FH·W-S TECHNISCHE HOCHSCHULE NÜRNBERG
GEORG SIMON OHM

Reinforcement Learning am Beispiel von Schach

Fakultät Angewandte Mathematik, Physik und Allgemeinwissenschaften
der technischen Hochschule Nürnberg
Masterstudiengang: Angewandte Mathematik und Physik

Projektarbeit

vorgelegt von

Tim Selig
geboren am 15.06.1997 in Lohr a. Main
Matrikelnummer: 3582116
und
Stefan Schramm
geboren am 02.04.1998 in Haßfurt
Matrikelnummer: 3579620

im Juli 2021

Betreuer: Prof. Dr. Elke Wilczok

Inhaltsverzeichnis

Abbildungsverzeichnis III

Abkürzungsverzeichnis IV

1 Einführung 1

2 Grundlagen 2
2.1 Neuronales Netz (NN) . . . 2
2.2 Faltungsnetze/Convolutional Neural Network (CNN) . . . 3
2.3 Reinforcement Learning (RL) . . . 4

3 Algorithmen 6
3.1 Deep Q-Learning . . . 6
3.1.1 Q-Learning . . . 6
3.1.2 Deep Q-Network (DQN) . . . 8
3.1.3 Experience Replay . . . 8
3.2 Monte Carlo Tree Search (MCTS) . . . 9
3.2.1 Selection-Schritt . . . 10
3.2.2 Expansion-Schritt . . . 11
3.2.3 Simulation-Schritt . . . 12
3.2.4 Backpropagation-Schritt . . . 13
3.2.5 UCB-Algorithmus . . . 14

4 Realisierung 15
4.1 Konvertierung von Schachnotationen . . . 15
4.2 Deep Q-Network . . . 17
4.2.1 Neuronales Netz . . . 17
4.2.2 Algorithmus . . . 18
4.2.3 Training . . . 19
4.3 Monte-Carlo-Tree-Search . . . 21
4.3.1 Neuronales Netz . . . 21
4.3.2 UCB-Algorithmus . . . 22
4.3.3 Loss-Funktion . . . 22
4.3.4 Training . . . 23
4.3.5 Hyperparameter . . . 24

5 Dokumentation 26

6 Ergebnisse **30**

7 Fazit **32**
7.1 DQN 32
7.2 MCTS 32

Literaturverzeichnis **33**

Abbildungsverzeichnis

2.1 Struktur eines NNs[1] 2
2.2 2D-Faltung der Input-Schicht mit einem Convolutional Filter[2] 3
2.3 3D-Faltung der Input-Schicht mit einem Convolutional Filter[3] 4
2.4 Struktur des Reinforcement Learnings[4] 5

3.1 Q-Matrix[5] 7
3.2 Deep Q-Network[6] 8
3.3 Beispiel eines MCTS-Baumes[7] 9
3.4 Selection-Schritt[8] 11
3.5 Expansion-Schritt[8] 11
3.6 Simulation-Schritt[8] 12
3.7 Backpropagation-Schritt[8] 13

4.1 Initialzustand eines Schachspiels[9] 15
4.2 Trainingsalgorithmus für das DQN[10] 19
4.3 Verwendete Hyperparameter im MCTS-Algorithmus 25

6.1 Graphen der Loss-Funktionen über 200 Trainingsiterationen: Gesamt-Loss, Policy-Loss, Value-Loss, Regularisierungs-Loss 31

Abkürzungsverzeichnis

DQN	**D**eep **Q**-**N**etwork
MCTS	**M**onte **C**arlo **T**ree **S**earch
FEN	**F**orsyth-**E**dwards-**N**otation
RL	**R**einforcement **L**earning
UCB	**U**pper **C**onfidence **B**ound
NN	**N**eural **N**etwork
CNN	**C**onvolutional **N**eural **N**etwork
ReLU	**Re**ctified **L**inear **U**nit
MSE	**M**ean **S**quared **E**rror
Adam	**Ad**aptive **m**oment **e**stimation
KI	**K**ünstliche **I**ntelligenz
UCI	**U**niversal **C**hess **I**nterface

1 Einführung

Heutzutage würde wohl niemand mehr bestreiten, wie wichtig künstliche Intelligenz (KI) für unsere Zukunft ist. Durch die zunehmende Komplexität anfallender Aufgaben sowie den sich kontinuierlich weiterentwickelnden Lösungsansätzen sind einige Methoden entstanden, die im Bereich des maschinellen Lernens angewandt werden. Hierzu gehört beispielsweise das Reinforcement Learning (RL), welches mit dem Aufkommen neuronaler Netze (NN) ständig neue Einsatzmöglichkeiten liefert und eines der aussichtsreichsten Ansätze für die Zukunft darstellt. RL wurde bereits 1992 von Gerald Tesauro und seiner Forschungsgruppe bei IBM verwendet, um eine KI zu entwickeln, die in der Lage ist Backgammon zu spielen.

Seitdem haben sich die Algorithmen und vor allem die Hardware stark verbessert, sodass sogar Privatpersonen die Möglichkeit haben eigene KIs zu entwickeln. Damit der Komplexitätsgrad dennoch zeitgemäß bleibt, haben wir uns mit der Frage beschäftigt, ob wir in der Lage sind eine eigene KI für das Spiel Schach umzusetzen. Schach genießt in den letzten Jahren immer größere Beliebtheit und existiert bereits seit über 1500 Jahren. Zudem ist der Großteil der Menschen mit dem Spiel vertraut oder hat es bereits selbst gespielt. Des Weiteren ist die Menge der möglichen Schachpartien so enorm, dass die genaue Zahl bis heute nicht bekannt ist, weswegen wir uns für das Spiel Schach entschieden haben. Für die Umsetzung haben wir zwei verschiedene RL-Strategien mit Faltungsnetzen kombiniert. Dabei wird das Deep Q-Lerning sowie der Monte-Carlo-Tree-Search-Algorithmus ausführlich behandelt.

In dieser Arbeit wird aus Gründen der besseren Lesbarkeit das generische Maskulinum verwendet. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint.

2 Grundlagen

Das folgende Kapitel erklärt einige fundamentale Grundlagen zum Verständnis der vorliegenden Arbeit.

2.1 Neuronales Netz (NN)

Abbildung 2.1 zeigt den schematischen Aufbau eines NNs, bestehend aus Neuronen (Knoten) und Synapsen (Kanten). Es existiert eine Eingabeschicht, beliebig viele versteckte Schichten und eine Ausgabeschicht, die aus Neuronen bestehen. Die Neuronen repräsentieren dabei Zahlen, die das NN passieren. Auf den Synapsen sind veränderliche Gewichte gespeichert, um die Zahlen der Neuronen für die nächste Schicht zu berechnen. Das erfolgt durch eine Linearkombination der gewichteten Neuronen mit anschließender Anwendung einer Aktivierungsfunktion (z.B. rectified linear unit (ReLU), Leaky ReLU, Tangens hyperbolicus).

