Exposure*-Slider kann zunächst die Belichtung eingestellt werden, also wie viel Licht den Bildsensor erreicht. Für eine korrekte Farbwiedergabe ist die Implementation eines Weißabgleichs unabdingbar, welche sicherstellt, dass die Farben des Bildes natürlicher aussehen und vergleichbare RGB-Werte in unterschiedlichen Lichtverhältnissen besitzen. Da sich die Lichtverhältnisse während der Nutzung ändern können, erscheint es zweckmäßig, eine Funktion zu integrieren, welche die ermittelten Werte für den Weißabgleich während der Nutzung anpasst. Durch Verwendung von Farbvergleichen mittels platzierter Farbkarten im Bild kann bei Veränderungen der Lichtbedingungen festgestellt werden, wie sich die RGB-Werte der Farbkarten verändert haben und die Parameter für den Weißabgleich angepasst werden müssen. Für beide Methoden sind entsprechende Schalter implementiert worden, mit denen diese ein- beziehunsgsweise ausgeschaltet werden können. Um einen Referenzpunkt für den Weißabgleich auszuwählen, muss zuvor der dafür vorgesehene Button in der Mitte angeklickt werden. Für die Auswahl der Farbkarten, welche für die dynamische Anpassung der Lichtverhältnisse notwendig sind, ist der rechte Button vorgesehen. Mit dem linken Button werden die Eckpunkte der Würfeloberseite gekennzeichnet, damit eine automatische Farberkennung von der Software durchgeführt werden kann.

Hardware, Hilfe und 3D

Über diesen Bereich lässt sich die angebundene Hardware mit der Software verbinden. Das Verbinden oder Trennen geschieht durch das Anklicken des zugehörigen Buttons *Connect* (links). Der Status des Gerätes wird hierbei über ein darunterliegendes Textfeld angezeigt. Dort befindet sich außerdem der *Help*-Button (Mitte), über den ein Fenster geöffnet werden kann, in dem die Funktionen aller abgebildeten Buttons näher erläutert werden (vgl. Abbildung 7.2). Über den *3D*-Button (rechts) lässt sich ein Pygame-Fenster

öffnen (vgl. Abschnitt 6.2.3 auf Seite 123), in welchem der Würfelzustand dreidimensional dargestellt wird. Durch das Drücken entsprechender Tasten auf der Tastatur kann der Würfelzustand mit den zugehörigen Rotationen verändert werden.

Algorithmen

Im letzten Bereich der GUI kann eingestellt werden, welcher Algorithmus zur Lösung des erfassten Würfelzustandes genutzt werden soll. Des Weiteren befindet sich dort der bereits erwähnte Button *Get Colors*, welcher für die Aufnahme der Würfelfarben zuständig ist. Wurde dieser betätigt, rotiert der Würfel automatisch zur nächsten Seite, damit auch dessen Farben aufgenommen werden können. Diese Farbaufnahme ist nicht vollständig automatisiert, da es bei der Nutzung durch schlechte Lichtverhätnisse trotz implementierter Gegenmaßnahmen dazu kommen kann, dass die Farbwiedergabe nicht korrekt ist. Deshalb wird Nutzer:innen die Möglichkeit geboten, diese selbstständig zu kontrollieren und gegebenenfalls mittels entsprechender Einstellungen zu korrigieren, bevor die Farberfassung durchgeführt wird. Sollte dennoch ein Fehler passieren, kann der Würfelzustand über den *Reset*-Button (rechts) zurückgesetzt werden. Ist der Zustand vollständig übertragen und der Algorithmus der Wahl eingestellt worden, kann der Würfel mittels *Solve*-Button (Mitte) gelöst werden. Durch Auswahllisten, welche sich ganz unten befinden, können weitere Einstellungen bezüglich der Algorithmen vorgenommen werden, wie die Anzahl der gleichzeitig untersuchten Knotenpunkte, den Einfluss der Distanzgewichtung im A*-Algorithmus und der maximalen Anzahl an Iterationen, bevor ein Suchdurchlauf abgebrochen wird.

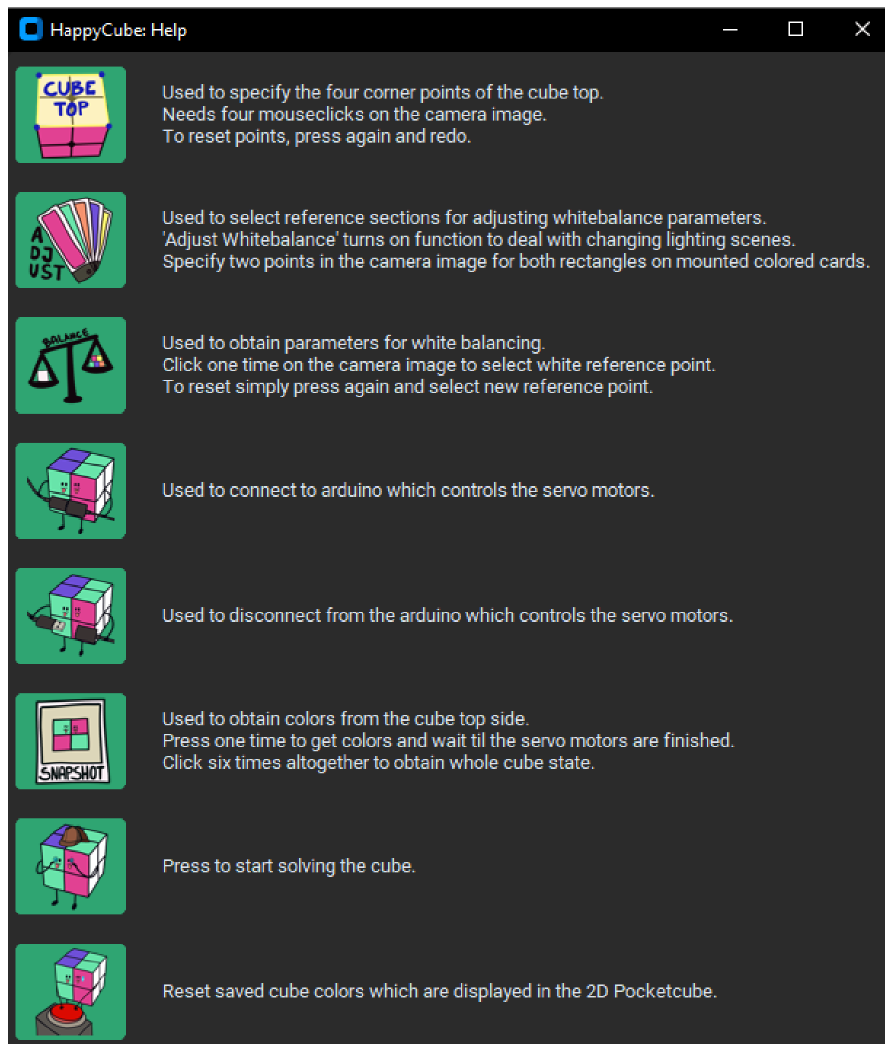


Abbildung 7.2: Hilfefenster der Anwendung.

8 Algorithmusentwicklung: Monte-Carlo-Baumsuche

In diesem Kapitel wird die in [21] vorgestellte Methode zur Lösung von Zauberwürfeln ausgearbeitet. Anstelle dass ein Rubik's Cube gelöst werden soll, kommt hier ein Pocketcube zum Einsatz. Dafür wird ein neuronales Netzwerk trainiert, welches es anschließend in eine Monte-Carlo-Baumsuche zu integrieren gilt, um die Suche zu leiten. Die für das Vorhaben benötigten Verfahren wurden von Lapan bereits in Code [18] umgesetzt und werden für diese Ausarbeitung als Grundlage dienen. Dieser Code soll in den folgenden Abschnitten erläutert und den Anforderungen entsprechend angepasst werden.

8.1 Datenrepräsentation

Um das Training des neuronalen Netzes zu ermöglichen, ist es zuallererst notwendig, die Zauberwürfelzustände in eine für das Netz verständliche Repräsentation zu bringen. Die kompakte Darstellung, welche Aufschluss über die Position und Orientierung der einzelnen Cubies gibt, ist nicht als Input für das Netzwerk geeignet. Stattdessen ist es gebräuchlich die Daten in One-Hot-Kodierung darzustellen [18]. Dafür wird ein 8x24 Tensor (8 Ecksteine á 8 Positionen und 3 Orientierungsmöglichkeiten) über eine dafür angefertigte Funktion erzeugt, welcher eine Zeile für jeden Cubie besitzt und eine Spalte, welche angibt, in welcher Position und Orientierung sich dieser befindet (vgl. Listing 8.1). Die Position jedes Cubies wird in seiner entsprechenden Zeile über eine 1 gekennzeichnet, während die restlichen Positionen eine 0 erhalten.

```
1     [[1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
2      [0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
3      [0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
4      [0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
5      [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0.]
6      [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0. 0. 0. 0.]
7      [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0. 0. 0. 0.]
8      [0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]]
```

Listing 8.1: Darstellung des Urzustandes des Zauberwürfels in One-Hot-Kodierung.

Abbildung 8.1: Minimal angepasste Architektur des von McAleer et al. verwendeten neuronalen Netzwerkes [21, modifiziert].

8.2 Neuronales Netz

8.2.1 Architektur

Die verwendete Architektur des neuronalen Netzes stimmt fast vollständig mit der von Quelle [21] überein. Die Forschungsgruppe aus [21] nutzt ein vorwärtsgerichtetes neuronales Netzwerk (*Feedforward*) mit vollständig verbundenen Schichten, dessen Struktur in Abbildung 8.1 dargestellt ist. Anders als bei den beiden Vorbildern dieser Arbeit werden hier zusätzlich Schichten für Batch-Normalisierung verwendet, welche die numerischen Eingabedaten im Netz normalisieren. Dies soll den Trainingsprozess stabilisieren und die Konvergenz beschleunigen [16]. Zudem werden die Gewichte des Netzwerkes zu Beginn nach der Glorot Initialisierungstechnik initialisiert. Nach welcher Technik die Netzwerke aus [18, 22] initialisiert wurden, fand in den Artikeln keine Erwähnung. Das Netzwerk gibt einerseits einen Skalar v (Value) aus, welcher den Gütewert des aktuellen Zustandes

repräsentieren soll, also wie weit dieser vom Zielzustand entfernt ist, und andererseits einen Vektor p (Policy), welcher Wahrscheinlichkeiten der möglichen Aktionen beinhaltet und somit angibt, welche am ehesten zum gewünschten Zielzustand führen wird (vgl. Abbildung 8.1). Zusammengefasst gibt der Policy-Part des Netzes also an, welche Aktion durchgeführt werden soll, und der Value-Part wie groß die Güte des aktuellen Zustandes ist.

8.2.2 Hyperparameterauswahl

Viele Gründe haben dafür gesprochen, die Hyperparameterauswahl manuell durchzuführen. Optimale oder sehr wahrscheinlich bessere Parameter wären mittels einer Gridsearch oder Randomsearch gefunden worden [7], allerdings muss dafür eine ausreichende Rechenleistung gegeben sein. Es hätten mehrere Netzwerke vollständig nach dem vorgestellten Schema trainiert werden müssen und selbst dann ist bei solch einer Methode nicht zwangsläufig gegeben, dass eindeutig ermittelt werden kann, welches Modell die Aufgabe tatsächlich besser erledigt. Ein Beispiel dafür wäre, wie oft welche Würfelzustände in welcher Reihenfolge dem Netz beim Training gezeigt wurden. Unter Umständen hat das eine Netz einfach mehr unterschiedliche Zustände gesehen und konnte die Daten besser generalisieren. Die Trainingsdaten werden hier während des Trainings generiert und durch ein immer besser werdendes Netz mit den seiner Berechnungen nach richtigen Labels gekennzeichnet, wodurch von vornherein Schwierigkeiten bei dem Lernerfolg auftreten können.

Ein weiterer ausschlaggebender Grund, die Hyperparameterauswahl händisch zu ermitteln, ist der Lerneffekt, der dabei erzielt wird. Es wird wesentlich deutlicher, welcher Parameter wie viel Einfluss auf das Lernverhalten eines Netzwerkes hat, wenn diese nach und nach feinjustiert werden, damit das neuronale Netz letztendlich den Ansprüchen entsprechend funktioniert. Da es in diesem Fall eine hohe Anzahl möglicher Kombinationen von Hyperparametern gibt, stützt sich diese Arbeit auf die von der Forschungsgruppe um McAleer in deren Artikel [21] und die von Max Lapan in seinem Buch [18] gewählten Parameter. Teilweise mussten diese angepasst werden, da entweder die zur Verfügung stehende Hardware nicht ausgereicht hat oder während des Anlernens beziehungsweise der Anwendung des Netzwerkes festgestellt wurde, dass nicht dieselben Ergebnisse erzielt werden konnten. Im Folgenden soll die hier getroffene Auswahl an Hyperparametern kurz erläutert werden. Dabei sei angemerkt, dass bei der Feinjustierung das Training oftmals vorzeitig beendet wurde, falls kein Lernprozess stattgefunden hat, dieser divergierte oder

viel zu früh anfing zu stagnieren. Deshalb ist es im Rahmen dieser Arbeit nicht möglich, jede Hyperparameterkombination mit Bildern zu begründen.

Aktivierungsfunktion

Sowohl McAleer et al. [21] als auch Lapan [18] verwenden bei ihrem neuronalen Netzwerk die eLU-Aktivierungsfunktion. Allerdings wird in beiden Fällen nicht begründet, aufgrund welcher Kriterien diese Art der Aktiverung ausgewählt wurde, und es wurden deshalb verschiedene Funktionen getestet. Da die zur Verfügung stehenden Hardwareressourcen begrenzt sind, wird hier auf eine andere Funktion gesetzt, welche auch auf der ReLU-Aktivierung basiert. Zur Anwendung kommt die Leaky-ReLU-Funktion, welche durch ihren angepassten Verlauf dem sogenannten „Dying ReLU“-Problem entgegenwirken soll. Anders als bei der artverwandten eLU-Aktivierungsfunktion besitzt diese im negativen Bereich jedoch einen linearen statt exponentiellen Verlauf, wodurch eine schnellere Berechnung möglich ist.

Optimierer

McAleer et al. verwenden bei ihrem Training des neuronalen Netzes einen RMSprop-Optimierer [21]. Sie erläutern allerdings wieder nicht, aufgrund welcher Kriterien dieser ausgewählt wurde. Lapan nutzt einen Adam-Optimierer [18], jedoch auch ohne nähere Erläuterungen. Um zu testen welcher Optimierer eingesetzt werden soll, wurde das in Abschnitt 2.2.5 auf Seite 36 erläuterte Vorgehen eingesetzt. Damit konnte ermittelt werden, welcher Optimierer in welchem Lernratenbereich agieren kann, und dementsprechend einer ausgewählt. Da Adam bei diesem Versuch am besten abgeschnitten hat, wird dieser bei dem Training des Netzes verwendet werden. Damit ist dieser nicht zwangsläufig der geeigneteste Optimierer für diese Aufgabe, aber es wären deutlich umfangreichere Tests mit einer Vielzahl von möglichen Einstellungskombinationen (Lernrate, Momentum, Gewichtsreduzierung etc.) von Nöten.

Lernrate

Die verwendete Lernrate wird in Artikel [21] nicht näher erläutert und Lapan nutzt bei dem Training seines Netzwerkes eine Rate von 10^{-5}, auf welche er in seinem Buch [18] auch nicht näher eingeht. Um eine passende Lernrate zu ermitteln, wird auf das

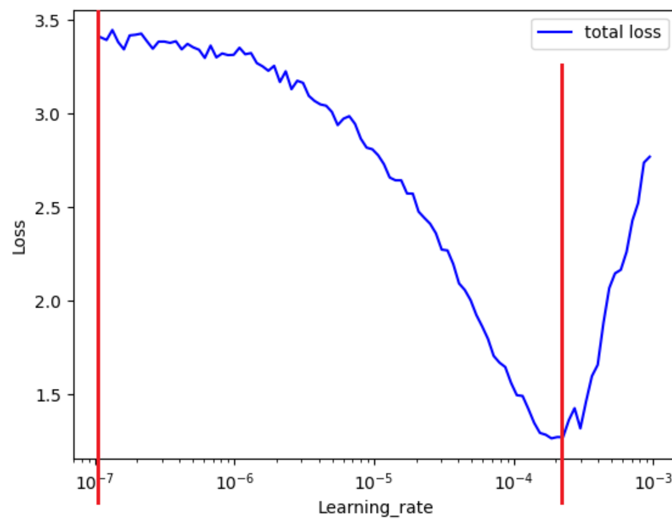


Abbildung 8.2: Gesamtverlust bei iterativer Erhöhung der Lernrate um den Faktor 1, 1 für das neuronale Netz mit MCTS.

im Abschnitt 2.2.5 auf Seite 36 erläuterte Verfahren zur Lernratenfindung zurückgegriffen. Dabei ergibt sich der Verlauf in Abbildung 8.2 bei iterativ ansteigender Lernrate beginnend bei 10^{-5} und endend bei 10^{-3}.

Die beiden roten Linien stellen den über die Lernratenfindung ermittelten Bereich dar. Hier wurde sich für einen Anwendungsbereich zwischen $5 \cdot 10^{-5}$ und 10^{-6} entschieden.

Lernratenscheduler

Aufgrund der Erkenntnisse über die Anpassung der Lernrate während des Trainings, welche in Abschnitt 2.2.5 auf Seite 37 näher erläutert wurden, wird hier ein zyklischer Lernratenscheduler verwendet.

Batchsize

Ein weiterer wichtiger Hyperparameter ist die Größe der Batches, die dem neuronalen Netz pro Epoche zum Lernen übergeben werden. Max Lapan verwendet dabei eine Größe von 10.000 Würfelzuständen pro Epoche als Netzinput, welche sich auch als guter Parameter für ein Training mit antiproportionaler Distanzgewichtung herausgestellt hat.

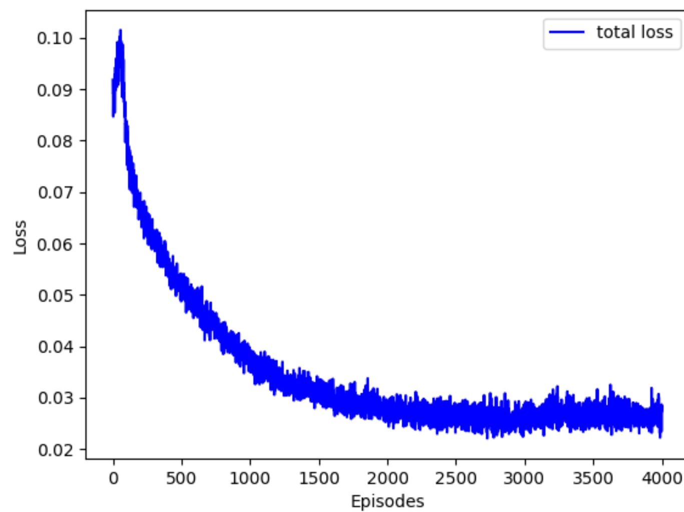


Abbildung 8.3: Gesamtverlust beim Training mit dem Pocketcube mit einer Batchsize von 10.000 Würfelzuständen und einer Distanzgewichtung.

Bei solch einer Gewichtung beeinflussen Zustände, die sich nah am Lösungszustand befinden, die Optimierung der Modellparameter stärker als solche, die weit entfernt sind. Somit lernte das Netz in einer kürzeren Zeit schneller, und es ergibt sich eine glattere Lernkurve mit weniger starken Ausreißern (siehe Abbildung 8.3). Welche Größe die Forschungsgruppe um McAleer et al. in [21] nutzt, geht aus dem Artikel nicht hervor, sie verweisen jedoch in diesem auf eine Quelle über „Mini-Batch-Gradientenabstieg", was auf eine geringere Größe schließen lässt. Da sich die benötigte Zeit bei Verwendung einer kleineren Batchsize jedoch deutlich vergrößert, wird zunächst der Ansatz von Lapan umgesetzt.