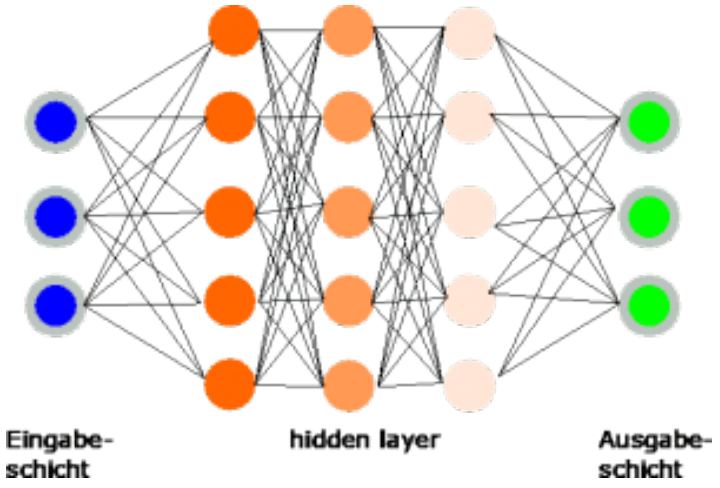

Abbildung 2.1: Struktur eines NNs[1]

Für das Training des NNs werden Soll- (Erwarteter Wert bei gegebenen Input) und Ist-Werte (Output des NNs) benötigt, die mithilfe einer sogenannten Loss-Funktion (z.B. mean squared error (MSE), Kreuzentropie) verglichen werden. Ziel des Trainings ist es die Loss-Funktion durch einen geeigneten Optimierer (z.B. adaptive moment estimation (Adam), stochastic gradient descent (SGD)) zu minimieren. Mithilfe eines Backpropagation-Algorithmus werden daraufhin die Gewichte im NN angepasst.
Damit das NN schneller und stabiler ist, kann zusätzlich Batch Normalization angewendet werden, was die Inputs der einzelnen Schichten neu zentriert und skaliert[11].

2.2 Faltungsnetze/Convolutional Neural Network (CNN)

Ein CNN ist in der Lage, die räumlichen und zeitlichen Abhängigkeiten für Input-Daten mit „gitterartigen“ Strukturen erfolgreich zu erfassen. Darunter zählen beispielsweise Bilder in zwei bzw. drei Dimension oder Brettspiele wie Schach. Hierfür werden die Neuronen durch Tensoren 2. Ordnung ersetzt, wodurch die Schichten des CNNs Tensoren 3. Ordnung entsprechen. Analog sind die Gewichte einer Schicht durch sogenannte „Filter-Tensoren“ zu ersetzen, welche ebenfalls Tensoren 3. Ordnung gleichkommen. Die Berechnung der nächsten Schicht erfolgt durch Faltung des Input-Tensors mit den Filter-Tensoren. Dieses Vorgehen ist für den Fall der zweidimensionalen Faltung in Abbildung 2.2 visualisiert.

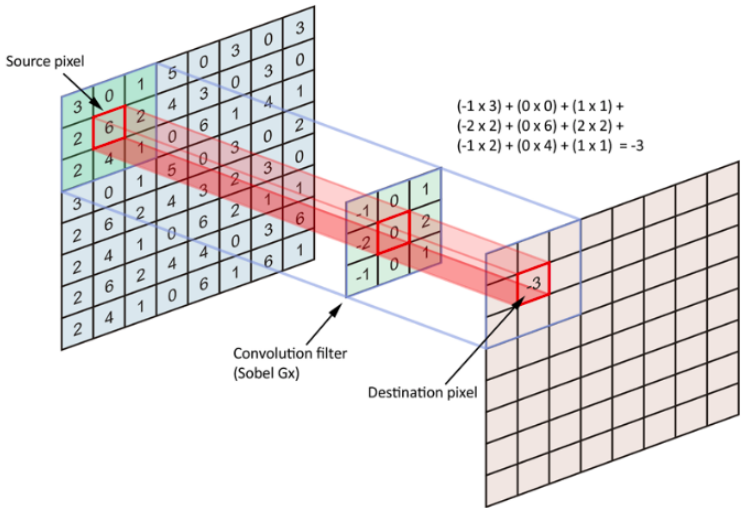

Abbildung 2.2: 2D-Faltung der Input-Schicht mit einem Convolutional Filter[2]

Die dreidimensionalen Faltung ist in Abbildung 2.3 dargestellt. Es ist darauf zu achten, dass die Tiefe (dritte Dimension, auch „Channel“ genannt) des Input-Tensors sowie der Filter-Tensoren identisch ist. Dadurch führt eine Faltung auf einen Tensor 2. Ordnung. Die Faltung verschiedener Filter-Tensoren liefert dann einen Tensor 3. Ordnung.

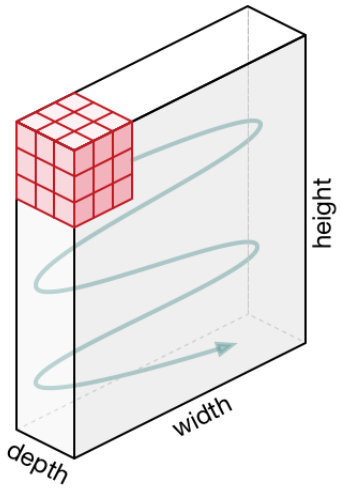

Abbildung 2.3: 3D-Faltung der Input-Schicht mit einem Convolutional Filter[3]

Da durch Faltung die Größe des Outputs verändert wird (vgl. Abbildung 2.2: (8x8)-Input liefert (6x6)-Output), werden sogenannte „Padding“-Techniken eingesetzt. Hierfür wird die Höhe und Breite des Input-Tensors vergrößert. Das bekannteste Beispiele ist das Zero-Padding, wobei der Rand des Inputs mit Nullen aufgefüllt wird. Für das Training werden analog zum gewöhnlichen NN mithilfe eines Backpropagation-Algorithmus die Einträge der Filter-Tensoren angepasst[3][11].

2.3 Reinforcement Learning (RL)

RL ist eine Methode des maschinellen Lernens, wobei ein Agent selbstständig eine Strategie erlernt, um erhaltene Belohnungen zu maximieren. Abbildung 2.4 visualisiert die Struktur des RLs, bestehend aus einer Umgebung (Environment) und dem Agenten. Der Agent ist lernfähig und agiert in einer Umgebung, die selbstständig oder durch Aktionen (Action) des Agenten verändert wird.

Im Beispiel von Schach entspricht die Umgebung dem Schachbrett, inklusive der Regeln des Spiels. Ein Zustand (Observation) ist eine konkrete Position und eine Aktion ist ein konkreter Zug. Der Agent handelt entsprechend seiner Strategie (Policy), um in einem beliebigen Zustand eine Aktion auszuwählen. Dadurch verändert sich der Zustand der Umgebung und der Agent erhält zusätzlich eine Bestrafung bzw. Belohnung (Reward) für die ausgeführte Aktion. Mithilfe eines RL-Algorithmus kann daraufhin die Belohnung verwendet werden, um die Strategie des Agenten anzupassen. Der Agent lernt somit eigenständig, in welcher Situation, welche Aktion die beste ist[12].

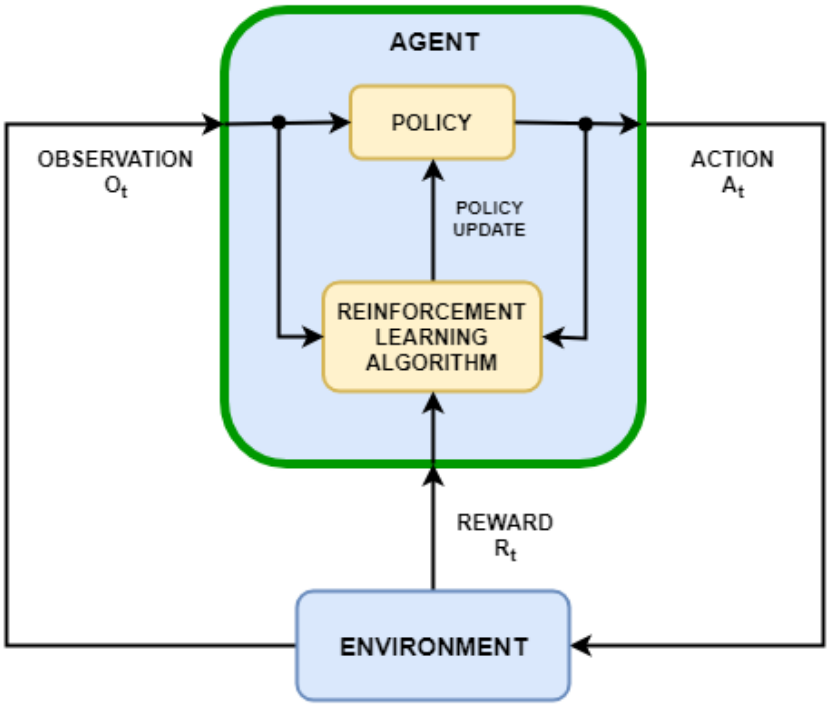

Abbildung 2.4: Struktur des Reinforcement Learnings[4]

Typische Anwendungsgebiete des RLs sind[13]:

- Spiele, wie Schach, Go oder 4-Gewinnt
- Robotik, z.B. Mäh- oder Saugroboter
- Autonomes Fahren

3 Algorithmen

Die folgenden Unterkapitel erklären die beiden grundlegenden Algorithmen, die für die vorliegende Arbeit verwendet werden.