In Abbildung 8.3 ist auffällig, dass der Verlust bereits in einem sehr kleinen Bereich (ca. 0.1) startet. Das ist sehr wahrscheinlich darauf zurückzuführen, dass das Modell die Verdrehungen, welche sich nah am Zielzustand befinden, stärker berücksichtigt. Diese hat das Netz schneller gelernt und es ergeben sich nicht so starke Diskrepanzen zwischen den vorhergesagten Aktionen und denen, die tatsächlich zielführend sind.

Es sieht jedoch anders aus, wenn auf die Gewichtung der Verluste verzichtet wird. Anders als bei beispielsweise Max Lapan [18] beginnen die Verluste etwa ab Epoche 3.000 stark zu steigen. In Abbildung 8.4 ist dieses Verhalten erkennbar. Es ist davon auszugehen, dass es sich hierbei um das Exploding Gradient Problem handelt, welches beim Training

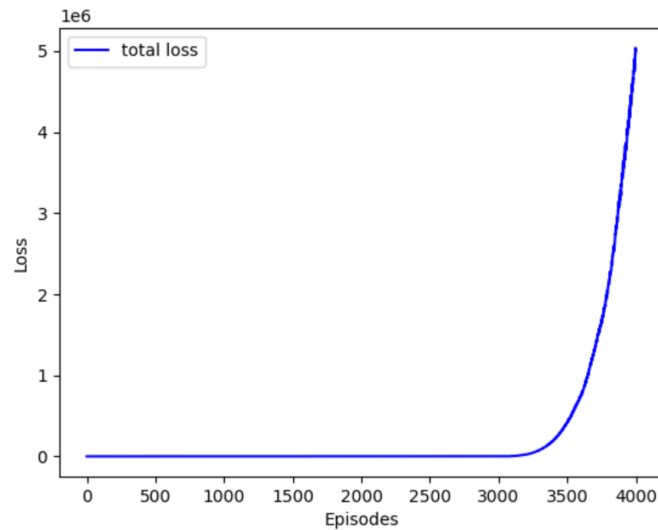


Abbildung 8.4: Gesamtverlust beim Training mit dem Pocketcube mit einer Batchsize von 10.000 Würfelzuständen ohne Distanzgewichtung.

von neuronalen Netzen auftreten kann. Das Verringern der Batchgröße hat schlussendlich zu einer Verbesserung geführt und ein ungewichtetes Training ermöglicht, wodurch auch spätere Verdrehungen besser erkannt und gelöst werden können. Um die Verkleinerung der Batchsize auszugleichen, wurde mit einer höheren Anzahl von Batches insgesamt trainiert. Max Lapan [18] trainierte sein neuronales Netzwerk für den Pocketcube mit insgesamt 4.000 Batches á 10.000 Würfelzuständen. Durch die Reduktion der Größe hat sich das Verhältnis bei diesem Training zu 40.000 Batches á 1.000 Würfelzuständen verschoben. Die Anpassung der Anzahl und der Größe der Batches hatte für das Training des Netzes sehr gute Auswirkungen auf die spätere Erfolgsquote und auf die Dauer, die dieses benötigt, um einen 2x2x2-Würfel zu lösen.

Verlustfunktion

Da das Netzwerk zwei Ausgaben besitzt (Value und Policy), werden auch zwei Verlustfunktionen benötigt. Dem Würfelzustand eine Güte (Value) zuzuweisen, ist ein klassisches Regressionsproblem und dementsprechend eignet sich die MSE (kurz für Mean Squared Error, *Mittlere Quadratische Abweichung*) Verlustfunktion, welche sowohl Lapan in seinem Buch [18] als auch McAleer et al. in ihrer Forschung [21] verwenden. Die bestmögliche Aktion in diesem Würfelzustand zu wählen (Policy), um dem Zielzustand

näher zu kommen, ist in die Kategorie einer Klassifizierungsaufgabe einzuordnen, weshalb hier eine andere Verlustfunktion besser geeignet ist. Wieder entscheiden sich beide oben genannten Quellen in ihren Arbeiten gleich und setzen auf eine Softmax-Kreuzentropie-Verlustfunktion. Da sich beide Quellen bei der Wahl der Funktionen einig sind und diese auch ihrer Anwendung entsprechend ausgewählt wurden (Regression und Klassifikation), werden auch in dieser Arbeit MSE für die Value und Softmax-Kreuzentropie für die Policy eingesetzt werden. Lapan erwähnt in seinem Buch [18], dass es sinnvoll sein könnte, den Policy-Verlust mit einem Entropie-Verlust zu ergänzen. Die Formel für die Entropie lautet hierbei

$$H(x) = -\sum_{i=1}^{n} P(x_i) \ln(P(x_i)) \quad , \tag{8.1}$$

wobei x_i die möglichen Aktionen repräsentiert und $P(x_i)$ die zugehörige Wahrscheinlichkeit, dass diese vom neuronalen Netz gewählt werden. Die zusätzliche Implementation der Entropie ist deshalb sinnvoll, da der Agent schnell dazu tendiert, Aktionen auszuwählen, die in der Vergangenheit hohe Gewinne eingebracht haben, ohne neue Aktionen zu testen, die unter Umständen im aktuellen Zustand noch mehr Gewinne einbringen würden. Dies ähnelt sehr dem Exploration-Exploitation-Dilemma, welches in Sektion 2.1.5 angesprochen wurde. Um den Agenten anzuregen mehr neue Möglichkeiten auszutesten, kann man den Policy-Verlust um den von Lapan vorgeschlagenen Entropieverlust ergänzen. Wie stark die Entropie das Verhalten des Agenten beeinflusst, ist über einen zusätzlichen Hyperparameter β einstellbar, mit welchem der Entropieverlust multipliziert wird. Aufgrunddessen wurde diese Maßnahme zusätzlich implementiert und ist in Listing 8.2 nachzuvollziehen.

```
1 policy_loss_t = F.cross_entropy(policy_out_t, y_policy_t, reduction='none')
2 entropy_loss = -(F.softmax(policy_out_t, dim=1)
3                 * F.log_softmax(policy_out_t, dim=1)).sum(dim=1).mean() * 0.05
4 loss_t = value_loss_t + (policy_loss_t - entropy_loss)
```

Listing 8.2: Regularisierung der Policy durch zusätzlichem Entropieverlust (β=0.05).

8.3 Training des neuronalen Netzwerkes

Das Training des Netzwerkes erfolgt mittels eines einfachen Verfahrens, welches die Forschungsgruppe aus Quelle [21] *Autodidactic Iteration* (kurz ADI, Autodidaktische Ite-

ration) getauft hat. Bei diesem erfolgt das Anlernen des neuronalen Netzes nicht nur mit zufällig generierten und unzusammenhängenden Würfelzuständen, sondern beginnt immer im Zielzustand des Würfels und wird anschließend iterativ mit einer zufälligen Aktion weiter verändert, bis die gewünschte Anzahl an Verdrehungen erreicht worden ist. Somit soll das Netz den Zusammenhang zwischen den aufeinander aufbauenden Zuständen lernen, anstatt zusammenhangslos zerwürfelte Zustände als Eingabe zu erhalten. Anschließend werden die möglichen Folgezustände eines jeden Zustands s in dieser Aneinanderreihung mit Hilfe des neuronalen Netzes auf ihre Güte untersucht. Durch diese ist es möglich zu ermitteln, ob einer der Folgezustände der gewünschte Zielzustand ist, und dem aktuellen Zustand s wird mittels folgender Formel ein neuer Wert zugewiesen [21]

$$y_{vi} = \max_a (v_s(a) + R(A(s,a))) \quad , \tag{8.2}$$

wobei $A(s,a)$ der Folgezustand ist, nachdem Aktion a auf Zustand s angewendet wurde, und $R(s)$ den entsprechenden Reward des Zustandes darstellt (vgl. Grundlagen 2.1.4 auf Seite 10) [18]. $R(s)$ ist 1 für den Zielzustand und -1 für alle anderen Zustände. Für das Netzwerk ist jedoch nicht nur der maximale Wert aller Folgezustände von s von Interesse (vgl. Formel 8.2), sondern auch welche Aktion zu diesem geführt hat. Durch eine leichte Anpassung der Formel 8.2 erhalten wir nicht den Wert, sondern den Index und somit die entsprechende Aktion:

$$y_{pi} = \operatorname*{argmax}_a (v_s(a) + R(A(s,a))) \tag{8.3}$$

Zusammen ergeben y_{vi} und y_{pi} das Trainingsziel $Y_i = (y_{vi}, y_{pi})$, welches den optimalen Wert und die optimale Aktion für den Zustand s repräsentiert. Y_i wird anschließend im weiteren Verlauf des Trainings als Label für den Zustand s verwendet.

Das Training erfolgt somit nach dem Ablauf, der in Abbildung 8.5 dargestellt ist.

1. Würfel befindet sich im Zielzustand s_{goal}
2. Anwendung aller möglichen Transformationen auf s_{goal}
3. Netzwerk ermittelt Value-Werte der Folgezustände

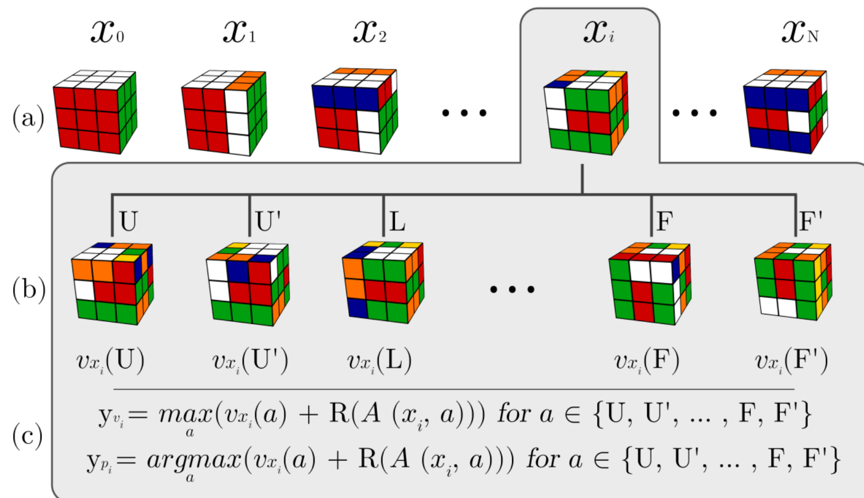


Abbildung 8.5: Visualisierung der Trainingsdatengenerierung mittels ADI [21].

4. Berechnen des maximalen Wertes y_{vi} (Formel 8.2) und des zugehörigen Indexes y_{pi} (Formel 8.3)
5. Zuweisung der Werte zu dem Zustand s_{goal}
6. Anwendung einer zufälligen Aktion a auf den Zustand s_{goal}
7. Wiederholen der Vorgänge (2. - 6.) auf den neuen Zustand s'
8. Wiederholen des gesamten Vorgangs bis gewünschte „Zerwürfeltiefe“ erreicht worden ist

Das neuronale Netz ist jedoch in der Lage mehrere Aneinanderreihungen von Zuständen parallel zu verarbeiten, weshalb es sich als sinnvoll erweist, diesem direkt eine größere Anzahl davon zu übergeben. Diese steht in Abhängigkeit zur gewählten Batchgröße und der gewünschten „Zerwürfeltiefe“ (realisiert über Variable „depth“). Bei einer Batchsize von 1.000 und einer Tiefe von 200 Würfelzuständen würde das Verfahren fünf mal wiederholt werden. Der Python-Code für das Erzeugen einer Reihenfolge ist dem Listing 8.3 zu entnehmen. Bei dem Code in Listing 8.3 ist anzumerken, dass in Zeile vier eine Maßnahme implementiert wurde, die verhindert, dass nach Drehung einer Würfelseite direkt die inverse Bewegung dazu folgen kann. Ansonsten könnte die Anzahl der tatsächlichen Würfeldrehungen variieren und unter Umständen die Qualität des Zielmodells beeinträchtigen. In Zeile sechs werden die jeweiligen Folgezustände ermittelt und in Zeile sieben

```python
for d in range(depth):
    move = random.choice(moves)
    moves = list(Action)
    moves.remove(inv_move)
    cube_state,_ ,_ ,_ = cube.step(cube_state, move)
    explored_states, goals = cube.explore_state(cube_state, encoded=True)
    result.append(d+1, encode_inplace(np.zeros((8,24)), cube_state),
                  explored_states, goals)
return result
```

Listing 8.3: Erzeugung einer zusammenhängenden Reihenfolge von Würfelzuständen nach dem ADI-Verfahren [21] in Python.

und acht zusammen übergeben. Das Auswerten von diesen geschieht in einer separaten Funktion, welche die Trainingsdaten labelt. Dafür wird jedem der aufeinanderfolgenden Zustände der beste Wert eines seiner Folgezustände (vgl. Formel 8.2) und die Aktion, die zu diesem führt (vgl. Formel 8.3) zugewiesen. Die Umsetzung in Python sieht dabei wie in Listing 8.4 aus. Von Zeile 23 bis 30 wurde zusätzlich die von Max Lapan in [18] geschilderte „Zero-Goal-Methode“ implementiert. Auch sein Modell neigte dazu, ab einem gewissen Zeitpunkt im Training zu divergieren, und der Wertverlust nahm infolgedessen exponentiell zu. Lapan ermittelte experimentell, dass dieses Verhalten auf fehlerhafte Zielwerte zurückzuführen ist. Die Formel 8.2 addiert den vom neuronalen Netz zurückgegebenen Wert immer zu der aktuellen Belohnung, unabhängig davon, ob der aktuelle Zustand bereits der Zielzustand ist. Somit kann der vom Netz ausgegebene Wert fast beliebig sein [18]. Deshalb wurde die Berechnung des Zielwertes für einen Zustand von Lapan angepasst und auch hier übernommen (vgl. Formel 8.4).

$$y_{vi} = \begin{cases} \max_a(v_s(a) + R(A(s,a))), & \text{wenn } s \text{ kein Zielzustand ist} \\ 0, & \text{wenn } s \text{ der Zielzustand ist} \end{cases} \tag{8.4}$$

8.4 Anwendung des trainierten Netzwerkes

Durch das Trainieren des Netzwerkes erhält man ein Modell, welches für einen beliebigen Zustand s ausgeben soll, welche Aktion am ehesten zum gewünschten Zielzustand führen wird. Man könnte annehmen, dass, wenn man immer die Aktion mit dem höchsten Wert auswählt (Greedy Best-First Search), auch irgendwann der Lösungszustand erreicht wird. Aufgrund der Qualität des Modells ist dies jedoch nicht der Fall. Die Größe des Zustandsraumes (Rubik's Cube), sowie die Art des neuronalen Netzes verhindern, dass für jeden Zustand s die ideale Aktion a ermittelt werden kann [18]. Stattdessen soll das Netzwerk

```python
def sample_batch(scramble_buffer, batch_size, net, zero_goal = True):

# Get random sample from buffer
data = random.sample(scramble_buffer, batch_size)
depths, states, explored_states, goals = zip(*data)

# Transform twelve explored states into a readable format for the neural network
explored_states = np.stack(explored_states)
shape = explored_states.shape
explored_states_t = torch.tensor(explored_states, dtype= torch.float32).to(device)
explored_states_t = explored_states_t.view(shape[0]*shape[1], *shape[2:])

# Evaluate value of the explored states
value_t = net(explored_states_t, value_only=True)
# Transform into right format
value_t = value_t.squeeze(-1).view(shape[0], shape[1])

# Add reward to the values
goals_mask_t = torch.tensor(goals, dtype=torch.int8).to(device)
goals_mask_t += goals_mask_t - 1   # has 1 at final states and -1 elsewhere
value_t += goals_mask_t.type(dtype=torch.float32)

# Zero goal method
if zero_goal is True: # mask to set goal states to 0 instead of network output
    goals_mask_t_inv = np.logical_not(goals).astype(int)
    goals_mask_t_inv = torch.tensor(goals_mask_t_inv, dtype=torch.int8).to(device)
    value_t *= goals_mask_t_inv

# Find target value and the respective index
max_val_t, max_act_t = value_t.max(dim=1)

# Transform data into tensor and return it
enc_input = np.stack(states)
enc_input_t = torch.tensor(enc_input, dtype=torch.float32).to(device)
depths_t = torch.tensor(depths, dtype=torch.float32).to(device)
weights_t = 1/depths_t
return enc_input_t.detach(), weights_t.detach(),
       max_act_t.detach(), max_val_t.detach()
```

Listing 8.4: Labeln der erzeugten Daten aus Listing 8.3 in Python.

eher richtungsweisend verwendet werden, um relevante Folgezustände zu erkunden. Vor allem bei dem Rubik's Cube ist die Größe des Zustandsraumes schlichtweg zu gewaltig und es werden immer Zustände auftreten, die das Netzwerk beim Training nicht „gesehen" hat. Es gibt eine Reihe verschiedener Algorithmen, welche das trainierte neuronale Netzwerk als richtungweisende Funktion verwenden, während sie den Zustandsraum nach einem Lösungsweg erkunden. Im Artikel von McAleer et al. aus 2018 [21] oder auch in dem Buch von Max Lapan [18] kommt eine Monte-Carlo-Baumsuche zum Einsatz. In [21] kombiniert das Forscherteam um McAleer das trainierte neuronale Netz (f_θ) mit einer asynchronen Monte-Carlo-Baumsuche. Dabei laufen parallel mehrere Instanzen des Suchalgorithmus gleichzeitig und suchen nach einem möglichen Lösungsweg. Hier soll der Einfachheit halber erst einmal ein Baum implementiert werden, damit das grundsätzliche

Vorgehen verstanden werden kann. Das Vorgehen einer Suche läuft dabei immer nach dem gleichen Schema ab. Der Suchbaum beginnt mit einem Startzustand $T = s_0$ und wird iterativ erweitert, bis ein Blattknoten von T erreicht wird. Würde der vollständige Baum für den Rubik's Cube aufgestellt werden, wäre dieser seinem Zustandsraum enstprechend groß ($4,33 \cdot 10^{19}$). Aufgrunddessen wird immer nur ein kleiner Teilbereich erstellt, da die Rechenzeit sonst ebenfalls entsprechend lang wäre. Es wird hierbei der Teilbereich eines Baumes näher betrachtet, den das Netz als vielversprechend analysiert.

McAleer's Forschungsgruppe [21] beschreibt zwei mögliche Ansätze zur Lösungsfindung:

- Naiver Ansatz: Sobald der Zielzustand gefunden wurde, wird der Pfad als Lösung verwendet.
- Anschließende Breitensuche: Nach Findung des Zielzustandes wird eine Breitensuche durchgeführt, um einen potenziell kürzeren Lösungsweg zu finden. Der Lösungsweg könnte Schleifen und unnötige Aktionsfolgen beinhalten (zum Beispiel R und darauffolgend R'), welche dadurch beeseitigt werden.

In dieser Arbeit stützt sich die MCTS-Implementation auf den von Max Lapan entworfenen Code, welcher in [18] verlinkt wurde. Dort wird nach der Findung des Lösungszustandes eine anschließende Breitensuche durchgeführt. Auch auf dies wird zunächst verzichet, um den allgemeinen Suchvorgang besser nachvollziehen zu können.

Jeder der Zustände innerhalb eines Suchbaumes wird mit zusätzlichen Informationen abgespeichert, um die Suche zu optimieren:

- $N_s(a)$: Zähler, welcher angibt, wie oft eine Aktion a im Zustand s ausgewählt wurde.
- $W_s(a)$: Die Güte einer Aktion a im Zustand s.
- $L_s(a)$: Der virtuelle Verlust einer Aktionen a im Zustand s.
- $P_s(a)$: A-priori-Wahrscheinlichkeit einer Aktion a im Zustand s.

Die zusätzlichen Daten, welche pro Zustand mit hinterlegt werden, wurden hierbei mittels Python Dictionaries umgesetzt, welche als Key den zugehörigen Zustand nutzen (vgl. Listing 8.5).