3.1 Deep Q-Learning

Um Deep Q-Learning im vollen Umfang zu verstehen, wird Vorwissen von Q-Learning benötigt, welches zunächst beschrieben wird.

3.1.1 Q-Learning

Den Namen erhält Q-Learning von der sog. Q-Funktion $Q(s, a)$, die den erwarteten Nutzen Q einer Aktion a im Zustand s beschreibt. Die Werte werden in der sog. Q-Matrix Q gespeichert ($Q \epsilon \mathbb{R}^{|S|x|A|}$ mit $S \widehat{=} Zustandsraum$ und $A \widehat{=} Aktionsraum$). Während des Trainings versucht der Agent, die Werte der Q-Matrix mittels Exploration zu approximieren, um diese später als Entscheidungsregel zu nutzen. Hierbei gibt es für jedes Zustands-Aktions-Paar einen entsprechenden Belohnungswert, welcher zum Update der Q-Matrix verwendet wird.

Die Approximation der Q-Werte kann im einfachsten Fall wie folgt ablaufen:
Der Agent startet in einem zufällig initialisierten Zustand s_t. Anschließend wählt der Agent zufällig eine Aktion a_t aus A, woraus sich der Folgezustand s_{t+1} ergibt, und er erhält eine entsprechende Belohnung r_t. Die Update-Regel der Q-Matrix ist dann wie folgt definiert:

$$Q_{new}(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q(s_{t+1}, a))$$

Es handelt sich um eine iterative Formel, da der bereits gelernte Q-Wert $Q(s_t, a_t)$ zur Berech-

nung des neuen Q-Wertes $Q_{new}(s_t, a_t)$ verwendet wird. Der zweite Summand der Gleichung setzt sich aus der Belohnung und einer gewichteten Schätzung des optimalen Wertes im kommenden Zustand zusammen, was als Temporal Diffence Target bezeichnet wird.
Der Parameter α beschreibt die Lernrate des Algorithmus und steuert durch eine Konvexkombination, zu welchem Anteil der neue Wert vom alten bzw. vom Temporal Diffence Target beeinflusst wird.
Der Parameter γ ist der sog. Diskontierungsfaktor (discount factor) und steuert, wie stark kurzfristige oder zukünftige Belohnungen in der Entscheidungsfindung des Agenten berücksichtigt werden[14].

In Abbildung 3.1 ist ein schematisches Beispiel für die erste Iteration des Trainings und die daraus resultierenden Anpassungen der Q-Matrix. Hierbei gibt es 6 verschiedene Aktionen und 500 Zustände. Um nun die trainierte Q-Matrix für die Auswahl einer Aktion in einen bestimmten Zustand zu nutzen, muss die Aktion mit dem höchsten Q-Wert in dieser Zeile verwendet werden. In diesem Beispiel wird für den Zustand 328 die Aktion "North (1)" ausgewählt.

Initialized

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	327	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	0	0	0	0	0	0

Training

Q-Table		Actions					
		South (0)	North (1)	East (2)	West (3)	Pickup (4)	Dropoff (5)
States	0	0	0	0	0	0	0
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	328	-2.30108105	-1.97092096	-2.30357004	-2.20591839	-10.3607344	-8.5583017
	. . .	. . .	. . .	. . .	. . .	. . .	. . .
	499	9.96984239	4.02706992	12.96022777	29	3.32877873	3.38230603

Abbildung 3.1: Q-Matrix[5]

3.1.2 Deep Q-Network (DQN)

Deep Q-Learning verbindet nun das Q-Learning mit einem tiefen neuronalen Netz, dem DQN. Dieses dient als Approximator für Funktionen, die besonders beim RL nützlich sind, wenn der Zustands- oder der Aktionsraum zu groß ist, um vollständig bekannt zu sein.
Anstatt wie beim gewöhnlichen Q-Learning eine Lookup-Tabelle zu verwenden, um alle möglichen Zustände und ihre Werte zu speichern, können wir ein neuronales Netzwerk trainieren. Diese Vorgehensweise ist in Abbildung 3.2 verdeutlicht. Das Training erfolgt auf Stichproben aus dem Zustands- oder Aktionsraum, um zu lernen, wie wertvoll diese relativ zu unserem Ziel beim Reinforcement Learning sind[15].

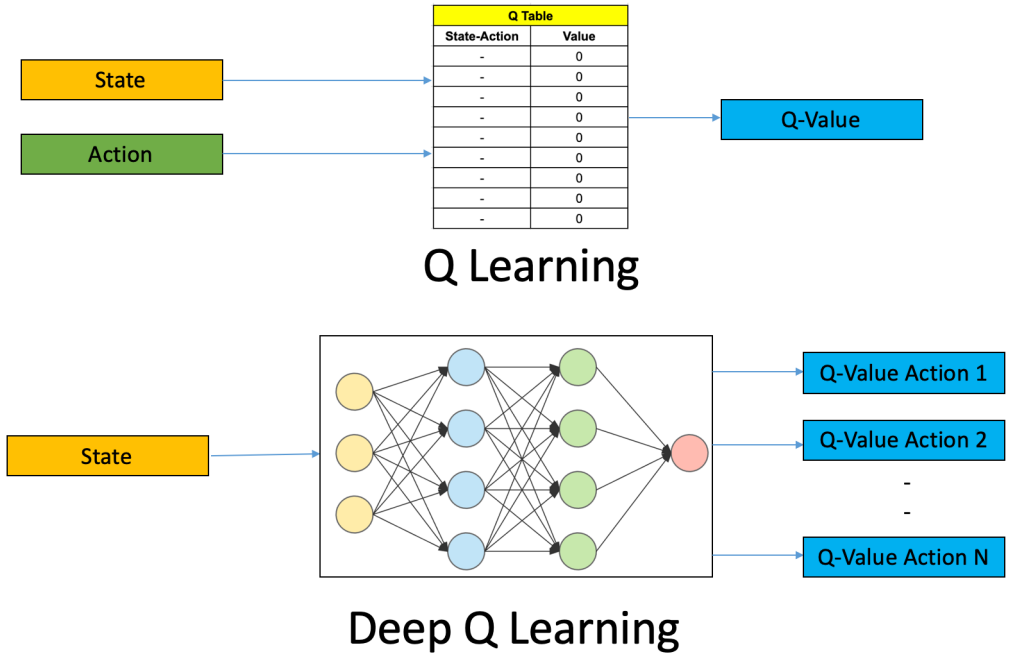

Abbildung 3.2: Deep Q-Network[6]

3.1.3 Experience Replay

Das Experience Replay ist ein Vorgang, um Trainingsdaten zu speichern und erneut wiederzugeben. Üblicherweise werden diese Trainingsdaten in der Form (s = Zustand, a = Aktion, r = Belohnung, s = Folgezustand) (kurz: $sars$-Tupel) abgespeichert, aus denen ein RL-Algorithmus lernen kann.