Diese Dictionaries gehören zu einem Objekt der Klasse „MCTS“, mit welchem eine Instanz des Suchalgorithmus erstellt werden kann.

```
1     # Tree state
2     shape = ((12), )
3     # correspond to N_s(a) in the paper
4     self.act_counts = collections.defaultdict(
5                             lambda: np.zeros(shape, dtype=np.uint32))
6     # correspond to W_s(a)
7     self.val_maxes = collections.defaultdict(
8                             lambda: np.zeros(shape, dtype=np.float32))
9     # correspond to P_s(a)
10    self.prob_actions = {}
11    # correspond to L_s(a)
12    self.virt_loss = collections.defaultdict(
13                            lambda: np.zeros(shape, dtype=np.float32))
14    # children states
15    self.edges = {}
```

Listing 8.5: Umsetzung der zusätzlichen Knoteninformationen für einen hinterlegten Zustand im MCTS-Verfahren [18].

```
1 class MCTS:
2
3 def __init__(self, cube_env, state, net, exploration_c=100,
4              virt_loss_nu=100.0, device="cpu"):
5
6 ...
```

Listing 8.6: Klassenobjekt für einen Monte-Carlo Suchbaum [18].

Jede Suche beginnt mit dem zerwürfelten Zustand $s_{scramble}$, den es zu lösen gilt, und es wird iterativ eine Aktion ausgewählt, bis ein Blattknoten erreicht wird. Die Aktionswahl wird hierbei nach folgender Formel getätigt: [21]

$$A_t = \underset{a}{\operatorname{argmax}}(U_{s_t}(a) + Q_{s_t}(a)) \tag{8.5}$$

$$U_{s_t}(a) = c \cdot P_{s_t}(a) \frac{\sqrt{\sum_{a'} N_{s_t}(a')}}{1 + N_{s_t}(a)} \tag{8.6}$$

$$Q_{s_t}(a) = W_{s_t}(a) - L_{s_t}(a) \tag{8.7}$$

mit:

c – Exploration Hyperparameter

N_{s_t} – Zähler, wie oft Aktion a im Zustand s_t ausgeführt wurde

P_{s_t} – Policy-Wert vom Netzwerk für den Zustand s_t

L_{s_t} – Virtueller Verlust

W_{s_t} – Maximaler Value-Wert für die Folgezustände von s_t

In Formel 8.5 sind die zuvor erwähnten Parameter der Knoten wiederzufinden, welche den Suchverlauf verbessern sollen. Bei dieser Formel handelt es sich um eine abgewandelte Version des UCT-Scores aus dem Grundlagenabschnitt 3.2.8 auf Seite 82. Die ausgegebenen Daten des Netzes sind dabei nicht immer für jeden Zustand hilfreich bei der Suche, aber es treten immer wieder solche auf, die in die Richtung des Zielzustandes weisen.

Der virtuelle Verlust L_{s_t} wird mit Hilfe eines weiteren zusätzlichen Hyperparameters ν aktualisiert [21]:

$$L_{s_t} \leftarrow L_{s_t} + \nu \tag{8.8}$$

Der virtuelle Verlust soll verhindern, dass derselbe Zustand mehr als einmal besucht wird. Laufen mehrere MCTS-Suchen parallel (asynchron), wie bei der Gruppe um McAleer, wird damit außerdem verhindert, dass alle Instanzen von diesem denselben Pfad beim Hinabsteigen des Baumes auswählen [21]. Das Hinabsteigen des Suchbaumes bis zu einem Blattknoten ist in Listing 8.7 implementiert.

Wird ein Blattknoten s_τ erreicht (vgl. Listing 8.7 Zeile 10), wird dieser um dessen Kindknoten erweitert (vgl. Listing 8.8), also alle möglichen Folgezustände von s_τ (vgl. Abbildung 8.6).

```
def _search_leaf(self):

s = self.root_state
path_actions = []
path_states = []

while True:
    next_states = self.edges.get(s)

    if next_states is None:
        break

    act_counts = self.act_counts[s]
    N_sqrt = np.sqrt(np.sum(act_counts))

    if N_sqrt < 1e-6:
        act = random.randrange(12)
    else:
        u = self.exploration_c * N_sqrt / (act_counts + 1)
        u *= self.prob_actions[s]
        q = self.val_maxes[s] - self.virt_loss[s]
        act = np.argmax(u + q)

    self.virt_loss[s][act] += self.virt_loss_nu
    path_actions.append(act)
    path_states.append(s)
    s = next_states[act]
return s, path_actions, path_states
```

Listing 8.7: Hinabsteigen des Suchbaumes bis zu einem Blattknoten [18].

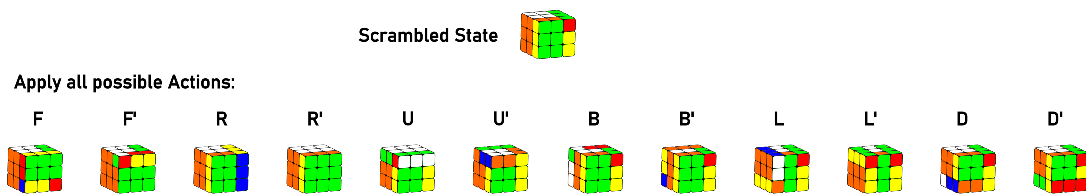


Abbildung 8.6: Erweiterung eines Blattknotens um seine Kindknoten.

```
    child_states, child_goal = self.cube_env.explore_state(s, encoded = False)
    self.edges[s] = child_states
```

Listing 8.8: Erweitern eines Blattknotens um seine Folgezustände [18].

Für jeden dieser Folgezustände s' wird wieder Speicher für die zusätzlichen Daten initialisiert ($N_{s'} = 0$, $P_{s'} = \mathbf{p}_{s'}$, $L_{s'} = 0$, $W_{s'} = 0$) [21]. Die A-priori-Wahrscheinlichkeit $\mathbf{p}_{s'}$ wird hierbei mit dem neuronalen Netzwerk bestimmt (Ausgabe des Policy-Zweigs:

$\mathbf{p}_{s'} = f_\theta(s')$). Anschließend werden Value und Policy für s_τ vom neuronalen Netz berechnet (v_{s_τ}, $\mathbf{p}_{s_\tau} = f_\theta(s_\tau)$) (vgl. Listing 8.9)

```
1 def _expand_leaves(self, leaf_states):
2     policies, values = self.evaluate_states(leaf_states)
3     for s, p in zip(leaf_states, policies):
4         self.prob_actions[s] = p
5     return values
```

Listing 8.9: Evaluieren von Baumknoten mittels neuronalem Netzwerk [18].

Der Value-Wert wird bei allen innerhalb des Pfades besuchten Zuständen aktualisiert, wenn der neu berechnete Wert größer ist als der vorherige (vgl. Listing 8.10). Dass nur die maximale Güte für einen Zustand hinterlegt wird, liegt darin begründet, dass der Zauberwürfel deterministisch ist. Deshalb müssen die Belohnungen nicht gemittelt werden, bevor sich für eine Aktion entschieden wird [21]. Auch Zähler N_{s_t} und virtueller Verlust L_{s_t} werden entsprechend aktualisiert, indem die Variable für den Zähler um den Wert Eins erhöht wird und von dem virtuellen Verlust der zugehörige Hyperparameter ν subtrahiert wird. Der Aktualisierungsvorgang ist von Lapan in Python implementiert worden (vgl. Listing 8.10)

```
1 def _backup_leaf(self, states, actions, value):
2 for path_s, path_a in zip(states, actions):
3     self.act_counts[path_s][path_a] += 1
4     w = self.val_maxes[path_s]
5     w[path_a] = max(w[path_a], value)
6     self.virt_loss[path_s][path_a] -= self.virt_loss_nu
```

Listing 8.10: Aktualisierungsvorgang der Baumknoten, nachdem ein neuer Blattknoten erreicht wurde [18].

Der Suchalgorithmus wird solange ausgeführt bis entweder der Zielzustand s_{goal} gefunden oder eine Abbruchbedingung (maximale Zeit, maximale Anzahl an Schritten) erfüllt wird. Ein vereinfachtes Beispiel für eine Monte-Carlo-Baumsuche ist in Abbildung 8.7 ersichtlich. Hierbei wurden die gewählten Aktionen über Simulationen, welche *Rolling Out* beinhalten, ausgewählt.

8.4.1 Breitensuche

Nachdem ein Lösungspfad zum Zielzustand gefunden wurde, ist es möglich, eine anschließende Breitensuche durchzuführen, um den Pfad gegebenenfalls zu verkürzen. Mit Hilfe

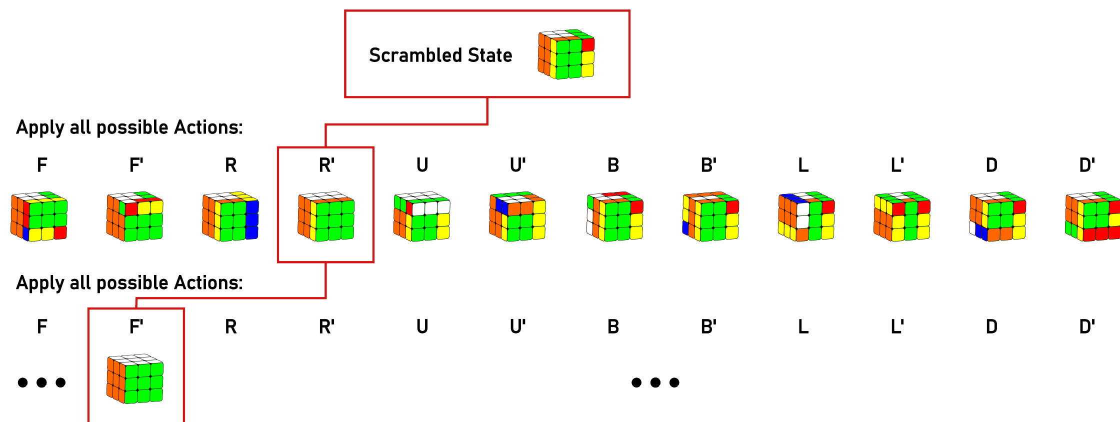


Abbildung 8.7: Vereinfachtes Beispiel der MCTS-Methode anhand eines Zauberwürfels, der zwei Mal verdreht wurde. Aktionen: Front, Right. Lösung: Right inverse, Front inverse (Lösungspfad in Rot).

der Tree Policy in Gleichung 8.6 untersucht der verwendete MCTS-Algorithmus nur die vielversprechendsten Pfade, da ein vollständiger Suchbaum eine enorme Anzahl von Knoten und Kanten benötigt und dementsprechend zu einen enormen Speicherbedarf und einer extrem langen Berechnungszeit führen würde. Durch Anwendung einer Breitensuche im Anschluss an den MCTS wird der erzeugte Suchbaum noch einmal systematisch untersucht und es können unter Umständen kürzere Lösungswege entdeckt werden, die zuvor vom Suchalgorithmus außer Acht gelassen wurden. Somit werden unnötige Aktionsabfolgen, wie vier aufeinanderfolgende identische Rotationen oder eine Drehung gefolgt von ihrer inversen Bewegung, aus dem Lösungspfad entfernt. Der zugehörige Code ist Listing 8.11 zu entnehmen.

```
1  def find_solution(self):
2      queue = collections.deque([(self.root_state, [])])
3      seen = set()
4  
5      while queue:
6          s, path = queue.popleft()
7          seen.add(s)
8          c_states, c_goals = self.cube_env.explore_state(s)
9          for a_idx, (c_state, c_goal) in enumerate(zip(c_states, c_goals)):
10             p = path + [a_idx]
11             if c_goal:
12                 return p
13             if c_state in seen or c_state not in self.edges:
14                 continue
15             queue.append((c_state, p))
```

Listing 8.11: Auf den MCTS-Algorithmus folgende Breitensuche [18].

8.4.2 Stapelverarbeitung

In Lapans Programmcode zum MCTS-Verfahren findet sich auch eine Funktion, welche einen Suchdurchlauf mit einer Stapelverarbeitung (*Batch Processing* oder *Batchverarbeitung*) durchführt. Somit würden mehrere vielversprechende Knotenpunkte *gleichzeitig* untersucht werden, was positive Auswirkungen auf Effizienz und Optimalität des Algorithmus haben könnte. Die dort implementierte Stapelverarbeitung geschieht dabei nicht tatsächlich gleichzeitig, da die Knoten zuvor in einer Liste gespeichert und anschließend sequentiell überprüft werden. Sie werden also nicht parallel verarbeitet, wie beispielsweise beim Multithreading oder Multiprocessing. Dennoch können dadurch die Laufzeit und die Anzahl der benötigten Iterationen pro Suche verbessert werden und es könnte sich als zweckmäßig erweisen, dies genauer zu untersuchen. Die Ergebnisse der eigentlichen Einzelverarbeitung besitzen allerdings Priorität und deshalb werden Versuche mit einer Batchverarbeitung hinten angestellt. Der zur Stapelverarbeitung zugehörige Code ist dem Listing 8.12 zu entnehmen.

```
1     batch_size = min(batch_size, len(self) + 1)
2     batch_states, batch_actions, batch_paths = [], [], []
3 
4     for _ in range(batch_size):
5         s, path_acts, path_s = self._search_leaf()
6         batch_states.append(s)
7         batch_actions.append(path_acts)
8         batch_paths.append(path_s)
9 
10    for s, path_actions in zip(batch_states, batch_actions):
11        child, goals = self.cube_env.explore_state(s)
12        self.edges[s] = child
13        if np.any(goals):
14            return path_actions + [np.argmax(goals)]
15 
16    values = self._expand_leaves(batch_states)
17    for val, path_states, path_actions in zip(values, batch_paths, batch_actions):
18        self._backup_leaf(path_states, path_actions, val)
19    return None
```

Listing 8.12: Auf den MCTS-Algorithmus folgende Breitensuche [18].

9 Algorithmusentwicklung: A*-Suchalgorithmus

Auch in diesem Kapitel wird ein neuronales Netzwerk trainiert, welches es im Anschluss mit einem Suchalgorithmus zu kombinieren gilt. Statt der zuvor verwendeten Monte-Carlo-Baumsuche wird in diesem Fall die in [22] angewandte A*-Suche zum Einsatz kommen. Das neuronale Netz unterscheidet sich von der Architektur zu dem aus Kapitel 8, da für den A*-Algorithmus nur eine Zustandswertschätzung notwendig ist. Aufgrund dessen muss die Art und Weise des Trainings an den entsprechenden Stellen modifiziert werden.

9.1 Datenrepräsentation

Bei der Umsetzung des A*-Suchalgorithmus in Kombination mit einem neuronalen Netz wird dieselbe Datenrepräsentation verwendet wie schon bei der Monte-Carlo-Baumsuche aus Abschnitt 8.1 auf Seite 133. Die gilt sowohl für die Würfelaktionen als auch die Würfelzustände.

9.2 Neuronales Netz

9.2.1 Architektur

Die Architektur des neuronalen Netzwerkes wurde aus [22] übernommen. In dieser Veröffentlichung setzt die Forschergruppe auf ein Residual Network, welches mit zwei vollverbundenen linearen Schichten (Größe 5.000 und 1.000) beginnt. Darauf folgen vier sogenannte „Residual Blocks“, die jeweils aus zwei versteckten linearen Schichten mit einer Größe von 1.000 bestehen. Der Value wird zuletzt von einer weiteren linearen Schicht

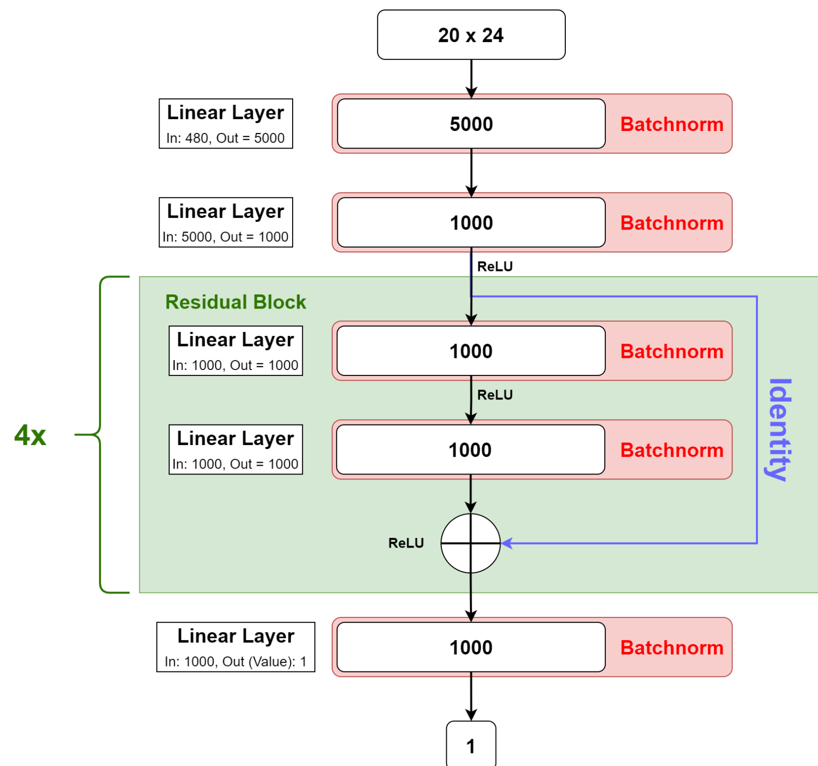


Abbildung 9.1: Architektur des neuronalen Netzwerkes, welches einen Zustand als Eingabe verwendet, um den Gütewert von diesem zu ermitteln.

der Größe 1.000 berechnet. Bei allen Schichten wurde Batch Normalisierung und eine ReLU-Aktivierungsfunktion verwendet. Der Aufbau des Netzwerkes ist in Abbildung 9.1 dargestellt.

Dass hier nur eine Ausgabe für die Güte eines Zustandes berechnet wird, liegt darin begründet, dass der A*-Suchalgorithmus einen Schätzwert benötigt, welcher aussagt, wie weit der ausgewertete Zustand ungefähr vom Zielzustand entfernt ist. Deshalb ist eine Policy-Abzweigung bei dieser Architektur obsolet.

Zielnetzwerk

Die Verwendung eines Zielnetzwerkes, welches eine vorherige Version des neuronalen Netzes nutzt, um Werte zu erhalten, könnte Verbesserungen hinsichtlich Oszillationen und Instabilitäten mit sich bringen [18]. Diese sind nämlich oftmals ein Hinweis auf ein gängiges Reinforcement Learning Problem, das durch Korrelation innerhalb eines Schrittes

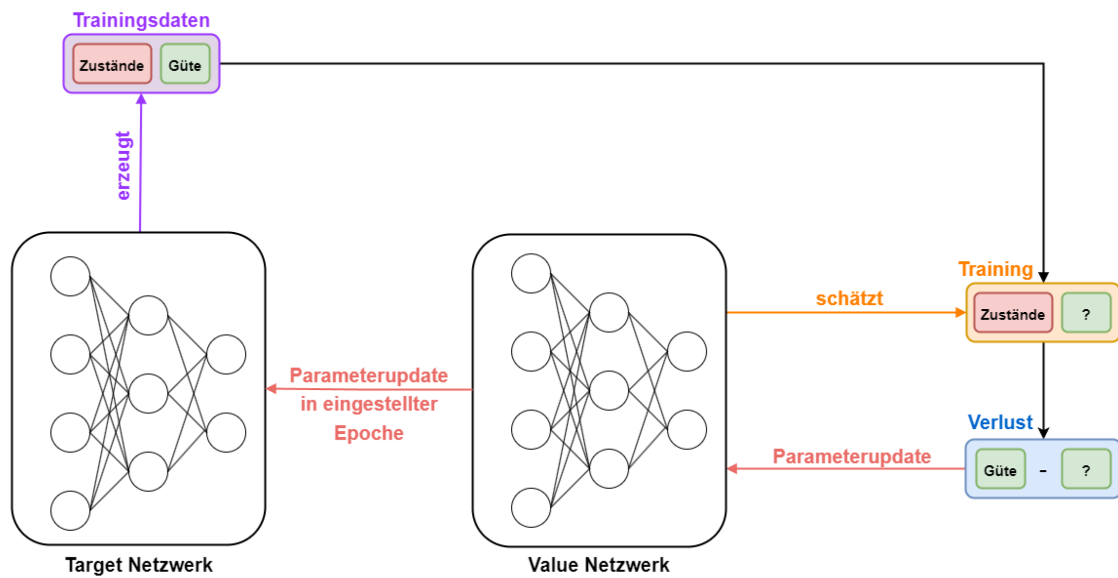


Abbildung 9.2: Grober Aufbau der Architektur bei Verwendung eines zusätzlichen Zielnetzwerkes.

entsteht. Es werden die Schätzungen des Netzwerkes für die Güte eines Zustandes mit den vorigen Schätzungen des Netzwerkes für die Güte der Trainingsdaten überprüft. Vermutlich verwendet das Team um McAleer in Quelle [22] aufgrund dessen ein sogenanntes Ziel- oder Generatornetzwerk, welches dieselbe Architektur aufweist wie das eigentlich zu trainierende Netz, die Parameter jedoch nicht nach jeder Epoche aktualisiert werden. Stattdessen werden dessen Parameter nach einer eingestellten Anzahl von Epochen mit denen des eigentlichen Netzwerkes upgedatet. Dies soll Korrelationen vorbeugen und die erzielten Resultate verbessern, weshalb auch hier ein Zielnetzwerk implementiert wird. Der grobe Aufbau bei der Verwendung eines solchen Zielnetzwerkes ist in Abbildung 9.2 dargestellt.