Deep Q-Learning verwendet Experience Replay, um in kleinen Batches zu lernen, die zufällig aus dem Trainingsdatensatz ausgewählt werden. Dieser Vorgang soll an das menschliche Lernen anlehnen und sorgt dafür, dass die Daten effizienter genutzt werden[16].

3.2 Monte Carlo Tree Search (MCTS)

Spiele wie Tic-Tac-Toe, Go, Schach und viele andere haben eine exponentiellen Zunahme der Anzahl möglicher Aktionen, die gespielt werden können. Im Idealfall kann jede mögliche Aktion und jeder zugehörige Folgezustand, der in der Zukunft auftreten kann, vorhergesagt werden. Da aber die Anzahl der Aktionen exponentiell zunimmt, steigt auch die Rechenleistung, die zur Berechnung erforderlich ist, exponentiell an.
MCTS ist eine Methode, die normalerweise in Spielen verwendet wird, um den Pfad (Folge von Aktionen) vorherzusagen, der die bestmögliche Aktion anhand einer Strategie ermittelt. Dabei wird ein Graph verwendet, dessen Knoten die Zustände und Kanten die Aktionen repräsentieren. Die verbundenen Zustände sehen wie ein Baum aus, woraus sich der Name „Tree Search“ herleitet. In Abbildung 3.3 ist ein Baum des MCTS-Algorithmus am Beispiel von Tic-Tac-Toe zu sehen. Dabei ist darauf zu achten, dass nicht alle Aktionen und Folgezustände dargestellt wurden.

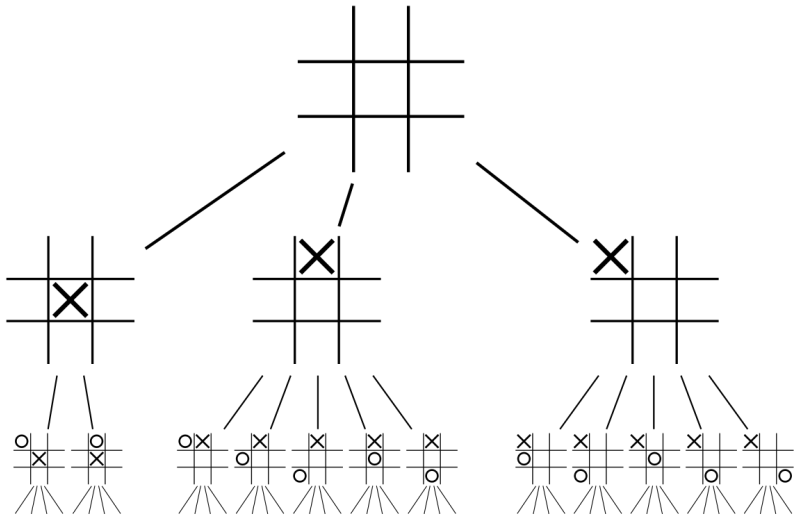

Abbildung 3.3: Beispiel eines MCTS-Baumes[7]

Die Suche nach einer endgültigen Lösung eines Problems in einem exponentiell wachsenden Baum unter Berücksichtigung aller Zustände erfordert eine enorme Rechenleistung. Eine schnellere Baumsuche kann erreicht werden, indem eine Strategie erstellt wird. Diese priorisiert einige Knoten höher als andere und sorgt dafür, dass ihre Folgezustände zuerst durchsucht werden, um die richtige Lösung zu finden.
Eine mögliche Strategie ist der MCTS-Algorithmus, welcher aus vier Einzelschritten besteht und die beste Aktion in einen gegebenen Zustand ermittelt. Die Einzelschritte Selection, Expansion, Simulation und Backpropagation werden in den nachfolgenden Kapiteln genauer erläutert. Sie werden so lange wiederholt, bis der Baum für gegebenes Problem groß genug ist, um eine zufriedenstellende Aktion auswählen zu können[17].

3.2.1 Selection-Schritt

Mit dem Selection-Schritt wird ein Knoten im Baum ausgewählt, der die höchste Gewinnchance hat. Diese wird mithilfe des UCB-Algorithmus (siehe Kapitel 3.2.5) ermittelt, welcher jede Aktion bzw. jeden Folgezustand bewertet[17].

Als Beispiel ist in Abbildung 3.4 ein Selection-Schritt veranschaulicht. Die Zahlen in den Knoten werden erst im Kapitel 3.2.3 erläutert und dienen in diesem Kapitel nur der Benennung der Zustände. Die beiden unterschiedlichen Farben der Knoten repräsentieren den jeweiligen Spieler und wechseln sich ab.

- Jeden Folgezustand „(7/10), (0/3), (3/8) “ des Initialzustandes „(11/21)“ wird ein UCB-Wert zugewiesen.
- Daraufhin wird der Zustand mit dem höchsten UCB-Wert, nämlich „(7/10)“, verwendet, der nun als neuer Ausgangspunkt dient.

Von hier aus wiederholen sich obige Stichpunkte bis ein Blatt, in diesem Fall „(3/3)“, erreicht ist.

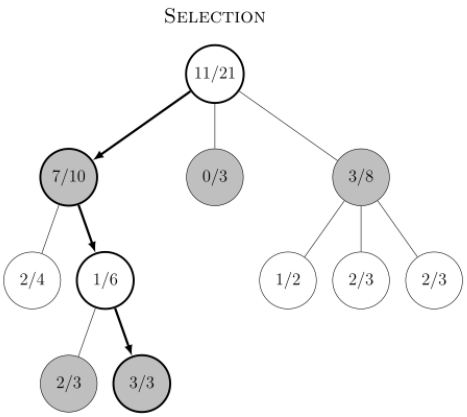

Abbildung 3.4: Selection-Schritt[8]

3.2.2 Expansion-Schritt

Der Expansion-Schritt ist dafür zuständig, den Baum zu vergrößern. Dabei wird das im Selection-Schritt ausgewählte Blatt durch jede legale Aktion inklusive den Folgezuständen erweitert[17].

Abbildung 3.5 zeigt die Erweiterung des Zustandes (3/3) um alle legalen Aktionen. In diesem Fall gibt es genau einen legalen Zug (0/0).

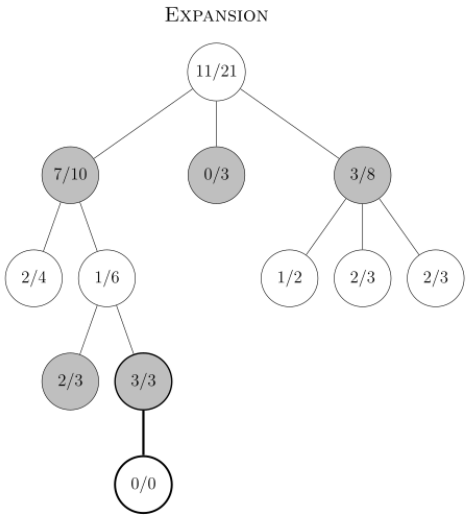

Abbildung 3.5: Expansion-Schritt[8]

3.2.3 Simulation-Schritt

Der Simulation-Schritt weist den expandierten Zuständen eine Belohnung zu. Das Ermitteln einer Belohnung ist hier nicht eindeutig. In der Literatur finden sich jedoch zwei gebräuchliche Wege, die anschließend erklärt werden.