9.2.2 Hyperparameterauswahl

Auch hier werden, genau wie in dem Kapitel zuvor, die Hyperparameter manuell ausgewählt. Es wird versucht, die angegebenen Parameter aus Artikel [22] zu übernehmen. Aufgrund von aufgetretenen Schwierigkeiten wie beispielsweise hardwarebedingten Einschränkungen und fehlenden Angaben innerhalb der Quelle, war dies jedoch nicht immer möglich.

Aktivierungsfunktion

Als Aktivierungsfunktion wird genau wie in [22] eine ReLU-Funktion verwendet. Es wird nicht genauer erläuert weshalb, es ist aber davon auszugehen, dass diese in Kombination mit den linearen Schichten gute Ergebnisse erzielt. Da diese nicht so ressourcenlastig wie die verwandte eLU-Funktion ist, wird diese auch bei dieser Implementation übernommen.

Optimierer

Die Gruppe um McAleer setzt in [22] auf einen Adam-Optimierer. Da ausgiebige Tests nötig wären, um zu ermitteln, ob es einen potenziell passenderen Optimierer für diese Aufgabe gibt und in Artikel [22] gute Ergebnisse erzielt worden sind, wird auch hier der Adam-Optimierer verwendet.

Lernrate

Quelle [22] gibt keinen Aufschluss darüber, mit welcher Lernrate gearbeitet wurde. Deshalb wird wieder auf das im Abschnitt 2.2.5 auf Seite 36 erläuterte Verfahren zur Lernratenfindung zurückgegriffen, welches in Abbildung 9.3 dargestellt ist.

In dem Graphen ist erkennbar, dass der Konvergierungsprozess und somit die obere Lernratengrenze bei ungefähr $6 \cdot 10^{-5}$ beginnt. Die untere Lerngrenze ist hier ein wenig schwieriger auszumachen. Bei 10^{-4} beginnt die Kurve noch einmal anzusteigen, aber von einer Divergierung kann hierbei noch nicht gesprochen werden. Danach sinkt der Verlust noch einmal bis ungefähr $3 \cdot 10^{-3}$ ab. Sicherheitshalber wird eine Lernrate von 10^{-4} verwendet.

Lernratenscheduler

Im Artikel [22] wurde die Verwendung eines Lernratenschedulers nicht erwähnt. Aufgrund der Argumente, die für den Einsatz einer zyklischen Lernrate sprechen (vgl. Abschnitt 2.2.5), und der guten Ergebnisse, die bei dem Training des neuronalen Netzes für die Monte-Carlo-Baumsuche erzielt wurden, sollte ursprünglich auch hier wieder ein solcher Scheduler eingesetzt werden. Da allerdings ein zweites neuronales Netzwerk genutzt wird,

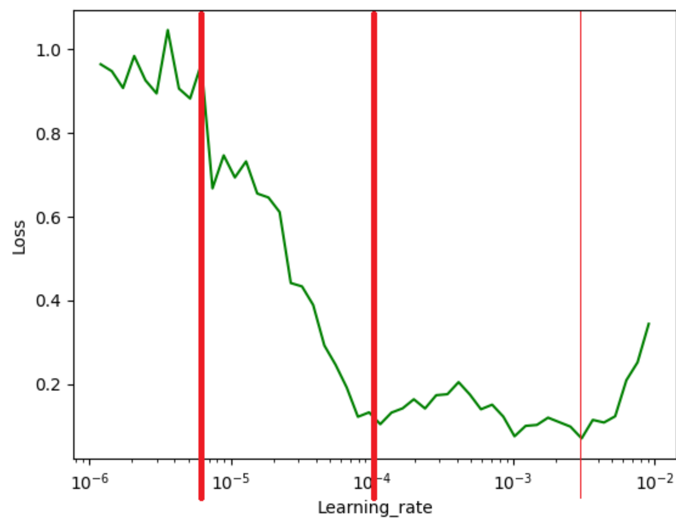


Abbildung 9.3: Gesamtverlust bei iterativer Erhöhung der Lernrate um den Faktor 1.1 für das neuronale Netz mit A*.

um die Trainingsdaten zu labeln, steigt der Verlust jedes Mal an, wenn die Parameter des Generators nach der eingestellten Anzahl an Episoden aktualisiert werden. Durch die abrupte Änderung der Generatorparameter verändert sich auch dessen Ausgabe für die Trainingsdatenlabel und der Verlust beginnt kurzzeitig anzusteigen, bis sich das Netzwerk wieder darauf eingestellt hat. Auch durch das Verändern der Lernrate mit Hilfe eines Schedulers kann der Verlust infolge der zyklischen Lernratenänderung ansteigen. Werden die Parameter beispielsweise zu stark angepasst, kann sich der Unterschied zwischen dem ausgegebenen Wert des Netzes und dem Label der Trainingsdaten vergrößern. Ein Training des Netzwerkes mit diesem Lernratenscheduler hat diesmal keine guten Ergebnisse erzielen können und deshalb wird auf diesen verzichtet. Eine Erklärung für die schlechten Resultate könnte sein, dass die Disparitäten zwischen Label und Output des Netzwerkes einfach zu groß werden, wenn die Generatorparameter upgedatet werden, während die Lernrate verändert wird und der ganze Lernprozess hierdurch möglicherweise ab einem gewissen Zeitpunkt divergiert.

Batchsize

Die Forschungsgruppe verwendet in [22] eine Batchsize von 10.000 Würfelzuständen. Aufgrund von Hardware-Limitationen ist die Implementation mit dieser Größe nicht umsetz-

bar und wird wieder auf ein Zehntel davon (1.000) verringert. Die Limitationen ergeben sich vor allem aus dem gestiegenen Ressourcenbedarf, welcher mit der Komplexität des neuronalen Netzwerkes einhergeht. Würde die Architektur des Netzes angepasst werden, bestände auch die Möglichkeit die Batchsize beizubehalten. Bei stichprobenartigen Versuchen dies umzusetzen, war ein sichtlicher Rückgang des Lernerfolges zu beobachten, weshalb auf die Anpassung der Größe eines Batches gesetzt wurde.

Verlustfunktion

Anders als im Kapitel 8, besitzt das Netzwerk hier nur eine Ausgabe (Value), weshalb auch nur eine Funktion zur Berechnung des Verlustes von Nöten ist. Die Gruppe um McAleer setzt dabei auf eine MSE-Verlustfunktion, welche aufgrund dessen auch hier zum Einsatz kommen soll.

Error Treshold Für die oben erläuterte Aktualisierung des Zielnetzwerkes wird von der Forschungsgruppe aus [22] ein weiterer Hyperparameter eingeführt, der *Error Threshold*. Dieser beschreibt den Wert für den Verlust, der mindestens erreicht werden muss, bevor die Parameter des Zielnetzwerkes, mit denen des eigentlich zu trainierenden Netzwerkes aktualisiert werden. Das soll laut Quelle [22] für eine stabilere, bessere Performance des neuronalen Netzes sorgen. Hierbei verwendet das Forscherteam einen Threshold von $\epsilon = 0,05$. Da der Verlauf der Lernkurve allerdings stark abhängig von den anderen gewählten Hyperparametern ist, ist dieser Wert ohne die eindeutige Angabe aller übrigen verwendeten Parameter nicht sehr aussagekräftig. Soll ein Error Threshold verwendet werden, muss dieser deshalb individuell ermittelt und dementsprechend angepasst werden. Eine Möglichkeit diesen zu ermitteln, wäre ein Probelauf des Trainings, um den Verlust näher untersuchen zu können. Beim Training des Netzes mit solch einem Threshold wurde jedoch beobachtet, dass das Netzwerk im späteren Verlauf nicht mehr so häufig aktualisiert wird wie zuvor (vgl. Abbildung 9.4). Durch die fehlenden Aktualisierungen stagnieren die Modellparameter des Generatornetzwerkes und die Qualität der Trainingsdatenlabel leiden. Dies kann sich auch auf die Performance des eigentlichen Netzwerkes auswirken und dazu führen, dass dieses anschließend schlechter performt als ein Netzwerk, das ohne die Verwendung eines solches Thresholdes trainiert wurde, vorausgesetzt die restlichen Hyperparameter sind so angepasst worden, dass ein stetiger Lernprozess gewährleistet wird. Deshalb wurde bei der Umsetzung auf solch einen zusätzlichen Hyperparameter verzichtet.

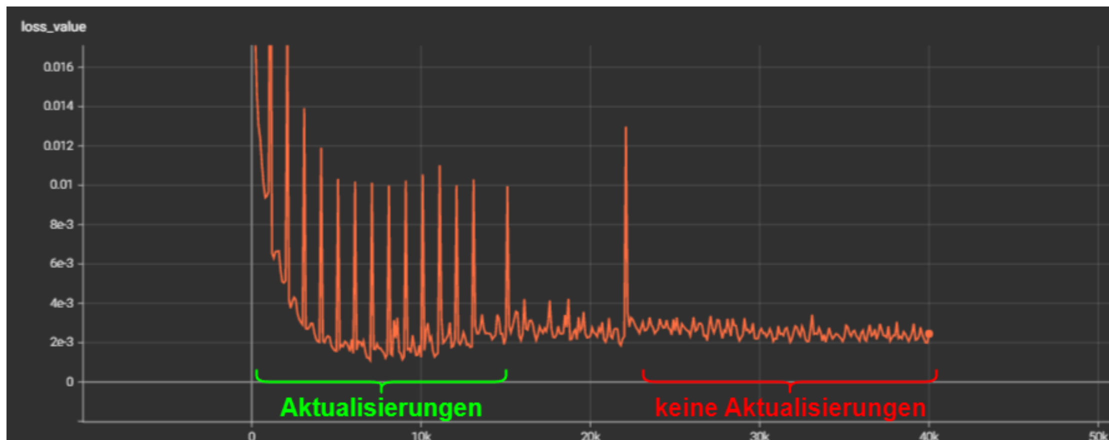


Abbildung 9.4: Lernkurvenverlauf bei dem Training eines neuronalen Netzwerkes unter Verwendung eines Error Thresholdes.

9.3 Training des neuronalen Netzwerkes

Das Training des neuronalen Netzwerkes findet wieder mittels dem in Abschnitt 8.3 erläuterten Verfahren statt. Das trainierte Netzwerk soll dieses Mal allerdings als eine heuritische Funktion verwendet werden, welche ermöglichen soll den ungefähren Abstand vom aktuellen Zustand zum Lösungszustand abzuschätzen, wobei eine Güte $J(s)$ (in Kapitel als $v(s)$ bezeichnet) eines Zustandes als Maß genommen wird. Dies ist notwendig, da der hier verwendete Algorithmus den Lösungspfad ermittelt, indem dieser immer wieder den Knotenpunkt, welcher die niedrigsten Übergangskosten besitzt, als aktuellen Zustand auswählt. Die Kosten eines Knotenpunktes innerhalb des Verfahrens werden mit Hilfe folgender Formel berechnet:

$$f(x) = g(x) + h(x) \tag{9.1}$$

Hierbei stellt $g(x)$ die Kosten der Transition zwischen zwei Knotenpunkten und $h(x)$ die heuristische Funktion, welche von dem neuronalen Netz übernommen wird, dar. In [22] wird beschrieben, dass $g(x)$ bei jedem Übergang einen Wert von 1 besitzt, da jeder Folgezustand von dem Zustand zuvor eine Aktion entfernt ist. $h(x)$ wird vom Netzwerk ausgegeben, indem es die Güte von Zuständen abschätzt, welche diesem vorher mittels dem ADI-Verfahren folgendermaßen antrainiert wird:

$$h(x) = \begin{cases} 0, & \text{wenn } s \text{ der Zielzustand ist} \\ 1, & \text{wenn } s \text{ eine Aktion vom Zielzustand entfernt ist} \\ J(A(s,a)), & \text{sonst} \end{cases} \tag{9.2}$$

Wie in Kapitel 8 wird auch hier einem Zustand der Wert 0 zugewiesen, wenn dieser bereits dem Zielzustand entspricht. Andernfalls ergibt sich die Güte aus der Ausgabe des Netzwerkes für diesen. Zusätzlich werden hier jedoch noch Zustände mit 1 gewertet, welche sich nur eine Aktion vom Lösungszustand entfernt befinden. Das soll das Propagieren der Güte vom Zielzustand zu den Folgezuständen fördern und ein besseres Abschätzen der Entfernung ermöglichen. Zustände werden hier mit dem niedrigsten Gütewert ihrer Folgezustände gelabelt, im Gegensatz zum Maximalwert, wie in Kapitel 8, um den Weg mit den niedrigsten Kosten zu finden, welcher im Idealfall den Lösungspfad darstellt. Die Formel für das Labeln eines Zustandes lautet hierbei

$$J'(s) = \min_a(g^a(s, A(s,a)) + J(A(s,a))) \quad , \tag{9.3}$$

wobei $g^a(s, A(s,a))$ die Kosten einer Transition ist und somit $g(x)$ entspricht, welche immer mit 1 gewertet wird, und $J(A(s,a))$ die Ausgabe des Netzwerkes darstellt. Die Anpassungen sind dem Listing 9.1 zu entnehmen (im Vergleich zu Listing 8.4). Des Weiteren fehlt noch die Implementation des Zielnetzwerkes, dessen Nutzen im Abschnitt zuvor erläutert wurde. Dafür wird zunächst ein weiteres neuronales Netz erzeugt, welches die Initialwerte des eigentlichen Netzes übernimmt (vgl. Listing 9.2). Anschließend müssen die Parameter des Zielnetzwerkes in einer voreingestellten Periode denen des eigentlichen Netzwerkes angepasst werden (vgl. Listing 9.3).

9.4 Anwendung des trainierten Netzwerkes

Das trainierte neuronale Netz soll als Heuristik in einem A*-Algorithmus verwendet werden, um die Distanz zwischen einem Zustand und dem Lösungszustand abzuschätzen. Hierbei wird die in McAleer et al. Artikel [22] vorgestelle Variante „Batch weighted A* search“ genutzt, welche nicht nur einen Knoten untersucht, sondern mehrere gleichzeitig. Durch Hinzufügen einer zusätzlichen Variable λ kann der Einfluss der Distanz $g(x)$ gewichtet werden und es ergibt sich folgende Anpassung der Kostenfunktion aus Gleichung 9.1:

```python
data = random.sample(scramble_buffer, batch_size)

depths, states, explored_states, goals, is_goal = zip(*data)

# Handle explored states
explored_states = np.stack(explored_states)
shape = explored_states.shape
explored_states_t = torch.tensor(explored_states,
                                 dtype= torch.float32).to(device)
explored_states_t = explored_states_t.view(shape[0]*shape[1], *shape[2:])
value_t = net(explored_states_t)
value_t = value_t.squeeze(-1).view(shape[0], shape[1])

# Create tensor from goal array
goals_mask_t = torch.tensor(goals, dtype=torch.int8).to(device)

# Mask to find goal states that are one turn away
goals_mask_t_inv = np.logical_not(goals).astype(int)
goals_mask_t_inv = torch.tensor(goals_mask_t_inv, dtype=torch.int8).to(device)

# Set states that are one turn away to 0
value_t *= goals_mask_t_inv

# Add rewards/punishments
value_t  += goals_mask_t_inv

# Set states that are one turn away to 1
value_t += goals_mask_t

# Mask to find states that are already the goal state
is_goals_mask_t_inv = np.logical_not(is_goal).astype(int)
is_goals_mask_t_inv = torch.tensor(is_goals_mask_t_inv,
                                   dtype=torch.int8).to(device)

# Label current state with the smallest value of its explored states
val_t,_ = value_t.min(dim=1)
# Label with value '0' if state is goal_state
val_t *= is_goals_mask_t_inv

# Return data
enc_input = np.stack(states)
enc_input_t = torch.tensor(enc_input, dtype=torch.float32).to(device)
depths_t = torch.tensor(depths, dtype=torch.float32).to(device)
weights_t = 1/depths_t
return enc_input_t.detach(), weights_t.detach(), val_t.detach()
```

Listing 9.1: Labeln der erzeugten Daten aus Listing 8.3 mit entsprechenden Anpassungen für den A*-Algorithmus.

```python
# Actual network to be trained
net = ResCubeNet()
# Target network
generator = ResCubeNet()
generator.load_state_dict(net.state_dict())
```

Listing 9.2: Erzeugung eines Zielnetzwerkes für das Labeln der Trainingsdaten in Python.

```
1 if step_idx % update_period == 0:
2
3 with torch.no_grad():
4     generator = update_generator(generator, net, 1, device)
```

Listing 9.3: Aktualisieren der Zielnetzwerkparameter in Python.

$$f(x) = \lambda g(x) + h(x) \tag{9.4}$$

Diese Änderung ermöglicht es die Qualität der Lösung und die Dauer der Suche zu beeinflussen [22]. λ kann einen Wert zwischen null und eins annehmen, wobei mit steigendem λ die Lösungslänge kürzer und die Lösungsdauer länger wird. Auch die Anzahl N der gleichzeitig untersuchten Knoten hat Auswirkungen auf Qualität und Dauer des Suchverfahrens. Je mehr Knoten überprüft werden, desto kürzer die Lösung, jedoch wieder umso länger der Zeitraum, bis diese gefunden wurden. Das Funktionsprinzip des A*-Algorithmus wurde im Grundlagenteil 3.2 auf Seite 80 bereits ausführlich erläutert und es gilt, dieses in Programmcode umzusetzen. Als Grundgerüst wird Code von Nicolas Swift[1] genutzt und den Anforderungen entsprechend angepasst. Dafür wird zunächst eine Klasse für die Knoten erzeugt, welche für die Suche notwendig sind. Diese beinhalten folgende Attribute:

- Zustand: Der Würfelzustand des untersuchten Knotenpunktes.
- Elternknoten: Verbindung zum vorrangehenden Knoten.
- Tiefe $g(x)$: Distanz vom Startpunkt.
- Heuristik $f(x)$: Abschätzung bis zum Zielpunkt.
- Aktion: Transformation, die zu diesem Zustand geführt hat.

Innerhalb des Suchalgorithmus werden diese Knotenpunkte genutzt, um den Lösungszustand zu ermitteln. Dafür werden zwei Datenstrukturen angelegt, die Aufschluss darüber geben, welche Knoten bereits untersucht wurden („closed“) und welche es noch zu überprüfen gilt („open“) (vgl. Listing 9.4). Hierbei ist „closed“ vom Typ Dictionary, damit später noch über die zugehörigen Keys auf die entsprechenden Knoten zugegriffen werden können. Die „open“-Liste wird vor jedem neuen Durchlauf der Suche in Bezug auf die

[1] https://gist.github.com/Nicholas-Swift/003e1932ef2804bebef2710527008f44 - Zugriffsdatum: 07.07.23

```python
# list to store nodes we want to explore
open = []
# dictionary to store nodes which already been visited
closed = {}
```

Listing 9.4: Strukturen, welche Aufschluss über die bereits untersuchten Knoten geben.

geringsten Kosten (vgl. Gleichung 9.4) sortiert. Anschließend werden die N besten Knoten ausgewählt und näher untersucht. Dafür werden deren Folgezustände mit Hilfe der zugehörigen Funktion als erstes darauf geprüft, ob sich der Zielzustand unter einem von diesen befindet. Ist dies der Fall, ist die Suche abgeschlossen und über die Variablen, welche die Aktionen der jeweiligen Knoten speichern, die zu diesem geführt haben, kann der Lösungsweg zurückverfolgt werden (vgl. Listing 9.5). Wurde die Lösung nicht gefunden,

```python
while True:
    move = goal_node.move
    if move != -1:
        solution_path.append(Action(move))
        goal_node = goal_node.parent
    else:
        break
solution_path.reverse()
return solution_path
```

Listing 9.5: Ermitteln des Lösungspfades über die Knotenvariable für die zugehörige Aktion, wobei -1 für den Startpunkt steht.

werden die untersuchten Knoten dem „closed"-Dictionairy hinzugefügt und neue Knoten für deren Folgezustände angelegt. Diese werden anschließend dahingehend überprüft, ob sich bereits andere Knotenpunkte mit demselben Würfelzustand in „closed" befinden. Trifft diese Aussage zu, wird kontrolliert, welcher von beiden Knoten die geringeren Kosten besitzt. Derjenige bleibt in „closed" hinterlegt, während der andere verworfen wird. Alle anderen neuen Knoten, bestehend aus den Folgezuständen, werden mit Hilfe des neuronalen Netzes auf ihren Abstand zur Lösung hin untersucht und anschließend in „open" hinterlegt (vgl. Listing 9.6). Dieses Vorhaben wird wiederholt, bis der Zielzustand ermittelt oder die eingestellte maximale Anzahl an Iterationen überschritten wurde.

```python
for i in range(len(nodes_to_open)):
    # use formula: f(x) = h(x) + lambda * g(x)
    nodes_to_open[i].f = heuristics[i].numpy() + (weight * nodes_to_open[i].depth)
# add nodes to open list
open.extend(nodes_to_open)
```

Listing 9.6: Untersuchen der Knotenpunkte mittels neuronalem Netz.

10 Evaluation

In diesem Kapitel soll überprüft werden, ob die in Kapitel 4 ab Seite 90 gestellten Anforderungen erfüllt werden konnten. Dafür werden entsprechende Versuche durchgeführt, mit denen sich der jeweilige Erfüllungsstatus der Aufgaben, welche eine Verifizierung erfordern, untersuchen lässt. Dies gilt vor allem für die Einhaltung der Randbedingungen, die an die beiden Algorithmen und an die benutzerbasierte Anwendung gestellt wurden.

10.1 Evaluierung der Software: Suchalgorithmen

Zunächst gilt es, die beiden implementierten Algorithmen auf ihre Performance zu testen. Dabei sind insbesondere Laufzeit, Erfolgsquote und die Länge des Lösungsweges von Interesse, da sich die A*- und MCTS-Methode in diesen Punkten am besten miteinander vergleichen lassen. Ein weiterer untersuchenswerter Aspekt ist die Anzahl der Iterationen, die benötigt wurden, bis eine Lösung ermittelt werden konnte. Da das A*-Verfahren jedoch während der Suche mit Batches arbeitet und mehrere vielversprechende Knotenpunkte gleichzeitig untersucht, benötigt diese Methode viel weniger Durchläufe als jene mit MCTS, weshalb ein Vergleich hier unter Umständen weniger zielführend ist. In Lapans Implementation [18] ist auch eine Monte-Carlo-Baumsuche mit Batchverarbeitung vorzufinden, welche laut seiner eigenen Auswertungen allerdings weniger oder gleich erfolgreich gewesen ist als das Suchverfahren ohne Batches. Möglicherweise bietet es sich dennoch an, dieses in einem eigenen Experiment zu überprüfen und mit der eigenen Implementation ohne Stapelverarbeitung zu vergleichen.

Aufgrund dessen, dass Lapan auch vorrangig mit dem Pocketcube experimentiert hat und seinen dazu vorliegenden Testergebnissen, ist es sinnvoll, diese als Referenz heranzuziehen und die eigenen Resultate mit seinen abzugleichen. Die Leistung des A*-Algorithmus zu beurteilen gestaltet sich etwas schwieriger, da keine Referenzen, in denen mit einem Pocketcube experimentiert wurde, vorhanden sind. Die Gruppe um McAleer [22] befasste

sich in ihrer Ausarbeitung lediglich mit dem Rubik's Cube und anderen Permutationspuzzeln, sodass Vergleiche zwischen den dort gewonnen Ergebnissen und den hier ermittelten Resultaten nicht sehr aussagekräftig wären. Deshalb wird, wie bereits weiter oben erwähnt, ein Vergleich zur MCTS-Methode vorgenommen unter den Gesichtspunkten, in welchen sich die Algorithmen gleichen.

10.1.1 MCTS

Über ein entworfenes Programm in Python wird die Implementation des MCTS-Algorithmus in Verbindung mit einem neuronalen Netzwerk systematisch untersucht. Dafür wird dem System ein zufälliger Würfelzustand übergeben und anschließend werden die relevanten Kriterien wie Lösungsdauer, ob der Zustand gelöst werden konnte, die Länge des Lösungsweges und die Anzahl der benötigten Iterationen notiert. Die Schwierigkeit der Zustände wird hierbei schrittweise erhöht, indem die Anzahl der Verdrehungen vom Startzustand iterativ vergrößert wird. Pro Verdrehungstiefe wird eine festgelegte Anzahl von Zuständen erzeugt, die es zu lösen gilt, bevor die Schwierigkeitsstufe weiter zunimmt. Lapan verwendet in seinen Experimenten eine Verdrehungstiefe von maximal 50 Rotationen, welche dem System jeweils 20-mal übergeben werden. Damit ergeben sich ingesamt 1.000 unterschiedliche Würfelzustände pro Versuchsdurchführung. Eine höhere Anzahl von Tests pro Schwierigkeitsgrad wäre in den Ergebnissen aussagekräftiger und würde weniger von Ausreißern beeinflusst werden, jedoch erhöht sich die benötigte Durchführungszeit drastisch. Benötigt eine Suche pro Würfelzustand beispielsweise zwei Minuten, ergibt sich eine Rechenzeit von $2\,\text{Minuten} \cdot 50\,\text{Verdrehungen} \cdot 20\,\text{Iterationen} = 2.000\,\text{Minuten}$, was fast anderthalb Tagen entspricht. Für Vergleichszwecke und um den zeitlichen Rahmen nicht zu überschreiten, werden hier deshalb dieselben Parameter verwendet wie bei Lapan [18].

Lösungsdauer

Ein wichtiges Kriterium für die Beurteilung der implementierten Methodiken ist die benötigte Zeit, bis eine Lösung gefunden wird. Die Laufzeit eines Algorithmus gibt direkt Aufschluss über seine Effizienz. Es ist ein ansehnliches Ergebnis, wenn ein System in der Lage ist, einen Zauberwürfel mit hundertprozentiger Erfolgsquote zu lösen, jedoch wenig praktikabel, benötigt dieser dafür mehrere Stunden. Soll das Verfahren in einer interaktiven Anwendung eingesetzt werden, ist das möglichst zeitnahe Finden einer Lösung

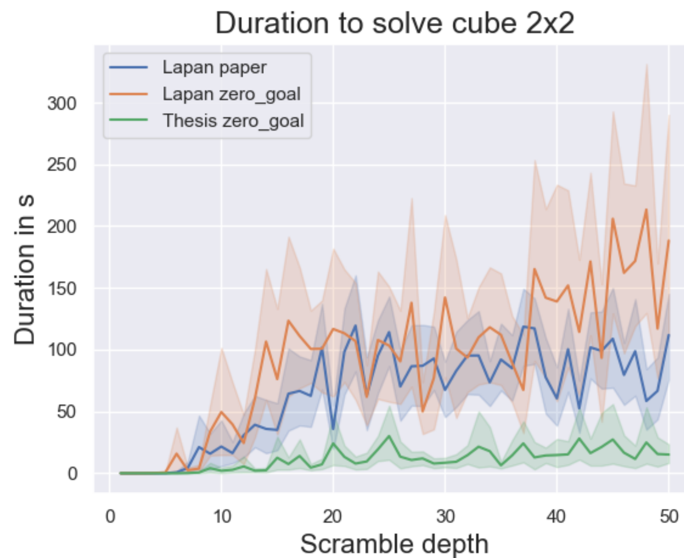


Abbildung 10.1: Durchschnittliche Laufzeiten des MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen [18].

ein Kernbestandteil der Benutzererfahrung. In den Anforderungen in Kapitel 4 wurde deshalb ein maximales Zeitlimit von fünf Minuten angesetzt (S-NF1), die ein gesamter Suchdurchlauf benötigen darf, bevor der Würfelzustand als für das angewandte Verfahren nicht lösbar deklariert wird. In Abbildung 10.1 sind die durchschnittlichen Laufzeiten inklusive Konfidenzintervall dargestellt, die der MCTS-Algorithmus in Kombination mit einem neuronalen Netzwerk pro Verdrehungstiefe benötigt. Hierbei wird in allen Darstellungen ein Konfidenzintervall von 95% verwendet.

In orange ist die benötigte Zeit des von Lapan implementierten Verfahrens abgebildet, welches sich an das Vorbild der in Artikel [21] vorgestellten Methodik hält. In blau ist hingegen das von ihm modifizierte Verfahren dargestellt, das beim Training des neuronalen Netzes eine veränderte Berechnung des Zielwertes für den Lösungszustand beinhaltete (nachlesbar in Abschnitt 8.3), und auf welches sich auch die hier vorgenommene Implementation (in grün dargestellt) stützt. In Abbildung 10.1 ist eine eindeutige Verbesserung der Laufzeiten zu erkennen, sowohl im Vergleich zur eigentlichen Methode wie auch zu der von Lapan modifizierten. Obwohl nur einige wenige Verbesserungen an dem neuronalen Netz (Gewichtsinitialisierung und Normalisierungsschichten) und am Training selbst (zyklischer Lernratenscheduler, zusätzlicher Entropieverlust und kleinere Batchsize) vorgenommen wurden, hatte dies offensichtlich deutliche Auswirkungen auf die schlussend-

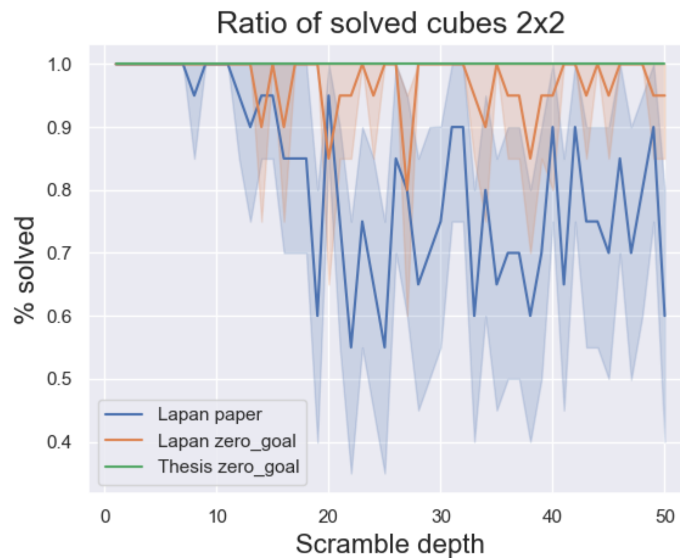


Abbildung 10.2: Erfolgsquote des MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen [18].

liche Effizienz des neuronalen Netzwerkes. Der Grafik ist zu entnehmen, dass Lapans Implementationen ab einer Tiefe von ungefähr 15 Rotationen durchschnittlich 100 Sekunden beziehungsweise anderthalb Minuten pro Suche benötigt, während hier im Mittel maximal 10 Sekunden pro Zustand nötig sind. Da der verwendete MCTS-Algorithmus zum Großteil von Lapan übernommen wurde, liegt die Steigerung der Rechengeschwindigkeit mit hoher Wahrscheinlichkeit an den vorgenommenen Optimierungen am neuronalen Netzwerk und denen, die an dem Training getätigt wurden, begründet.

Erfolgsquote

Der wohl wichtigste Gesichtspunkt, unter welchem die Methodiken verglichen werden sollten, ist die Erfolgsquote, mit welcher der Algorithmus in der Lage ist, zufällige Würfelzustände zu lösen. Sie gibt Aufschluss darüber, wie zuverlässig das Verfahren angesichts neuer unbekannter Würfelzustände agiert und ob es praktisch einsetzbar ist für eine benutzerbasierte Anwendung. Abbildung 10.2 ist der Anteil der gelösten Würfel pro Verdrehungstiefe im Vergleich zu Lapans Resultaten zu entnehmen.

Obwohl Lapans System mit einer Erfolgsquote von 90% schon sehr zuverlässig unbekannte Würfelzustände löst, konnte mit dieser Implementation noch eine Performancesteigerung

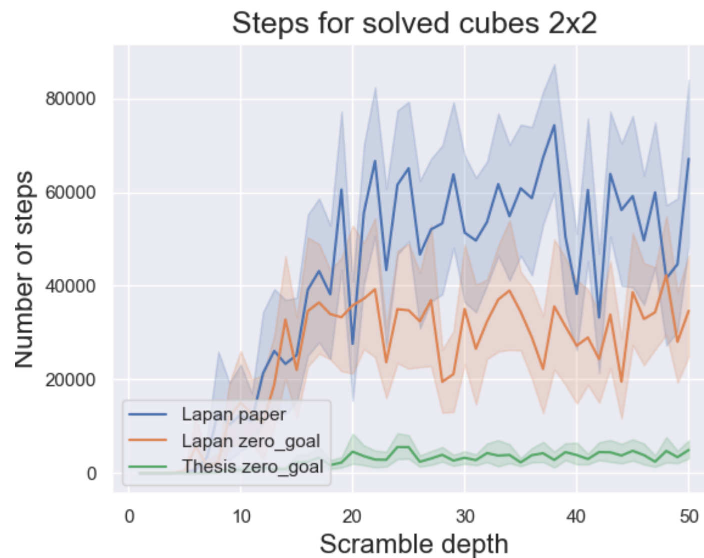


Abbildung 10.3: Anzahl der benötigten Schritte pro Suche mittels MCTS-Verfahren im Vergleich zu Lapans Ergebnissen [18].

erzielt werden. Jeder übergebene Zustand konnte erfolgreich gelöst werden. Dies spricht für die am System vorgenommenen Optimierungen, welche mit hoher Wahrscheinlichkeit wieder maßgeblich entscheidend für die erbrachten Resultate sind.

Iterationen pro Suchdurchlauf

Die benötigten Iterationen pro Suchdurchlauf geben vor allem Aufschluss über die Unterschiede in der Performance. Sie stehen fast in direkter Relation zu der Laufzeit, weshalb auch hier Besserungen, also weniger Durchläufe pro Suche, zu erwarten sind. Den Erwartungen entsprechend spiegelt sich dies deutlich in Abbildung 10.3 wieder. Wo Lapans Monte-Carlo-Baumsuche im Durchschnitt 30.000 Schritte zur Lösungsfindung benötigt, kommt dieses Suchverfahren mit einem Drittel davon aus. Auch dies liegt mit großer Sicherheit in den bereits genannten Verbesserungen begründet.

Länge des Lösungsweges

Eine weitere in Kapitel 4 gestellte Anforderung beinhaltet die Optimalität des gefundenen Lösungsweges (S-F8). Ein Pfad gilt als optimal, wenn dieser kürzer oder gleich der *Gottes*

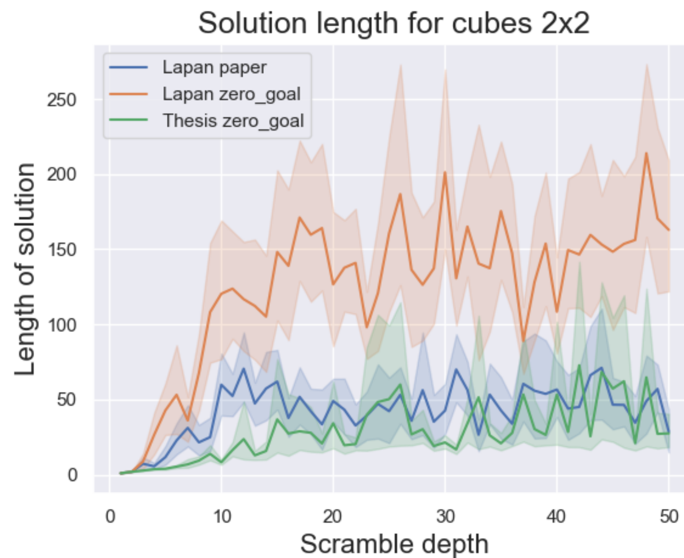


Abbildung 10.4: Lösungsweglängen des MCTS-Verfahrens im Vergleich zu Lapans Ergebnissen [18].

Zahl für den entsprechenden Zauberwürfel ist. Es ist wenig praktikabel, wenn ein System in der Lage ist eine Lösung innerhalb kürzester Zeit zu finden, allerdings die Lösungsweglänge beispielsweise ≥ 100 ist. Für eine benutzerbasierte Anwendung, die mit einer angeschlossenen Hardware wie der aus Kapitel 5 arbeiten soll, sind zu lange Lösungswege nicht zweckmäßig und für die reibungslose Benutzererfahrung eines/einer Anwender:in hinderlich. Abbildung 10.4 vergleicht die Längen der ermittelten Lösungswege von Lapans Verfahren mit dem hier vorgestellten. Das System, welches mit einem neuronalen Netzwerk arbeitet, das ohne Modifikationen an der Zielwertberechnung trainiert wurde, besitzt hierbei Lösungslängen ≥ 100, die wie bereits angesprochen nicht sehr praktikabel sind. Die modifizierten Varianten ermöglichen bereits deutliche Verbesserungen und halbieren durchschnittlich die Gesamtzahl der benötigten Rotationen pro Lösung, befinden sich aber immer noch oberhalb der angesprochenen Optimalität, welche bei dem Pocketcube einer Anzahl von 14 Rotationen oder weniger entspricht. Obwohl der Zielzustand schneller und auch zuverlässiger gefunden wird, befinden sich höchstwahrscheinlich unnötige Aktionsabfolgen wie zum Beispiel vier aufeinanderfolgende identische Verdrehungen im Lösungspfad und mindern die Qualität der Lösung.

Eine anschließende Breitensuche könnte Abhilfe schaffen und diese redundanten Bewegungsfolgen entfernen, allerdings auf Kosten einer verlängerten Laufzeit. Lapan notierte

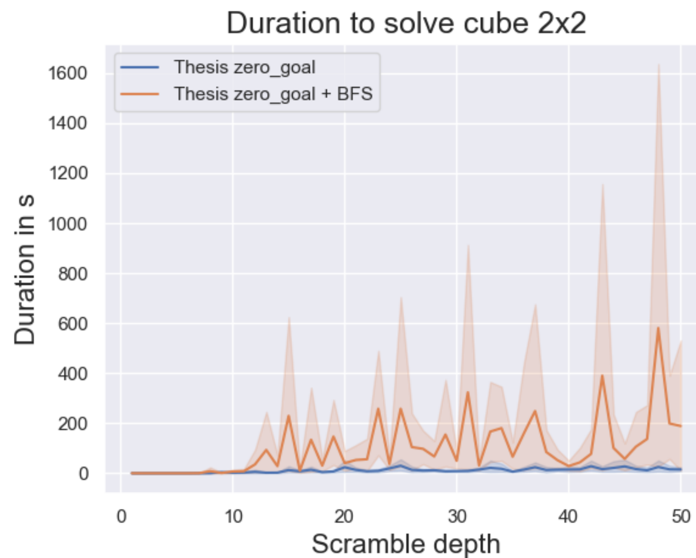


Abbildung 10.5: Vergleich der erforderlichen Zeit pro Suchdurchlauf zwischen dem MCTS-Verfahren mit und ohne anschließender Breitensuche.

in seinen Versuchen nicht, wie viel länger ein gesamter Durchlauf an Zeit benötigen würde, wird zusätzlich am Ende eine Breitensuche durchgeführt. Es ist für die Beurteilung des Verfahrens mit BFS nämlich durchaus relevant, ob sich die erforderliche Zeit merklich vergrößert und die Effizienz darunter leidet. Für die Benutzererfahrung von Anwender:innen macht es einen großen Unterschied, ob ein Durchlauf wenige Sekunden oder mehrere Minuten in Anspruch nimmt. In Abbildung 10.5 ist der Vergleich der durchschnittlichen Zeitspanne pro Würfelzustand je nach Tiefe der Verdrehungen zwischen der hier entworfenen Methode mit und ohne BFS abgebildet.