Die Abbildung 3.6 zeigt einen sogeannten „Random Rollout“. Dabei wird ausgehend vom Blatt eine vollständige Partie mit zufälligen Aktionen gespielt. Abhängig vom Ausgang (Gewonnen, Verloren und evtl. Unentschieden) bekommen die Blätter eine entsprechende Belohnung bzw. Bestrafung zugewiesen.

Zum weiteren Verständnis müssen nun die Zahlen in den Knoten erklärt werden. Dabei beschreibt die rechte Zahl des Zustandes, wie oft dieser in allen Selection-Schritten ausgewählt wurde. Die linke Zahl hingegen gibt die Summe aller Belohnungen der Simulation-Schritte an. In diesem Beispiel liefert der Random Rollout die Zahlen (0/1). Da der Zustand das erste mal durchlaufen wurde bekommt die rechte Zahl den Wert 1 zugewiesen. Die linke Zahl ist abhängig vom Ergebnis des zufälligen Spiels und liefert den Wert 0, was in diesem Fall bedeutet, dass der weiße Spieler das Spiel verloren hat. Die beiden Zahlen werden zudem zur Berechnung des UCB-Wertes, welcher im Selection-Schritt berechnet wird, verwendet.

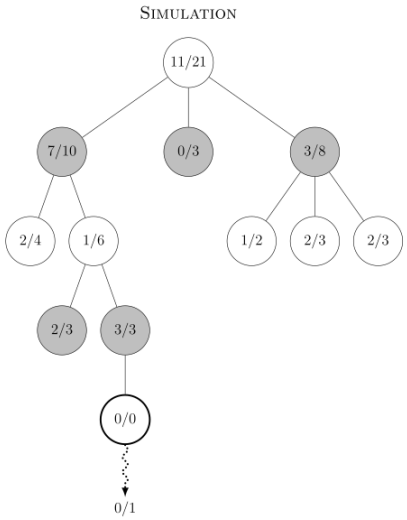

Abbildung 3.6: Simulation-Schritt[8]

Ein weiterer Weg zur Ermittlung der Belohnung kann mithilfe eines neuronalen Netzwerks erfolgen. Dieses liefert für einen gegebenen Zustand einen Belohnungswert. Für die Umsetzung des MCTS-Algorithmus wurde diese Variante gewählt (vgl. Kapitel 4.3).

3.2.4 Backpropagation-Schritt

Im Backpropagation-Schritt werden die Zahlen in den Knoten aktualisiert. Hierfür wird der Pfad der Aktionen aus dem Selection-Schritt rückwärts durchlaufen. Die rechte Zahl jedes Knotens, die die Anzahl der Durchläufe beschreibt, erhöht sich stets um eins. Die linken Zahlen werden abhängig von der Belohnung des neuen Blattes verändert. Dabei bekommen die gleichfarbigen Knoten die Belohnung und die ungleichfarbigen Knoten die Inverse der Belohnung aufaddiert.

Für das Beispiel in Abbildung 3.7 existieren die Belohnungen 0 (Verloren) und 1 (Gewonnen), also sind 0 und 1 invers zueinander. Für Spiele, die ein Unentschieden zulassen, werden gewöhnlicherweise die Belohnungen mit -1 (Verloren), 0 (Unentschieden) oder 1 (Gewonnen) belegt. In diesem Fall würde die Inverse gerade die additiv Inverse beschreiben. Die neuen aktualisierten Zahlen verändern damit den Zustand des Baumes und können auch die UCB-Werte neuer zukünftiger Knoten für den Selection-Schritt beeinflussen.

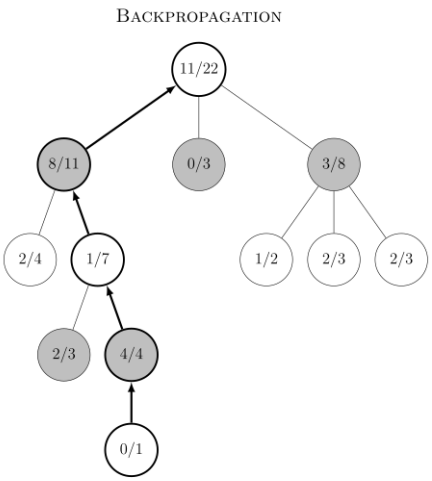

Abbildung 3.7: Backpropagation-Schritt[8]

3.2.5 UCB-Algorithmus

UCB steht für "Upper Confidence Bound" und wird zur Bewertung im Selection-Schritt verwendet. Die größte Schwierigkeit bei der Auswahl der Aktionen bzw. deren Folgezuständen besteht darin, ein gewisses Gleichgewicht zwischen Pfaden mit hoher erwarteter Belohnung und der Erkundung von Zügen mit wenigen Durchläufen zu halten. Allgemein wird folgende Formel für den UCB-Algorithmus angegeben[8]:

$$U(s,a) = \frac{W(s,a)}{N(s,a)} + const \cdot R(s,a)$$

- $W(s,a)$: Summe der Belohnungen für Aktion a im Zustand s (linke Zahl im Knoten)
- $N(s,a)$: Anzahl der Durchläufe von Aktion a im Zustand s (rechte Zahl im Knoten)
- $R(s,a)$: Funktion, die die Erkundung des Baumes beschreibt
- $const$: Hyperparameter zur Erkundung

Hierbei beschreibt der erste Summand $\frac{W(s,a)}{N(s,a)}$ die erwartete Belohnung für Aktion a im Zustand s. Der zweite Summand ist für die Erkundung des Baumes zuständig, wobei die Funktion $R(s,a)$ nicht eindeutig ist. Die in dieser Arbeit verwendete Definition von $R(s,a)$ wird in Kapitel 4 erläutert[18].

4 Realisierung

Das folgende Kapitel beinhaltet die Konvertierung von Schachnotationen für das neuronale Netzwerk sowie die Realisierungen der Algorithmen aus Kapitel 3.

4.1 Konvertierung von Schachnotationen

Schach (vgl. Abbildung 4.1) ist ein sehr komplexes Spiel mit enorm vielen Zuständen. Eine Schätzung für die Anzahl möglicher Schachpartien kann folgendermaßen aufgestellt werden. Mit der Annahme, dass im Mittel 33 mögliche Aktionen pro Zustand existieren und ein Spiel im Durchschnitt aus 80 Halbzügen besteht, ergibt sich für die Anzahl verschiedener Partien $33^{80} \approx 10^{120}$ Möglichkeiten[19].

Abbildung 4.1: Initialzustand eines Schachspiels[9]

Dadurch kann der Zustandsraum für den Computer nicht in Form einer Liste abgespeichert werden und es wird eine alternative Vorgehensweise gewählt. Die Schachpositionen (Zustände) und -züge (Aktionen) werden in Tensoren 3. Ordnung konvertiert um mit dem neuronalen Netz zu kommunizieren.

Zustände:
Jeder mögliche Zustand wird mithilfe eines (8, 8, 21)-Tensors beschrieben. Die ersten beiden Inidizes (8, 8) repräsentieren dabei das Schachbrett. Durch den dritten Index besteht der Tensor aus 21 dieser Schachbretter, wobei jedes eine der nachfolgenden Aufgaben erfüllt, um einen beliebigen Zustand eindeutig zu beschreiben.

- 1. bis 12. Schicht:
  Die ersten zwölf (8, 8)-Schichten geben die Positionen der Figuren an. Beispielsweise steht die erste Schicht für die weißen Bauern. Jede Position, auf der ein weißer Bauer steht, erhält eine 1. Alle anderen Stellen werden mit 0 belegt. So werden alle Positionen der zwölf unterschiedlichen Figuren sukzessiv beschrieben.