Aus dem Graphen wird ersichtlich, dass eine Kombination von der Monte-Carlo-Baumsuche mit anschließender Breitensuche teilweise extreme zeitliche Ausreißer besitzt. Die längste Zeit, die eine Breitensuche im Anschluss benötigte, beträgt 10513 Sekunden und damit fast drei Stunden. Nichtsdestotrotz besitzt das Verfahren mit BFS den Vorteil, dass vor allem bei den Würfelzuständen, welche mit Rotationen ≥ 10 erzeugt wurden, kürzere Lösungsweglängen erzielt werden konnten (vgl. Abbildung 10.6). Zuvor waren dort starke Schwankungen in den Lösungspfadlängen zu vermerken, welche sich durch die angewandte Breitensuche stark reduzierten. Die Anzahl der benötigten Rotationen, um aus einem beliebigen Würfelzustand zur Lösung zu gelangen, beträgt im Mittel weniger als zwölf Drehungen und ist somit im Schnitt optimal. Dennoch bleibt es fraglich,

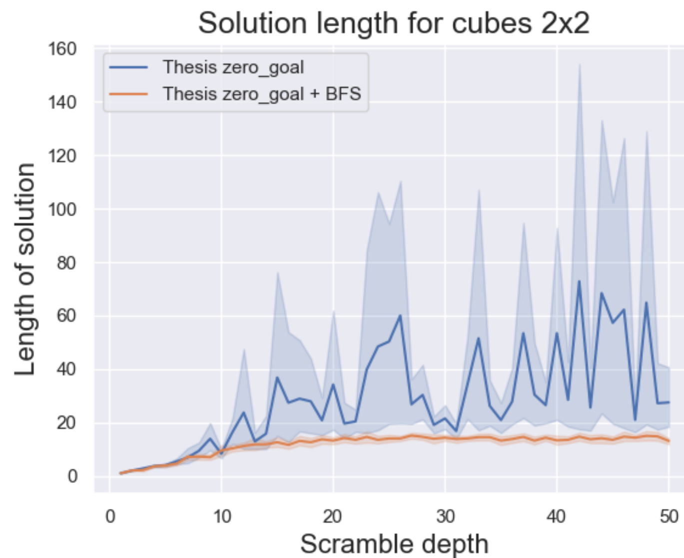
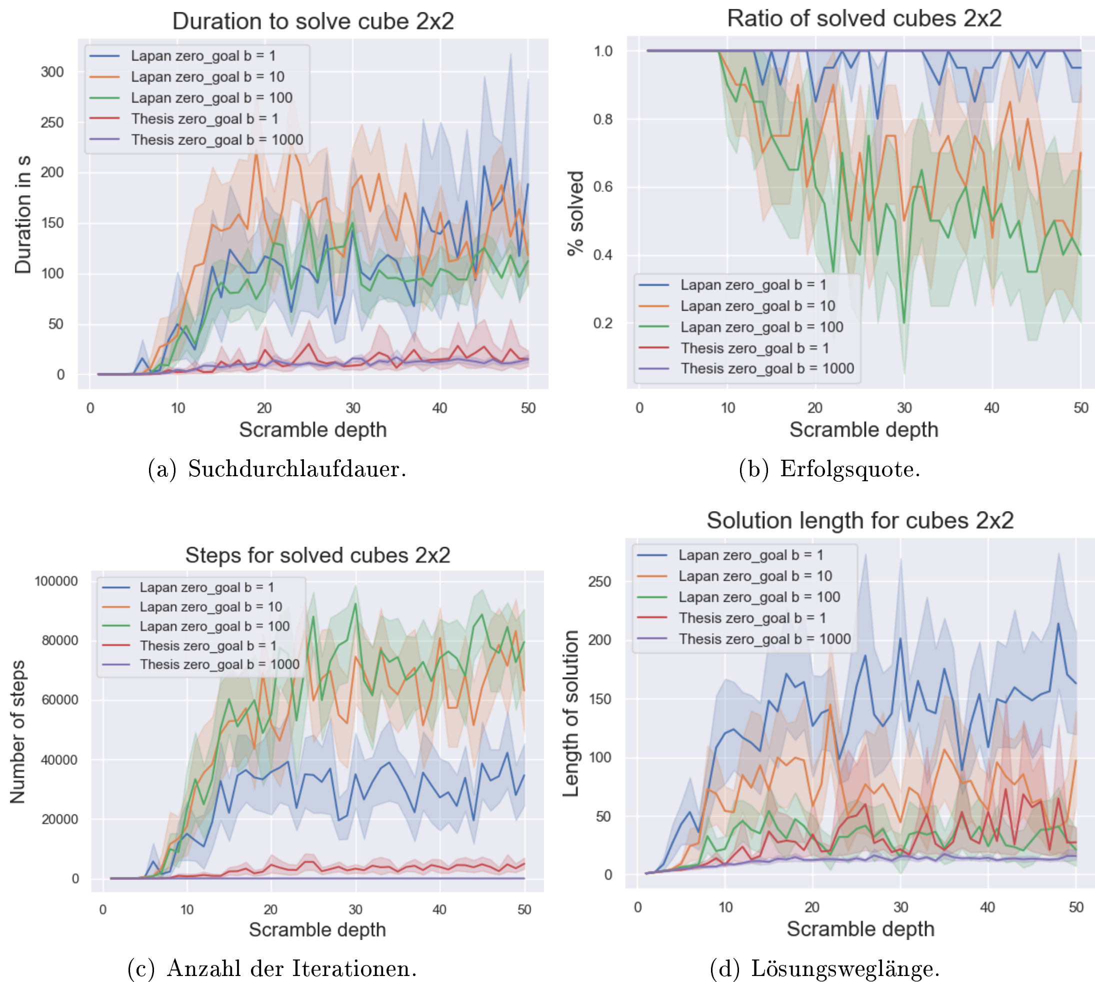


Abbildung 10.6: Vergleich der Lösungsweglängen zwischen dem MCTS-Verfahren mit und ohne anschließender Breitensuche.

ob es sich lohnt, einen solch starken Kompromiss zwischen Effizienz und Optimalität einzugehen. Ein möglicher Mittelweg wäre, gegebenenfalls die Breitensuche zeitlich zu begrenzen, um die Nutzererfahrung einer Software zu erhalten.

10.1.2 Batched MCTS

Um eine bessere Grundlage für den Vergleich zwischen MCTS- und A*-Verfahren zu erhalten, bietet es sich an die Monte-Carlo-Baumsuche ebenfalls mit einer Batchverarbeitung durchzuführen. Lapan besitzt in seinem Programmcode [18] dafür bereits eine entsprechende Funktion, die für diese Versuche genutzt werden soll. In seiner Dokumentation befinden sich zudem Vergleiche zwischen einer Einzel- und einer Batchverarbeitungsmethode, welche mit unterschiedlich vielen Knotenpunkten pro Batch getestet wurden. Den Erwartungen entgegen verschlechterte sich bei Lapans Versuchsdurchführungen die Effizienz und Erfolgsquote des MCTS-Verfahrens mit zunehmender Batchgröße in einigen Bereichen deutlich. Da in den hier durchgeführten Experimenten bereits in fast allen Kritikpunkten erwähnenswerte Verbesserungen erzielt werden konnten, lohnt es sich auch einen weiteren Versuch mit einer Monte-Carlo-Baumsuche, welche eine Batchverarbeitung inkorporiert, durchzuführen und Lapans Resultate zu verifizieren. In Abbildung 10.7 sind alle im Abschnitt zuvor behandelten Bewertungskriterien übersichtlich für ein



(a) Suchdurchlaufdauer.

(b) Erfolgsquote.

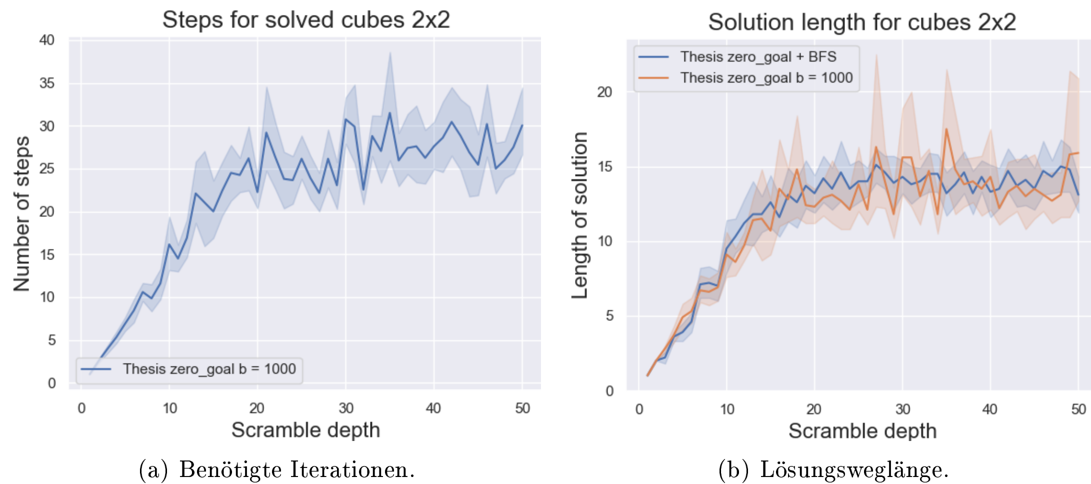
(c) Anzahl der Iterationen.

(d) Lösungsweglänge.

Abbildung 10.7: Vergleich zwischen den von Lapan implementierten [18] und den hier entworfenen darauf aufbauenden Methoden mit und ohne Batchverarbeitung.

MCTS-Suchverfahren mit Stapelverarbeitung dargestellt. Dabei ist nicht nur der Vergleich zu Lapans Experimenten von Interesse, sondern auch zu der hier ausgearbeiteten Methode mit Einzelverarbeitung. Es wäre durchaus denkbar, dass das Verfahren, welches *gleichzeitig* mehrere Knotenpunkte untersucht, bessere Ergebnisse erzielt, als die von Lapan implementierten, aber nicht zwangsläufig besser ist, als das hier entworfene, welches vielversprechende Knotenpunkte einzeln verarbeitet.

In den Grafiken in Abbildung 10.7 wird ersichtlich, dass sich die Resultate der Implementationen von Lapan mit zunehmender Batchgröße teilweise sogar verschlechtern. Wäh-



(a) Benötigte Iterationen.

(b) Lösungsweglänge.

Abbildung 10.8: MCTS-Verfahren mit einer Batchverarbeitung der Größe $b = 1.000$ im Detail.

rend die benötigte Dauer eines Suchdurchlaufs im Vergleich zur Einzelverarbeitung ungefähr gleich bleibt, steigt die Anzahl der benötigten Iterationen entgegen den Erwartungen sogar an, obwohl diese beiden Aspekte in fast direkter Relation zueinander stehen. Auch die gefundene Lösungsweglänge verbessert sich mit der Anzahl der *gleichzeitig* untersuchten Knotenpunkte. Die große Schwachstelle ist bei Lapans Implementationen, welche eine Batchverarbeitung beinhalten, aber vor allem die erbrachte Erfolgsquote. Wo vorher eine Lösung mit $\geq 90\,\%$ Erfolgsrate gefunden werden konnte, verbleiben nun noch ungefähr $60\,\%$. Dies macht die Verfahren weniger zielführend als zuvor. Die hier implementierte Einzelverarbeitungsmethode unterscheidet sich in Suchdurchlaufsdauer und Erfolgsquote kaum von ihrem Pendant mit Stapelverarbeitung, außer dass die benötigte Zeit pro Zustand sich mehr stabilisiert und weniger Ausreißer vorzufinden sind. Die Anzahl der benötigten Iterationen sinkt erwartungsgemäß, da sich die Gesamtanzahl der überprüften Knoten pro Suche sehr wahrscheinlich nicht verändert und somit pro Durchlauf einfach mehr Knotenpunkte untersucht werden. Die Ergebnisse sind in Grafik 10.8(a) abgebildet.

Unerwarteterweise sinkt mit steigender Batchgröße die Lösungsweglänge im Durchschnitt auf ≤ 12 Rotationen bis der Zielzustand aus einem beliebigen Würfelzustand erreicht wird. Die Länge ist im Mittel somit optimal. Da sich sogar die benötigte Dauer im Gegensatz zur Einzelverarbeitungsmethode minimal verbessert und somit in kürzerer Zeit eine optimale Lösung gefunden werden kann, macht dies im Umkehrschluss eine

anschließende Breitensuche obsolet und das entworfene Verfahren profitiert immens durch die Verwendung einer Stapelverarbeitung (vgl. Abbildung 10.8(b)).

10.1.3 A*-Algorithmus

Dadurch, dass das MCTS-Verfahren erfolgreich mit einer Stapelverarbeitung durchgeführt werden kann, erleichtert dies einen Vergleich mit der Methode, welche den A*-Algorithmus inkorporiert. Die Forschungsgruppe um McAleer nutzt in ihrem Artikel zum *DeepcubeA* [22] eine Batchgröße von $b = 10.000$ für den Rubik's Cube. Da der Pocketcube einen viel kleineren Suchraum besitzt als sein größeres Gegenstück, wird hier eine Batchgröße von $b = 1.000$ mehr als ausreichend sein und bei beiden untersuchten Verfahren verwendet werden. Das System mit dem A*-Algorithmus besitzt zudem einen weiteren Hyperparameter λ, welcher die Gewichtung der Distanz regelt und somit Einfluss auf die Effizienz und Optimalität der gefundenen Lösungen hat (vgl. Gleichung 9.4 auf Seite 161). Je kleiner λ hierbei gewählt wird, desto weniger Speicher soll verbraucht werden, aber desto länger sind die potenziell gefundenen Lösungswege [22]. McAleer und sein Team setzen hierbei einen Wert von $\lambda = 0,6$ an, welcher in Kombination mit einer Stapelgröße von $b = 10.000$ die kürzesten Lösungswege ermitteln soll, ohne die Laufzeit zu stark negativ zu beeinflussen. Deshalb soll auch in den nachfolgenden Experimenten dieser Wert eingesetzt werden.

Lösungsdauer

Zunächst wird wieder die benötigte Laufzeit pro Suchdurchlauf abhängig von der Verdrehungstiefe der unbekannten an das System übergebenen Würfelzustände untersucht. Um die Methode besser bewerten zu können, wird sie dem zuvor untersuchten MCTS-Verfahren, welches eine Stapelverarbeitung beinhaltet, gegenübergestellt. Aus Grafik 10.9 geht hervor, dass den Erwartungen entgegen die MCTS-Methode durchschnittlich weniger Zeit benötigt als das Verfahren mit dem A*-Suchalgorithmus. Durch die komplexere Funktionsweise der Monte-Carlo-Baumsuche und dem oft sehr großen Suchbaum liegt die Vermutung nahe, dass diese Methode mehr Zeit in Anspruch nehmen würde als der vom Aufbau einfachere A*-Algorithmus. Großen Einfluss auf die Laufzeit besitzt hierbei höchstwahrscheinlich der Gewichtungsfaktor λ. In ihrem Artikel [22] schildern McAleer et al., dass je größer λ gewählt wird, desto größer auch die Laufzeit, aber desto kleiner

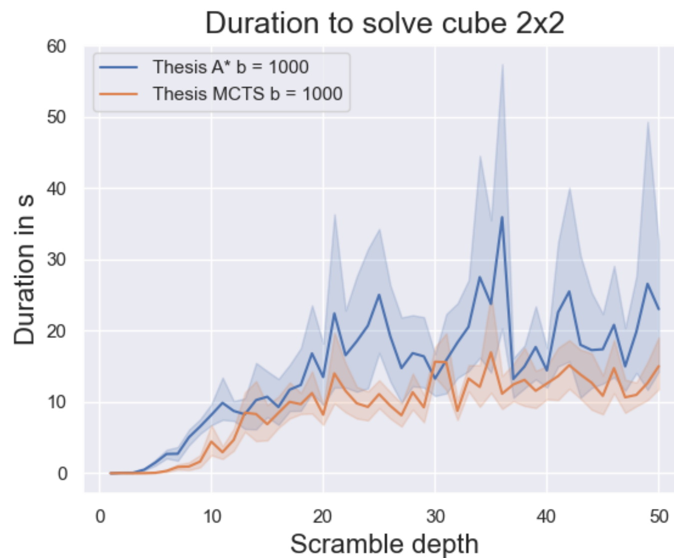


Abbildung 10.9: Benötigte Laufzeit des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$.

die Lösungslänge wird. Würde λ somit kleiner gewählt werden, könnten Verbesserungen in der Laufzeit zuungunsten der Lösungsweglänge erzielt werden.

Erfolgsquote

Der Anteil der gelösten Zauberwürfel unterscheidet sich bei den beiden entworfenen Systemen nicht. Beide erzielen mit 100% Erfolgsrate optimale Resultate (vgl. Abbildung 10.10).

Iterationen pro Suchdurchlauf

Da die Methode mit A*-Algorithmus ein wenig mehr Zeit benötigt als die mit MCTS, könnte vermutet werden, dass auch die Anzahl der benötigten Iterationen pro Suchdurchlauf größer ist. Doch in Abbildung 10.11 wird deutlich, dass dies nicht der Fall und genau umgekehrt ist. Dies wird das Ergebnis einer größeren erforderlichen Laufzeit pro Suchschritt sein und mit großer Wahrscheinlichkeit wieder mit dem gewählten Gewichtungsfaktor $\lambda = 0,6$ zusammenhängen. Die Gewichtung der Distanz wird vergrößert, was die Suche präziser werden lässt, aber auch mehr Ressourcen erfordert. Durch ein größeres

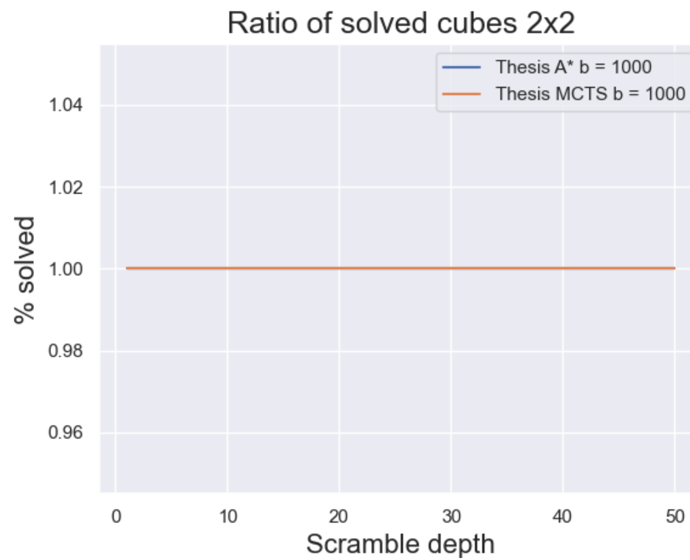


Abbildung 10.10: Erfolgsquote des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$.

λ wären sehr wahrscheinlich weniger Iterationen nötig, aber die benötigte Zeit pro Schritt würde vermutlich noch weiter zunehmen.

Länge des Lösungsweges

Die Länge des Lösungsweges des Systems mit MCTS befindet sich mit einem durchschnittlichen Wert von ≤ 12 bereits sehr nah an der Optimalität. Das in Grafik 10.12 dargestellte Konfidenzintervall macht deutlich, dass dennoch nicht jede gefundene Lösung auch wirklich optimal ist und einige Ausreißer noch immer mehr als 14 Rotationen benötigen bis der Zielzustand erreicht wird. Kurvenverlauf und Konfidenzintervall zeigen hierbei, dass das A*-Verfahren wesentlich häufiger in der Lage ist einen optimalen Lösungpfad zu ermitteln. Mit etwa 9 Rotationen im Durchschnitt pro Suchdurchlauf ist das Verfahren noch um einiges zuverlässiger und präziser als das Pendant mit MCTS und findet durchweg einen optimalen Lösungspfad.

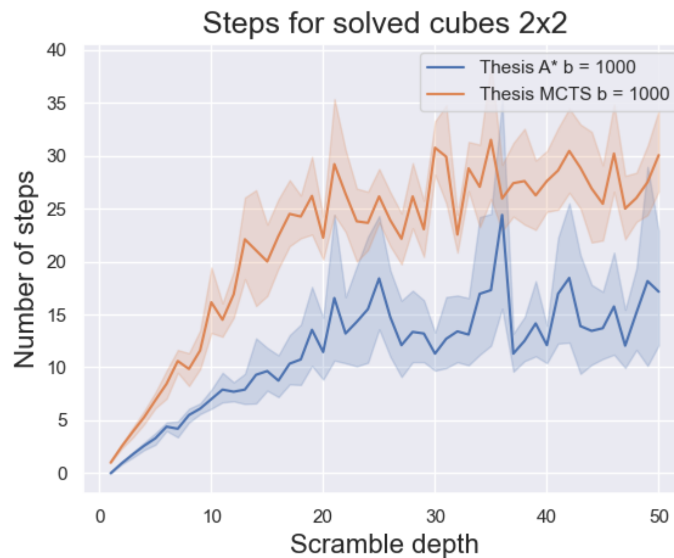


Abbildung 10.11: Anzahl der benötigten Iterationen des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$.

Gridsearch Hyperparameter

Um die Festlegung der Hyperparameter $\lambda = 0,6$ und $b = 1.000$ zu überprüfen und gegebenenfalls Trends für bessere Parameterkonfigurationen zu ermitteln, sollen mittels einer eigens entworfenen Gridsearch verschiedene Kombinationen systemisch getestet werden. Dafür wird eine Verdrehungstiefe von 20 genutzt, da die benötigte Zeit, die Anzahl der Schritte und die Lösungsweglänge des A*-Algorithmus ab dort ungefähr stabil bleiben (vgl. Abbildungen 10.9, 10.11 und 10.12) und vermutet werden kann, dass dies bereits den höchsten Schwierigkeitsgrad darstellt. Statt der vorherigen fünf Minuten Laufzeit, welche sich aus der in Kapitel 4 gestellten Anforderung S-NF1 ergibt, wird ein Suchdurchlauf hier nach zwei Minuten automatisch unterbrochen und als nicht erfolgreich deklariert. Dies liegt einerseits darin begründet, dass sich in den Versuchen im Abschnitt zuvor gezeigt hat, dass das Verfahren mit der A*-Suche bei einer Batchverarbeitung mit $b = 1.000$ im Durchschnitt weniger als eine Minute pro Suchdurchlauf benötigt. Andererseits würden Suchdurchläufe, welche eine ungünstige Parameterkombination von λ und Batchgröße besitzen, die maximal eingestellte Laufzeit beanspruchen und das für jeden übergebenen Würfelzustand.

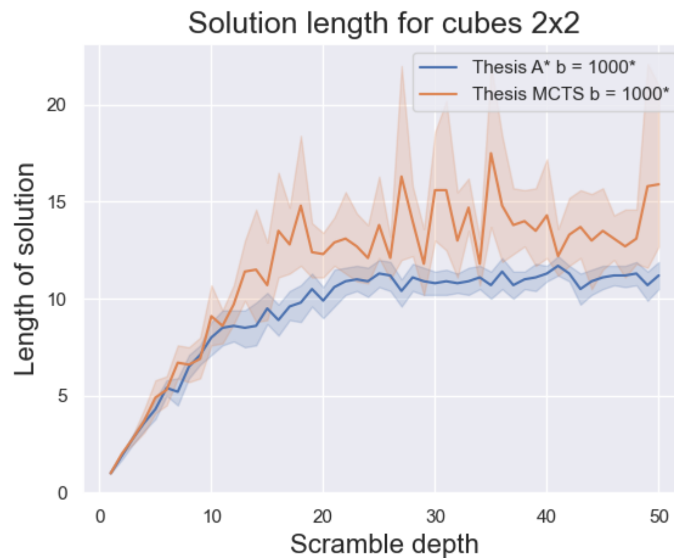
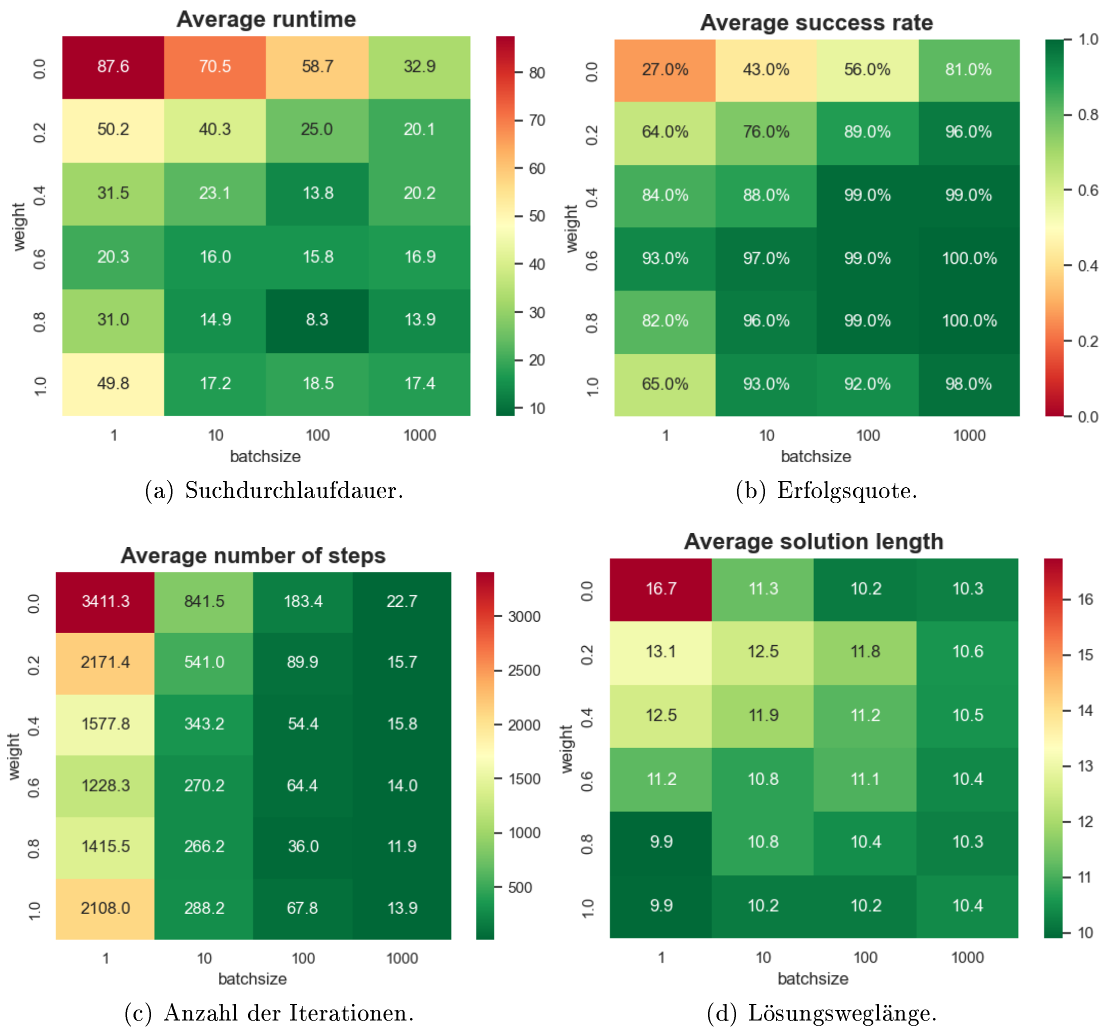


Abbildung 10.12: Lösungsweglänge des A*-Verfahrens im Vergleich zur MCTS-Methode mit einer Batchverarbeitung der Größe $b = 1.000$ und $\lambda = 0,6$.

Um qualitativere Aussagen über den Einfluss der Paramter λ und Stapelgröße b treffen zu können, werden deshalb diesmal 100 unbekannte Würfelzustände pro Konfiguration übergeben. Nach jedem Durchlauf werden wieder die Kriterien wie zuvor notiert und anschließend ausgewertet. Für die Suche wird jede Kombination vom Gewichtungsfaktor $\lambda \in \{0; 0,2; 0,4; 0,6; 0,8; 1\}$ und Stapelgröße $b \in \{1; 10; 100; 1.000\}$ mit dem A*-Suchalgorithmus getestet. Die Werte sind hierbei an denen angelehnt, die McAleer et al. für eine eigene Gridsearch verwendeten, um ebenfalls die optimale Parameterkonfiguration zu ermitteln. Bei einer Stapelgröße $b = 10.000$ traten Speicherplatzprobleme bei dem verwendeten System auf, weshalb $b = 1.000$ den maximalen Wert für diesen Versuch darstellt. Die hierbei erzeugten Resultate sind der Abbildung 10.13 zu entnehmen. In den Grafiken sind eindeutige Trends für geeignete Parameterkombination zu erkennen. Ist eine Erfolgsquote von 100% zwingend erforderlich, verbleiben in diesem Szenario allerdings nur zwei mögliche Konfigurationen, welche sich nur minimal in den erbrachten Resultaten unterscheiden. Die erste ist die bereits gewählte Einstellung der vorherigen Tests mit einer Batchgröße von $b = 1.000$ und Gewichtungsfaktor $\lambda = 0,6$. Die zweite ist die mit einer wenig größeren Gewichtung von $\lambda = 0,8$ und einer gleichen Stapelgröße. Die einzigen Unterschiede sind dabei wenig kürzere Lösungswege sowie eine verringerte Laufzeit bei der Parameterkonfiguration mit $\lambda = 0,8$. Auch eine Aufstellung mit $\lambda = 0,8$ und $b = 100$ könnte durchaus vielversprechend sein. Hierbei löst das System die über-



(a) Suchdurchlaufdauer.

(b) Erfolgsquote.

(c) Anzahl der Iterationen.

(d) Lösungsweglänge.

Abbildung 10.13: Ergebnisse der Gridsearch für die Findung der optimalen Parameterkonfiguration (λ, b).

gebenen Würfelzustände „nur“ mit einer Erfolgsquote von 99%, dafür aber schneller als alle anderen getesteten Konfigurationen und sogar doppelt so schnell wie die mit $\lambda = 0,6$ und $b = 1.000$.

10.2 Evaluierung der Software: Bedienoberfläche

Für die Auswertung der erstellten benutzerbasierten Anwendung wurde diese dem Erstprüfer Prof. Dr. Hensel übergeben, welcher diese auf die Intuitivität und Nutzbarkeit als unparteiischer Anwender prüfte.

10.3 Evaluierung der Hardware

Eine erneute Validierung der Funktionstüchtigkeit der Hardware ist an dieser Stelle redundant, da diese extra von Erstprüfer Prof. Dr. Hensel für diesen Zweck entworfen wurde und dementsprechend von Anfang an einsatzbereit gewesen ist. Die Hardware hat in Kombination mit der entworfenen benutzerbasierten Anwendung fehlerfrei funktioniert, und die Anforderungen an die Hardware werden deshalb als erfüllt betrachtet.

10.4 Auswertung der Ergebnisse

Einige Anforderungen aus Kapitel 4 erfordern keine zusätzliche Auswertung und konnten bereits während der Entwicklung verifiziert werden. Mit den in den Abschnitten zuvor erbrachten Resultaten lässt sich die Ausarbeitung nun in der Gesamtheit im Bezug auf alle gestellten Forderungen auswerten. Dafür werden die Tabellen 4.6, 4.7 und 4.8 aus der Anforderungsanalyse erneut aufgeführt und um eine dem Erfüllstatus entsprechende Symbolik ergänzt:

- ✓ : Anforderung wurde in vollem Umfang erfüllt.
- ● : Anforderung wurde teilweise oder mit Einschränkungen erfüllt.
- ✗ : Anforderung konnte nicht erfüllt werden.

10.4.1 Software

Zunächst wird der Software-Teil des entworfenen Systems bewertet, begonnen mit den funktionalen Anforderungen. Anzumerken ist, dass Anforderung S-F8 im Kapitel 5 angepasst wurde, da die *Gottes Zahl* für den Pocketcube 14 beträgt. In Tabelle 10.1 wurden die funktionalen Forderungen an das System der vorgestellten Symbolik nach markiert.

Tabelle 10.1: Auswertung der funktionalen Anforderungen, welche an die Software gestellt wurden.

ID	Priorität	Anforderung	Status
S-F1	Hoch	Berechnung des Lösungsweges mittels KI	●
S-F2	Hoch	Eingabe des Würfelzustandes in einer benutzerbasierten Anwendung	✓
S-F3	Mittel	Anzeige des aktuellen Zustandes in 2D	✓
S-F4	Niedrig	Anzeige des aktuellen Zustandes in 3D	✓
S-F5	Hoch	Erkennen, ob aktueller Zustand der Lösungszustand ist	✓
S-F6	Hoch	Erzeugen zufällig verdrehter Würfelzustände für das Training des neuronalen Netzes	✓
S-F7	Hoch	Automatisches Abbrechen nach einer eingestellten Anzahl von Berechnungsschritten	✓
S-F8	Niedrig	MCTS: Finden eines optimalen Lösungsweges (≤ 14, *Gottes Zahl*)	●
S-F8	Niedrig	A*: Finden eines optimalen Lösungsweges (≤ 14, *Gottes Zahl*)	✓

Dabei sind vor allem solche erwähnenswert, die nur mit Einschränkungen realisiert werden konnten, da bei erfolgreicher Erfüllung keine weitere Erläuterung zu dem Kriterium erforderlich ist und keine Punkte gar nicht umgesetzt werden konnten.

Dabei fiel bereits in Kapitel 5 die Schwierigkeit bei der Umsetzung von S-F1 auf, welche die Berechnung des Lösungsweges alleinig durch eine künstliche Intelligenz beinhaltet. Die Schwierigkeit dieser Aufgabe wurde bereits von Lapan in seinem Buch [18] angesprochen und liegt besonders an der Modellierung des Würfels in Programmcode und der Größe des Zustandsraumes bei dem Rubik's Cube. Aus diesem Grund wurde den Vorbildern von McAleer et al. und Lapan entsprechend ein neuronales Netzwerk mit geeigneten Suchalgorithmen kombiniert, das die Lösungswegberechnung vornimmt. Deshalb gilt S-F1 nur als teilweise erfüllt, da die Berechnungen nicht allein durch eine künstliche Intelligenz geschehen sind. Anforderung S-F8 wurde in Tabelle 10.1 in zwei Punkte unterteilt, da hier Unterscheidungen zwischen den beiden genutzten Verfahren gemacht werden müssen. Obwohl das entworfene MCTS-Verfahren wesentlich besser performte als das von Lapan, ist die gefundene Lösungsweglänge noch immer sehr lang und nur bei dem Verfahren mit Stapelverarbeitung im Durchschnitt wirklich optimal. Einige Ausreißer zeigen deutlich, dass abhängig von dem unbekannten Zustand die MCTS-Methode noch Probleme bei der Findung eines optimalen Weges aufweist. Durch Anwendung einer anschließen-

Tabelle 10.2: Auswertung der nicht funktionalen Anforderungen, welche an die Software gestellt wurden.

ID	Priorität	Anforderung	Status
S-NF1	Hoch	Maximale Rechenzeit von 5 Minuten	✓
S-NF2	Hoch	Software ist in der Lage mindestens 90 % der Würfel zu lösen	✓
S-NF3	Hoch	Redundanz in Würfelzustandsdarstellung vermeiden [18]	✓
S-NF4	Hoch	Effiziente Speicherung von Würfelzuständen [18]	✓
S-NF5	Hoch	Perfomante Implementation von Würfeltransformationen [18]	✓
S-NF6	Hoch	Geeignete Zustandsrepräsentation für ein neuronales Netz [18]	✓
S-NF7	Niedrig	Eingängige Bedienoberfläche für die Verwendung der Software (GUI)	✓

den Breitensuche könnte dieses Problem behoben werden, konkurriert allerdings mit der höher priorisierten nicht funktionalen Anforderung S-NF1 an die maximale Laufzeit, welche ausschlaggebend für eine positive Benutzererfahrung der Software ist. Natürlich sind auch zu lange Lösungspfade problematisch, vor allem, wenn eine angeschlossene Hardware im Anschluss entsprechend lange benötigt, um den Würfel zurück in den Zielzustand zu versetzen. Deshalb ist eine ausgewogene Balance zwingend erforderlich.

In den Experimenten mit dem MCTS-System, welches eine Batchverarbeitung beinhaltet, ergab sich, dass dies ein durchaus möglicher Lösungsansatz für diese Anforderung sein könnte. Da der A*-Algorithmus mit einer größeren Stabilität noch um einiges kürzere Wege zum Zielzustand ermittelte als die MCTS-Methode mit Stapelverarbeitung (im Mittel etwa 3 Rotationen weniger), gilt das Kriterium für A* als vollständig und für MCTS als nur teilweise erfüllt.

Tabelle 10.2 ist die Auswertung der nicht funktionalen Anforderungen zu entnehmen. Hier konnten alle Anforderungen ohne Einschränkungen erfüllt werden. Aus den Versuchsdurchführungen geht hervor, dass die Verfahren durchweg in den Rahmenbedingungen an Laufzeit (S-NF1) und Erfolgsquote (S-NF2) agieren. Die Anforderungen an die Würfelzustandsdarstellung und -transformation (S-NF3 bis S-NF6), welche von Lapan übernommen wurden, gelten als erfüllt, da sich diese Implementation stark auf seine Ausarbeitung stützt und dort bereits auf die Einhaltung dieser Kriterien geachtet wurde, weshalb sich ein erneutes Verfizieren erübrigt. Erstprüfer Prof. Dr. Hensel wurde die

entworfene Anwendung für das Testen auf Intuivität (S-NF7) in der Bedienbarkeit übergeben, welcher diese ohne Einwände bestätigen konnte. Aus diesem Grund gilt auch diese Rahmenbedingung als erfolgreich umgesetzt.

10.4.2 Hardware

Wie bereits in Abschnitt 10.3 erläutert, gilt die Hardware von Beginn als funktionsfähig, da diese von Prof. Dr. Hensel für diese Ausarbeitung entworfen und einsatzbereit an den Ersteller dieser Arbeit übergeben wurde. Deshalb werden die beiden Anforderungen in Tabelle 10.3 als erfüllt angesehen.

Tabelle 10.3: Auswertung der funktionalen Anforderungen, welche an die Hardware gestellt wurden.

ID	Priorität	Anforderung	Status
H-F1	Hoch	Erfassen des aktuellen Würfelzustandes	✓
H-F2	Hoch	Rotieren der Würfelseiten mittels Motoren	✓

11 Schlussfolgerungen

Um die hier vorgestellte Thesis zu einem Abschluss zu bringen, werden die erbrachten Arbeiten und zugehörigen Resultate an dieser Stelle noch einmal zusammengefasst. Anschließend werden die offenen Punkte, welche sich bereits im vorherigen Kapitel angedeutet haben, diskutiert und um weitere Aspekte, welche über die Anforderungen hinausgehen, ergänzt. Zu guter Letzt werden im Ausblick Anregungen formuliert, mit welchen die in dieser Thesis erarbeiteten Leistungen über- oder weiterbearbeitet werden könnten.

11.1 Zusammenfassung

Das ursprüngliche Ziel dieser Arbeit war das Entwerfen einer künstlichen Intelligenz, welche in der Lage sein sollte, eigenständig einen Rubik's Cube zu lösen. Dafür sollte mittels Reinforcement Learning ein neuronales Netz trainiert werden, welchem anschließend Würfelzustände übergeben werden können, die es dann schrittweise zurück zum Zielzustand überführt. Doch bereits während anfänglicher Recherchearbeiten zeigte sich, dass dies eine weit umfangreichere Aufgabe ist als zunächst angenommen. Dies liegt vor allem in der Komplexität und der Zustandsraumgröße des 3x3x3-Zauberwürfels, die mit $4,33 \cdot 10^{19}$ möglichen Zuständen ungefähr genauso groß ist wie die geschätze Anzahl der Sandkörner auf der Erde ($\approx 7,5 \cdot 10^{18}$). Um neue, unbekannte Würfelzustände lösen zu können, bedarf es eines geeigneten Würfelmodells, mit welchem sich ein Zauberwürfel auf dem Computer simulieren lässt, sowie einer ausgereiften Trainingsmethode. Das neuronale Netzwerk muss lernen, Strukturen in Aktionsfolgen zu erkennen und anschließend korrekt anzuwenden, um einen Würfel selbstständig zurück in den Zielzustand zu überführen. Deshalb wurde sich frühzeitig entschieden, den Fokus auf die notwendigen und umfangreichen Grundlagen des maschinellen Lernens, die für ein solches Vorhaben relevant sind, zu legen und bereits erfolgreiche Methoden zu studieren und wenn möglich, eigenständig umzusetzen. Diese Thesis soll somit insbesondere der Wissensvermittlung

dienen und anderen Arbeiten einen Einstieg in diese Thematik erleichtern, indem die wichtigsten Bereiche ausführlich vorgestellt und erläutert werden.