- 13. Schicht:
  Die 13. Schicht ordnet jedem Feld eine 1 zu auf dem ein En-Passant-Schlag möglich ist.

- 14. bis 17. Schicht:
  Diese vier Schichten sind für die Rochaderechte zuständig. Falls im gegebenen Zustand eine der Rochaden ausgeführt werden darf wird die jeweilige Schicht mit Einsen vorbelegt.

- 18. bis 19. Schicht:
  Um zu beschreiben welcher Spieler am Zug ist, werden im Falle von Weiß alle Felder der 18. Schicht und im Falle von Schwarz alle Felder der 19. Schicht mit Einsen vorbelegt. Die übrigen Felder erhalten wieder eine 0.

- 20. Schicht: Mit dieser Schicht wird die 50-Züge-Regel gespeichert. Falls es im nächsten Zug möglich ist das Spiel durch die 50-Züge-Regel zu beenden wird die 20. Schicht mit Einsen vorbelegt.

- 21. Schicht: Die letzte Schicht beschreibt die Regel der Stellungswiederholung (threefold repetition). Wie in der 20. Schicht wird die 21. Schicht mit Einsen vorbelegt, wenn im kommenden Zug eine Stellungswiederholung möglich ist.

Aktionen:
Jede mögliche Aktion wird durch einen (8, 8, 73)-Tensor beschrieben. Dabei wird eine konkrete Aktion, anders als im Fall des Zustands-Tensors, durch genau einen Eintrag beschrieben. Dabei definieren die ersten beiden Indizes (8, 8) die Anfangsposition der Figur auf dem Brett. Der dritte Index gibt eine der möglichen Bewegungen an, welche nachfolgend erklärt werden.

- 1. bis 56. Schicht:
  Von jeder Position aus existieren maximal acht mögliche, geradlinige Bewegungen die maximal sieben Felder weit reichen. Die ersten 56. Schichten ergeben sich somit aus dem Produkt $7 \cdot 8 = 56$.

- 57. bis 64. Schicht:
  Diese acht Schichten beschreiben alle möglichen Springerzüge.

- 65. bis 73. Schicht:
  Die letzten neun Schichten sind für die verschiedenen Bauernumwandlungen zuständig. Dabei wird die Umwandlung zur Dame in den ersten 56. Schichten angegeben. Die letzten neun Schichten dienen der Umwandlung in die Figuren Turm, Springer und Läufer, die sich jeweils in drei verschiedene Richtungen umwandeln können.

4.2 Deep Q-Network

Dieses Kapitel beschreibt die Umsetzung des DQNs und den verwendete Trainingsalgorithmus.

4.2.1 Neuronales Netz

Bei dem verwendeten NN handelt es sich um ein CNN mit Input-Tensor der Größe (8, 8, 21) und Output-Tensor der Größe (8, 8, 73) (vgl. Kapitel 4.1). Die Architektur des Netzes

sieht dabei wie folgt aus[20]:

- Erster Convolutional Layer ((3, 3)-Filter):
  In der ersten Schicht erhöht sich zunächst die Anzahl der Channel von 21 auf 256 und anschließend wird der Input mithilfe von Batch Normalization neu zentriert und skaliert. Als Aktivierungsfunktion wird die ReLU verwendet.

- 19 Residual Blocks:
  Jeder der 19 Blöcke hat die gleiche Struktur und besteht aus zwei der oben beschriebenen Convolutional Layers. Wobei in der zweiten Schicht vor der Batch Normalization zusätzlich eine Skip Connection zwischen dem Block-Input und dem Output der zweiten Schicht durchgeführt wird. Die Skip Connection bewirkt eine Addition der Tensoren. Das dient zur Vermeidung des sogenannten „vanishing gradient problem“, was bei großen Netzen auftreten kann.

- Letzter Convolutional Layer ((1, 1)-Filter):
  Diese Schicht ist dafür zuständig den Tensor auf die gewünschte Output-Größe von (8, 8, 73) zu bringen. Dabei wird ebenfalls eine Batch Normalization und eine ReLU verwendet.

Insgesamt liefert diese Architektur $\sim$ 22.5 Mio. trainierbare Parameter.

4.2.2 Algorithmus

Der Algorithmus für das Training des Netzes ist als Pseudocode in Abbildung 4.2 dargestellt. Es handelt sich um ein iterativen Algorithmus der schrittweise eine Liste, genannt „Memory“, von *sars*-Tupeln (vgl. Kapitel 3.1.3) aufbaut. Das Netzwerk wird nun mit der Memory und dem Vorgang des Experience Replay (vgl. Kapitel 3.1.3) trainiert. Dabei wird der Backpropagation-Algorithmus auf eine Loss-Funktion angewendet, welche aus dem MSE der Ist- und Sollwerte besteht. Die Ist-Werte sind dabei die Q-Werte der gegebenen Zustands-Aktions-Paare und die Soll-Werte berechnen sich mit einer modifizierten Variante der in Kapitel 3.1.1 vorgestellten Update-Regel. Das iterative Aufbauen der Memory sowie die Aktualisierung der Gewichte mithilfe von Backpropagation kann beliebig lange wiederholt

werden. Für weitere Informationen zum Trainingsalgorithmus wird auf das Paper [10] verwiesen.

Algorithm 1: deep Q-learning with experience replay.
Initialize replay memory D to capacity N
Initialize action-value function Q with random weights θ
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, M **do**
  Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
  For t = 1,T **do**
    With probability ε select a random action a_t
    otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D
    Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
    Perform a gradient descent step on $\left(y_j - Q(\phi_j, a_j; \theta)\right)^2$ with respect to the network parameters θ
    Every C steps reset $\hat{Q} = Q$
  End For
End For

Abbildung 4.2: Trainingsalgorithmus für das DQN[10]

4.2.3 Training

Der Trainingsalgorithmus wird auf verschiedene Arten umgesetzt, welche nachfolgend beschrieben werden:

- Datenbank-Training:
  Mithilfe der Großmeisterpartien aus https://www.pgnmentor.com/files.html wird, vor dem eigentlichen Training, die gesamte Memory erstellt, auf der anschließend trainiert wird. Dafür muss jeder einzelne Zug in Form eines *sars*-Tupels gespeichert werden. Die Wahl der Belohnung r ist dabei nicht eindeutig. Für diese Arbeit werden die letzten zehn Züge linear aufsteigend von 0 bis 1 bzw. linear absteigend von 0 bis -1 bewertet. Die übrigen Züge erhalten keine Belohnung.

- Selfplay-Training:
  Im Falle des Selfplay-Trainings spielt das NN gegen sich selbst, wobei die Memory iterativ aufgebaut wird. Die Züge müssen anschließend analog zum Datenbank-Training in $sars$-Tupel übersetzt werden. Als Belohnung r wird ausschließlich der terminale Zug mit -1, 0 oder 1, abhängig vom Spielausgang, bewertet.

- Stockfish-Training:
  Für das Stockfish-Training spielt das NN gegen die starke Schach-Engine Stockfish, wobei, wie im Fall des Selfplay-Training, nur die terminalen Züge bewertet werden.

Alle drei Trainingsarten sind mit Nachteilen verbunden. Das Datenbank-Training ist nicht in der Lage zu verallgemeinern, was dem riesigen Zustandsraum geschuldet ist. Das Selfplay-Training ist hingegen sehr langsam, da durch anfänglich zufällige Züge des NNs das Spiel häufig im Unentschieden endet. Im Fall des Stockfish-Trainings verliert das NN logischerweise jedes Spiel und erhält nur Bestrafungen. Da es im Schach normalerweise mehr schlechte als gute Züge gibt, ist es für das Training von Vorteil Belohnungen zu erhalten. Damit kann Rechenzeit eingespart werden.

Um den Nachteilen der einzelnen Trainingsarten entgegen zu wirken, wird das Datenbank-Training mit dem Selfplay-Training kombiniert. Hierbei wird zunächst mithilfe des Datenbank-Trainings dem NN das Eröffnungsverhalten von Schachgroßmeistern beigebracht. Anschließend verwendet das NN Selfplay-Training, um komplexe Zusammenhänge im Zustandsraum zu verallgemeinern.

4.3 Monte-Carlo-Tree-Search

4.3.1 Neuronales Netz

Die Architektur des verwendeten NNs ist ähnlich aufgebaut wie das DQN (siehe Kapitel 4.2.1), unterscheidet sich jedoch hauptsächlich im Output. Wie bereits in Kapitel 3.2.3 erwähnt, wird das NN und der MCTS-Algorithmus kombiniert. Hierfür wird ein zusätzlicher Output für die Bewertung des Zustands benötigt. Außerdem wird als Aktivierungsfunktion aller Convolutional Layer die Leaky ReLU gewählt, deren Gewichte regularisiert (siehe Kapitel 4.3.3) und Batch Normalization angewendet[20]:

- Erstes Convolutional Layer ((3, 3)-Filter):
  Analog zu DQN
- 19 Residual Blocks:
  Analog zu DQN
- Aufteilung in Policy- und Value-Head:
  - Policy-Head:
    Der Policy-Head nutzt zwei Convolutional Layers ((3, 3)-Filter) um die nötige Output-Größe von (8, 8, 73) zu erhalten und wird für die Berechnung des UCB-Wertes verwendet (vgl. Kapitel 4.3.2).
  - Value-Head:
    Der Value-head hat die Aufgabe eine reelle Zahl zwischen -1 und 1 zu generieren. Hierfür wird der Input, also ein Tensor 3. Ordnung, auf einen Tensor 0. Ordnung reduziert. Die letzte Aktivierungsfunktion ist der Tangens hyperbolicus, um den gewünschten Output zu erhalten. Dieser liefert im Simulation-Schritt des MCTS-Algorithmus den Belohnungswert für die Knoten.

Insgesamt liefert diese Architektur $\sim$ 23.5 Mio. trainierbare Parameter.

4.3.2 UCB-Algorithmus

Aus Kapitel 3.2.5 ist folgende Definition des UCB-Wertes bekannt:

$$U(s,a) = \frac{W(s,a)}{N(s,a)} + const \cdot R(s,a)$$

Für den Erkundungssummanden $R(s,a)$ wurde folgende Definition gewählt[18]:

$$R(s,a) = P(s,a)\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$$

- $P(s,a)$: Policy-Output des NNs für Aktion a im Zustand s
- $N(s,a)$: Anzahl der Durchläufe von Aktion a im Zustand s

Durch den Faktor $P(s,a)$ nimmt der trainierbare Output des Policy-Heads Einfluss auf die Erkundung des Baumes. Der Bruch $\frac{\sqrt{\sum_b N(s,b)}}{1+N(s,a)}$ sorgt dafür, dass selten erkundete Zustände einen größeren UCB-Wert erhalten und dadurch wahrscheinlicher durchlaufen werden.

4.3.3 Loss-Funktion

Die Loss-Funktion setzt sich aus einer Linearkombination des Value- und Policy-Losses sowie einen Regularisierungsterm zusammen[21].

- Value-Loss:
  Für die Loss-Funktion des Value-Heads wird der MSE gewählt. Hierbei ist der Ausgang des Schachspiels (Gewonnen $\hat{=}$ 1, Unentschieden $\hat{=}$ 0, Verloren $\hat{=}$ -1) mit dem Output des Value-Heads zu vergleichen.

- Policy-Loss:
  Im Policy-Loss werden die Soll- und Ist-Werte mittels Kreuzentropie verglichen. Die Ist-Werte erhält man durch Anwendung einer Softmax-Funktion auf den Output des Policy-Heads, nachdem alle illegalen sowie nicht durchlaufenen Aktionen im Output auf -100 gesetzt werden. Das sorgt dafür, dass irrelevante Züge keinen Einfluss haben. Die Soll-Werte ergeben sich aus dem gewichteten (8, 8, 73)-Tensor aller $N(s,a)$-Werte des Zustands s, um eine Wahrscheinlichkeitsverteilung zu erhalten.

- Regularisierungsterm:
  Zur Vermeidung von Überanpassung wird der Loss-Funktion ein L2-Regularisierungsterm angehängt.

Insgesamt ergibt sich für die Loss-Funktion

$$\text{Loss} = \tfrac{1}{2} \cdot \text{Value-Loss} + \tfrac{1}{2} \cdot \text{Policy-Loss} + const_{Reg} \cdot \|\text{Gewichte}\|_2,$$

wobei $const_{Reg}$ ein Hyperparameter zur Gewichtung des Regularisierungsterms ist.

4.3.4 Training

Der Ablauf des Trainings mit dem MCTS-Algorithmus findet folgendermaßen statt:

Zunächst wird ein NN mit zufälligen Gewichten namens „best_player“ sowie eine Kopie von best_player namens „current_player“ erstellt.

- Selfplay:
  In diesem Schritt spielt best_player gegen sich selbst, wobei in jedem Spiel ein eigener Baum für den MCTS-Algorithmus erstellt wird:

  1. Für jeden Zug werden die vier Schritte des MCTS-Algorithmus beliebig oft wiederholt, um den Baum iterativ zu vergrößern.
  2. Daraufhin wird die Aktion mit den höchsten $N(s, a)$-Wert ausgeführt, wobei dessen Folgezustand als neuer Startpunkt für den MCTS-Algorithmus dient.

  Die Schritte 1. und 2. werden anschließend wiederholt bis das Spiel vorbei ist. Nach jedem Spiel wird dann für jeden Zug ein Eintrag in der Memory angelegt. Daraus lassen sich Ist- und Soll-Werte für die Loss-Funktion generieren:

  – Ist-Werte:
    Hierfür wird lediglich der Zustand in der Memory gespeichert. Die Ist-Werte ergeben sich dann aus den beiden Outputs des NNs.

  - Soll-Werte:
    Da der Ausgang des Schachspiels (Gewonnen $\hat{=}$ 1, Unentschieden $\hat{=}$ 0, Verloren $\hat{=}$ -1) mit dem Output des Value-Heads verglichen wird, wird der Spielausgang als Soll-Wert abgespeichert.
    Für die Soll-Werte des Policy-Heads werden die gewichteten $N(s, a)$-Werte benötigt (vgl. Kapitel 4.3.3), welche in der Memory gespeichert werden.

- Trainieren von current_player:
  Für das Training wird zunächst die Loss-Funktion auf die Ist- und Soll-Werte, die aus der Memory generiert werden, angewendet. Daraufhin werden mithilfe des Adam-Optimierers und dem Backpropagation-Algorithmus die Gewichte des NNs aktualisiert. Das Training erfolgt hierbei nach dem Schema des Experience Replay (vgl. Kapitel 3.1.3).

- Evaluation:
  Nach einer bestimmten Anzahl von Spielen wird current_player evaluiert, indem er mehrfach gegen den untrainierten best_player spielt. Falls current_player häufiger gewinnt, wird best_player mit einer Kopie von current_player überschrieben.

Diese Schritte können daraufhin beliebig oft wiederholt werden, wodurch best_player immer besser wird.