Dadurch, dass frei zugängliche Artikel über Ausarbeitungen von McAleer et al. [21, 22] und ein ausführliches Kapitel zu [21] von Max Lapan in seinem Buch [18] vorliegen, konnte an dieser Stelle auf die dort verwendeten Methoden zurückgegriffen werden. In diesen werden Ansätze beschrieben, wie ein neuronales Netzwerk geschickt mit Suchalgorithmen kombiniert werden kann, um beliebige Zauberwürfelzustände zu lösen. Aufgrund der benötigten Rechenressourcen, welche sich nur durch nur eine immens verlängerte Laufzeit des Trainings kompensieren lässt, wurde beschlossen, sich auf die Lösungsansätze zu konzentrieren und diese an dem kleinen 2x2x2-Zauberwürfel, auch Pocketcube genannt, zu testen. Der Aufbau der neuronalen Netze gleicht hierbei denen der angegebenen Vorbilder, wobei wieder teilweise Verbesserungen vorgenommen wurden. Auch bei dem Training von diesen wurden Änderungen zur Minimierung der Verlustfunktion getätigt, um die bereits sehr beeindruckenden Ergebnisse der veröffentlichten Artikel unter Umständen noch weiter zu verbessern.

In den Evaluationen bestätigte sich, dass vor allem bei dem Verfahren mit MCTS diese Änderungen im Vergleich zu Lapans Implementation aussagekräftige Resultate erzielen konnten. Für eine Bewertung der Methode mit A*-Algorithmus fehlt an dieser Stelle eine geeignete Referenz, da McAleer et al. ihr System nicht am Pocketcube getestet haben und somit keine Auswertung zu solchem vorliegt. Dennoch wurden genauso gute Ergebnisse wie mit dem MCTS-Algorithmus erzielt und im Durchschnitt sogar kürzere Lösungswege für beliebige Würfelzustände gefunden.

Um die entworfenen Systeme praktisch einsetzbar zu machen, entwickelte Prof. Dr. Hensel eine Apparatur, in welche sich ein 2x2x2-Würfel einlegen lässt und dessen Seiten sich über eine Python-API mittels zwei Servomotoren rotieren lassen. Welche Seite des Würfels rotiert wird, lässt sich dabei durch eine bestimmte Reihenfolge, in der die Servomotoren angesteuert werden und welche für jede Seite unterschiedlich ist, realisieren. Zu diesem Zweck wurde zusätzlich eine Bedienoberfläche entworfen, welche die Vereinigung der Lösungsverfahren mit der Hardware übernimmt und es Nutzer:innen ermöglicht den kleineren Zauberwürfel mit Hilfe dieses Systems lösen zu lassen. Dabei kann die Zustandsaufnahme des Pocketcubes manuell geschehen, indem die Farben des Würfels in dafür vorgesehene Felder übertragen werden, oder halb automatisch, indem eine Kamera angeschlossen wird und die vom System erkannten Farben unter Umständen manuell korrigiert werden. Ingesamt konnten die von McAleer et al. und Lapan vorgestellten

Verfahren erfolgreich nachimplementiert und sogar teilweise verbessert werden. Durch Anschließen der für diesen Zweck entwickelten Hardware lassen sich sogar beliebige (reale) 2x2x2-Zauberwürfel zurück in ihren Zielzustand versetzen. Dies ermöglicht es auch unerfahrenen Benutzer:innen die erarbeiteten Lösungsverfahren zu nutzen.

11.2 Offene Punkte

Ein Großteil der offengebliebenen Anforderungen wurde bereits im vorherigen Abschnitt 10.4, in welchem die Auswertung der Ergebnisse vorgenommen wurde, diskutiert. Hierbei konnte festgestellt werden, dass keine Anforderung gänzlich unbearbeitet geblieben ist und nur wenige mit Einschränkungen umgesetzt wurden. Um die Zuverlässigkeit des MCTS-Verfahrens, den optimalen Lösungsweg zu ermitteln, zu steigern, können weitere Experimente bezüglich der dort verwendeten Hyperparameter Verlust ν und Erkundungskonstante c vorgenommen werden (vgl. Gleichung 8.6 und Listing 8.6 auf Seite 146). Dafür könnte eine Gridsearch durchgeführt werden, welche ähnlich wie bei dem A*-Verfahren aufgebaut ist (vgl. Abschnitt 10.1.3 auf Seite 176). Die wichtigsten offenen Punkte sind jedoch einerseits das Ermitteln des Lösungsweges alleinig durch eine künstliche Intelligenz, welches zuvor das eigentliche Ziel gewesen ist und eine noch intensivere Bearbeitung der Thematik und vor allem größere Rechenkapazitäten erfordert hätte, und, dass statt des eigentlichen Lösens des Rubik's Cube der Pocketcube verwendet wurde, um die ausgearbeiteten Ansätze zu testen. Wird die hier geschaffene Grundlage mit mehr Rechenleistung für die vorgestellten Methoden kombiniert, sollten diese mit großer Zuversicht genauso gute Resultate mit einem 3x3x3-Würfel erzielen.

11.3 Ausblick

Der Bereich des maschinellen Lernens ist an sich bereits ein komplexes und umfangreiches Themengebiet, dessen Komplexität je nach Aufgabenstellung, die es zu bewältigen gilt, noch steigen kann. Das Lösen eines Zauberwürfels stellte sich als schwierige Aufgabe heraus, welche nicht ohne nötiges Fachwissen gemeistert werden kann. Denn es ist nicht nur ausreichendes Wissen im Bereich der künstlichen Intelligenz notwendig, sondern auch in dem Bereich, wo diese eingesetzt werden soll.

Der hier untersuchte Zauberwürfel ist ein kompliziertes mathematisches Objekt, und daher ist eine korrekte Darstellung davon für das Training eines neuronalen Netzes unerlässlich. Bereits die Zustandsrepräsentation und die dazugehörigen Würfeltransformationen bieten Optimierungspotenzial, um das Training eines Netzwerkes weiter positiv zu beeinflussen. Die Umsetzung der Würfelzustandsrepräsentation könnte somit beispielsweise über eine bitweise Darstellung geschehen, welche weniger Rechenressourcen erfordert und die Geschwindigkeit der durchgeführten Rotationen verbessern könnte, indem die genutzten Bits, der durchgeführten Aktion entsprechend, einfach neu sortiert werden. Systeme wie jene, die hier entworfen wurden, bieten einige Anhaltspunkte, um diese weiter zu optimieren, jedoch ohne Garantie, dass dies auch wirklich bessere Resultate erzielt.

Das Ändern eines Hyperparameters kann häufig starke ungeahnte Auswirkungen auf den Trainingserfolg eines neuronalen Netzwerkes nach sich ziehen. Werden diese händisch einzeln ausgesucht, kann dies schnell zu einer zeit- und nervenraubenden Angelegenheit werden. Um ein Verständnis für den Einfluss von den Hyperparamtern zu entwickeln, kann eine manuelle Auslegung dennoch zweckmäßig sein. Möchte man auf den in dieser Thesis vorgestellten Ansätzen aufbauen, ist vor allem eine gezielte Findung optimaler Hyperparameter mittels Grid- oder Randomsearch sinnvoll. Auch ein Testen verschiedener neuronaler Netzwerke könnte sehr wahrscheinlich noch eine Performancesteigerung der Verfahren bewirken. Lapan nennt in seinem Buch [18] in diesem Zusammenhang das in Abschnitt 2.2.7 vorgestellte Convolutional Neural Network. Er führt dort aus, dass der Rubik's Cube ein komplexes Objekt ist und ein Feedforward Neural Network möglicherweise nicht so geeignet ist wie ein Netzwerk, welches Faltungen inkorporiert [18]. Die Wahl eines Residual Neural Networks im zweiten Teil dieser Arbeit hat gezeigt, dass es durchaus möglich ist andere Netze mit ähnlichen oder besseren Resultaten im selben Kontext zu verwenden. Es gibt eine Vielzahl von Netzwerkarchitekturen, welche eine nähere Untersuchung wert sein könnten, um jene in Kombination mit geeigneten Suchverfahren zu testen.

Aber auch die verwendeten Suchverfahren können verbessert oder gänzlich gegen neue Methodiken getauscht werden. Der Suchbaum des MCTS-Verfahrens ist noch immer immens groß und speicherintensiv. Funktionen, um diesen zu beschneiden und nicht vielversprechende Zweige frühzeitig aus diesem zu entfernen, könnten verringerte Laufzeiten oder auch kürzere Lösungsweglängen erzielen. Von dem A*-Algorithmus gibt es einige Abwandlungen, die verwendet werden könnten, um auch diesen zu beschleunigen und die Länge der gefunden Lösungen noch näher an den optimalen Wert zu bringen. Auf Amid Patels Internetseite [26] befinden sich hierzu mögliche Anregungen, die in Betracht ge-

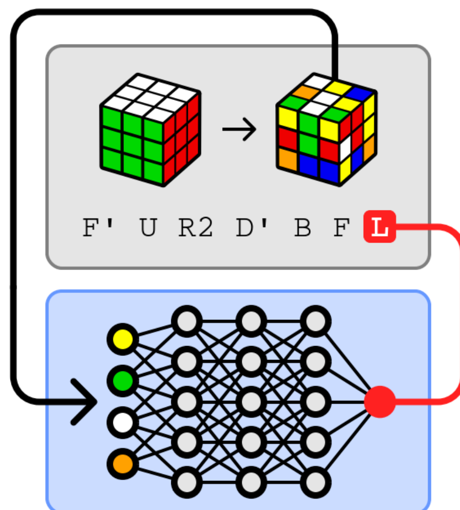


Abbildung 11.1: Darstellung der Funktionsweise des von Kyo Takano entwickelten Verfahrens [33].

zogen werden könnten. Darunter beispielweise das bidirektionale A*-Verfahren, welches sowohl im Start- als auch im Zielzustand die Suche beginnt und sich optimalerweise in der Mitte der Lösung trifft. Aber auch ganz andere Ansätze, die sich einfacher vom methodischen Aufbau gestalten, könnten genauso zielführend oder sogar erfolgreicher sein als eine Kombination von einem neuronalen Netz mit einem Suchalgorithmus.

In einer späteren Rechercheарbeit dieser Ausarbeitung wurde ein weiterer relevanter Artikel entdeckt, der genau dies beinhaltet. Im Artikel [33] von Kyo Takano beschreibt dieser, wie er ein neuronales Netzwerk (Residual Neural Network) mit Hilfe von zufällig generierten Würfelzuständen trainiert, die durch zufällige Aktionen ausgehend vom Zielzustand erzeugt werden. Dabei wird der aktuelle Zustand dem Netzwerk als Eingabe übergeben und anschließend gibt dieses die Aktion aus, welche diesen Zustand laut seiner Berechnungen am ehesten hervorgerufen hat (vgl. Abbildung 11.1). Diese Ausgabe wird mit der tatsächlichen Aktion, die zu dem übergebenen Würfelzustand geführt hat, abgeglichen und die Modellparameter dementsprechend aktualisiert. Er schildert, dass seine Methode genauso gut und sogar besser sein soll, als die von *DeepCubeA* erbrachten Leistungen. Aufgrund der mangelnden Zeit kann der dort vorgestellte Ansatz in dieser Ausarbeitung nicht näher untersucht werden, sollte in einer Neuaufnahme dieser Thematik aber definitiv erforscht und nachimplementiert werden.

Soll ein neuronales Netz dahingehend trainiert werden einen 3x3x3-Zauberwürfel zu lösen, wird an dieser Stelle noch einmal darauf hingewiesen, dass es zielführend ist, sicherzustellen, dass ausreichende Rechenkapazitäten zur Verfügung stehen, um die Lernzeit des Netzes in einem angemessenen Rahmen halten zu können.

Literaturverzeichnis

[1] Berg, Mark de ; Cheong, Otfried ; Kreveld, Marc van ; Overmars, Mark: *Computational Geometry*. Springer, 2008

[2] Chaslot, Guillaume ; Bakkes, Sander ; Szita, Istvan ; Spronck, Pieter: Monte-Carlo Tree Search: A New Framework for Game AI. In: *Universiteit Maastricht / MICC* (2008)

[3] Dauphin, Yann N.: Identifying and attacking the saddle point problem in high-dimensional non-convex optimization. (2014). – URL https://arxiv.org/pdf/1406.2572.pdf. – Zugriffsdatum: 07.07.23

[4] Diebel, James: Representing Attitude: Euler Angles, Unit Quaternions, and Rotation Vectors. In: *Stanford University* (2006). – URL https://www.astro.rug.nl/software/kapteyn-beta/_downloads/attitude.pdf. – Zugriffsdatum: 07.07.23

[5] GitHub: *PYPL PopularitY of Programming Language*. – URL https://pypl.github.io/PYPL.html?country=DE. – Zugriffsdatum: 07.07.23

[6] Glorot, Xavier ; Bengio, Yoshua: Understanding the difficulty of training deep feedforward neural networks. (2010)

[7] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[8] Gridin, Ivan: *Practical Deep Reinforcement Learning with Python: Concise Implementation of Algorithms, Simplified Maths, and Effective Use of TensorFlow and PyTorch*. BPB Publications, 2022. – URL https://www.amazon.de/-/en/Ivan-Gridin/dp/9355512058/ref=tmm_pap_swatch_0?_encoding=UTF8&qid=1675413549&sr=8-2-fkmr2. – ISBN 978-9355512055

[9] Géron, Aurélien: *Praxiseinstieg Machine Learning mit Scikit-Learn und TensorFlow: Konzepte, Tools und Techniken für intelligente Systeme*. O'Reilly-Büchern, 2018

[10] Göllmann, Laurenz: *Lineare Algebra: im algebraischen Kontext*. Springer Spektrum, 2023. – 3., überarbeitete und erweiterte Auflage

[11] He, Horace: The State of Machine Learning Frameworks in 2019. (2019). – URL https://thegradient.pub/state-of-ml-frameworks-2019-pytorch-dominates-research-tensorflow-dominates-industry/. – Zugriffsdatum: 07.07.23

[12] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Deep Residual Learning for Image Recognition. In: *Microsoft Research* (2015)

[13] He, Kaiming ; Zhang, Xiangyu ; Ren, Shaoqing ; Sun, Jian: Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification. (2015)

[14] Hensel, Marc: *Pocket Cube Solver: A Motorized Device for Mini Cubes*. Juli 2023. – URL https://github.com/MarcOnTheMoon/cubes

[15] Howard, Jeremy ; Gugger, Sylvain: *Deep Learning for Coders with Fastai and PyTorch: AI Applications Without a PhD*. O'Reilly Media, 2020. – URL https://www.amazon.com/Deep-Learning-Coders-fastai-PyTorch/dp/1492045527. – ISBN 978-1492045526

[16] Ioffe, Sergey ; Szegedy, Christian: Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift. (2015). – URL https://arxiv.org/pdf/1502.03167.pdf. – Zugriffsdatum: 07.07.23

[17] Korf, Richard: Finding Optimal Solutions to Rubik's Cube Using Pattern Databases. (1997). – URL https://www.cs.princeton.edu/courses/archive/fall06/cos402/papers/korfrubik.pdf. – Zugriffsdatum: 07.07.23

[18] Lapan, Max: *Deep Reinforcement Learning Hands-On: Apply modern RL methods to practical problems of chatbots, robotics, discrete optimization, web automation, and more*. Packt Publishing, 2020. – URL https://www.amazon.de/Deep-Reinforcement-Learning-Hands-optimization/dp/1838826998. – ISBN 978-1838826994

[19] Levine, John: Monte Carlo Tree Search. In: *Class "CS310: Foundations of Artificial Intelligence"* University of Strathclyde (2017). – URL https://www.youtube.com/watch?v=UXW2yZndl7U. – Zugriffsdatum: 07.07.23

[20] Mao, Hongzi ; Alizadeh, Mohammad ; Menache, Ishai ; Kandula, Srikanth: Ressource Management with Deep Reinforcement Learning. (2016). – URL https://people.csail.mit.edu/hongzi/content/publications/DeepRM-HotNets16.pdf. – Zugriffsdatum: 07.07.23

[21] McAleer, Stephen ; Agostinelli, Forest ; Shmakov, Alexander ; Baldi, Pierre: Solving the Rubik's Cube Without Human Knowledge. In: *Conference paper at ICLR 2019* (2018)

[22] McAleer, Stephen ; Agostinelli, Forest ; Shmakov, Alexander ; Baldi, Pierre: Solving the Rubik's cube with deep reinforcement learning and search. In: *Nature machine intelligence* (2019)

[23] Miller, Tim: *COMP90054: Reinforcement Learning*. The University of Melbourne, 2022. – URL https://gibberblot.github.io/rl-notes/intro.html. – Zugriffsdatum: 07.07.23

[24] Million, Elizabeth: The Hadamard Product. (2007). – URL http://buzzard.ups.edu/courses/2007spring/projects/million-paper.pdf. – Zugriffsdatum: 07.07.23

[25] Nielsen, Micheal A.: *Neural Networks and Deep Learning*. Determination Press, 2015

[26] Patel, Amit J.: Introduction to the A* Algorithm. (2014). – URL https://www.redblobgames.com/pathfinding/a-star/introduction.html. – Zugriffsdatum: 07.07.23

[27] PyTorch: *How to use TensorBoard with PyTorch*. – URL https://pytorch.org/tutorials/recipes/recipes/tensorboard_with_pytorch.html. – Zugriffsdatum: 07.07.23

[28] Raschka, Sebastian ; Patterson, Joshua ; Nolet, Corey: Machine Learning in Python: Main Developments and Technology Trends in Data Science, Machine Learning, and Artificial Intelligence. (2020)

[29] Ruder, Sebastian: An overview of gradient descent optimization algorithms. In: *Insight Centre for Data Analytics, NUI Galway* (2017)

[30] Singmaster, David ; Alexander H. Frey, Jr.: *Handbook of Cubik Math*. Enslow Publishers, 1982

[31] Smith, Leslie N.: Cyclical Learning Rates for Training Neural Networks. (2017). – URL https://arxiv.org/pdf/1506.01186.pdf. – Zugriffsdatum: 07.07.23

[32] Sutton, Richard S. ; Barto, Andrew G.: *Reinforcement Learning, second edition: An Introduction (Adaptive Computation and Machine Learning series)*. Bradford Books, 2018. – URL https://www.amazon.de/-/en/Richard-S-Sutton/dp/0262039249/ref=sr_1_4?crid=1DV9OUPB4SWGX&keywords=richard+sutton+verst%C3%A4rkung+lernen&qid=1675413666&sprefix=richard+sutton+reinforcement+learnin%2Caps%2C75&sr=8-4. – ISBN 978-0262039246

[33] Takano, Kyo: Self-Supervision is All You Need for Solving Rubik's Cube. (2022). – URL https://arxiv.org/pdf/2106.03157v5.pdf. – Zugriffsdatum: 07.07.23

A Anhang

Auf dem zugehörigen Datenträger befinden sich folgende Anhänge:

- Die entworfene Bedienoberfläche mit und ohne einer 3D-Darstellungsmöglichkeit inklusive der Dateien mit den neuronalen Netzwerkparametern.
- Das Custom Environment des Pocketcubes für OpenAI Gym.
- Der Code für eine 3D-Darstellung des Pocketcubes mittels Pygame.
- Skripte für das Training der vorgestellten neuronalen Netze nach dem autodidaktischen Iterationsverfahren.
- Die Skripte der Versuchsdurchführungen inklusive der Ergebnisse.
- Relevante Literatur zur Thesis.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original