4.3.5 Hyperparameter

Das Projekt liefert eine Reihe von verstellbaren Hyperparametern, welche in Abbildung 4.3 dargestellt sind. Die Optimierung dieser Parameter ist sehr aufwendig und nachfolgend werden drei von ihnen genauer erläutert[22]:

- LEARNING RATE:
  Die Lernrate hat einen entscheidenden Einfluss auf das Verhalten des Trainings. Dabei liefern zu große Werte (>0.001) unbrauchbare Ergebnisse, wodurch die Schrittweite zu groß ist und Minima in der Loss-Funktion übersprungen werden. Falls die Lernrate jedoch zu klein gewählt wird, kann das Training sehr lange dauern. Nach mehrmaligen Testen ist der Wert 0.00002 gewählt worden.

- CONST:

  Der Parameter CONST ist für die Gewichtung des Erkundungssummanden im UCB-Algorithmus zuständig. Falls der Wert zu klein gewählt wird, enden die Spiele häufig im Unentschieden, da nicht viele verschiedene Aktionen durchsucht werden. Umgekehrt sorgt ein zu großer Wert dafür, dass der Baum zu breit ist und nicht tief genug erkundet wird. Dadurch kann das NN nicht vorausschauend agieren. Als Kompromiss ist der Wert 3.5 gewählt worden.

- NUM_SIMULATIONS:

  Dieser Parameter gibt an wie häufig die vier Schritte des MCTS-Algorithmus ausgeführt werden, bevor eine Aktion ausgewählt wird. Logischerweise liefert ein großer Wert bessere Ergebnisse, benötigt aber auch einen größeren Rechenaufwand. Mit einer gewöhnlichen Hardware sind zwar Werte bis zu 100 möglich, die Auswahl einer Aktion benötigt dann aber etwa 20 Sekunden.

```
############ Hyper Parameters ############

MODEL_UPDATES = 20            # Number of times the model is updated
NUM_RES_LAYER = 20            # Number of Residual Layer for the model

#### SELF PLAY
EPISODES = 15                 # Amount of selfplay-games before updating the model
NUM_SIMULATIONS = 50          # Amount of simulations each turn
MEMORY_SIZE = 30000           # Maximum size of memory
TURNS_UNTIL_NOT_RANDOM = 8    # Amount of turns before model plays deterministically
CONST = 3.5                   # Constant for MCTS-algorithm
EPSILON = 0.2                 # Parameter for MCTS-algorithm
ALPHA = 0.9                   # Parameter for MCTS-algorithm

# RETRAINING
STARTING_MEMORY_SIZE = 2000   # Size of memory before model starts learning
BATCH_SIZE_1 = 1024           # Size of minibatch from memory (minibatch1)
BATCH_SIZE_2 = 256            # Size of minibatch from minibatch1 (minibatch2)
EPOCHS = 2                    # Number of times the minibatch2 is used for updating the model parameters
TRAINING_LOOPS = 20           # Number of times we sample minibatch1 from memory for updating the model
REG_CONST = 0.00001           # Weight of the L2-regularization term
LEARNING_RATE = 0.00002       # Learning rate for Adam optimizer
BETA_1 = 0.9                  # Parameter for Adam optimizer
BETA_2 = 0.999                # Parameter for Adam optimizer
CLIP_NORM = 1                 # Gradient Clipping parameter

# EVALUATION
START_EVALUATION = 4          # After (START_EVALUATION * EPISODES) Games the model gets evaluated
EVAL_EPISODES = 10            # Amount of games played to determine the better player after updating one model
SCORING_THRESHOLD = 1.3       # The new model is used if: new_model_wins/old_model_wins > SCORING_THRESHOLD
```

Abbildung 4.3: Verwendete Hyperparameter im MCTS-Algorithmus

5 Dokumentation

Der Quellcode des MCTS wurde auf GitHub veröffentlicht und kann unter folgendem Link eingesehen werden: https://github.com/TimSelig/ChessAI

Dieses Kapitel liefert eine kurze Zusammenfassung des Quellcodes, inklusive der verwendeten Bibliotheken:

Wichtige Bibliotheken:

- chess: Diese Bibliothek wird für Schach-relevante Funktionen verwendet, beispielsweise zur Bestimmung aller legalen Züge einer bestimmen Position. Die Zustände sind in sogenannten „FEN-Strings“ gespeichert, eine Schachnotation, mit der beliebige Positionen angegeben werden können.
- tensorflow: Für das Erstellen des NNs sowie dessen Training wird die Bibliothek tensorflow genutzt.
- numpy: Die Zustände und Aktionen wurden stattdessen mithilfe der numpy-Bibliothek konvertiert, da diese einfacher zu bedienen ist.

Hyperparameter:
Das Projekt hat einige verstellbare Hyperparameter, mehr dazu kann in Kapitel 4.3.5 nachgelesen werden.

Konvertierung der Schachnotationen:
Zum Verständnis folgender Funktionen wird auf das Kapitel 4.1 verwiesen.

- FEN_converter(board):
  Die Funktion konvertiert eine Klasse „board“ in einen (8, 8, 21)-Tensor für das NN. Diese Klasse wird in der Bibliothek python-chess verwendet und beinhaltet den FEN-String eines Zustandes sowie weitere Informationen, wie die Stellungswiederholung und die 50-Züge-Regel.

- array_index_to_move(board, action):
  In dieser Funktion wird ein Index-Tupel der Form (-, -, -), der eine Aktion im (8, 8, 73)-Output-Tensor beschreibt, zu einem UCI-String (Universal Chess Interface) (Bsp.: „e2e4“, „d7d8q“) konvertiert.

- move_to_array_index(move):
  Diese Funktion wandelt einen UCI-String in einen Index-Tupel der Form (-, -, -) um.

Neuronales Netzwerk:
Die Klasse „myModel()“ definiert den Aufbau des CNNs, welches im Kapitel 4.3.1 beschrieben wird. Zusätzlich wird die Loss-Funktion, deren Gewichte und der verwendete Adam-Optimierer angegeben.

Loss-Funktion des Policy-Heads:
Die Loss-Funktion des Policy-Heads „softmax_cross_entropy_with_logits(y_true, y_pred)“ ist eine benutzerdefinierte Funktion, die im Kapitel 4.3.3 erklärt wird.

Umgebung:
Die Umgebung des Schachspiels wird in der Klasse „Game“ definiert. Diese dient zum Speichern von Zuständen, ist in der Lage Aktionen auszuführen, kann eine Liste aller legalen Züge zurückgeben und enthält weitere nützliche Funktionen zum Arbeiten in der Schachumgebung.

MCTS-Baum

Zur Beschreibung des MCTS-Baumes werden drei Klassen verwendet.

- Die Klasse „Node()“ repräsentiert die Knoten des Baumes und speichert dessen Zustände.
- Die Klasse „Edge()“ repräsentiert die Kanten des Baumes und speichert die Aktionen. Zusätzlich werden die veränderlichen Größen $N(s, a)$, $W(s, a)$ und $P(s, a)$ (vgl. Kapitel 4.3.2) für jede Kante gespeichert.
- Die Klasse „MCTS()“ beinhaltet die Struktur des Baumes sowie Funktionen, die für den Selection-, Expansion- und Backpropagation-Schritt des MCTS-Algorithmus notwendig sind.

Agent

Die Klasse „Agent()“ beinhaltet ein NN sowie eine Funktion um dieses zu trainieren. Des Weiteren ist die Klasse für die Ausführung des MCTS-Algorithmus zuständig. Hierbei führt die Funktion „act(self, state, tau)“, mithilfe der restlichen Funktion, die vier Schritte des MCTS-Algorithmus mehrfach aus und wählt die Aktion mit dem größten $N(s, a)$-Wert. In der Funktion „replay(self, ltmemory)“ wird das NN mit der Tensorflow-Funktion „fit“ trainiert und die Loss-Funktionen graphisch dargestellt.

Memory

Die Klasse „Memory“ speichert $sars$-Tupel (vgl. Kapitel 3.1.3) in Listen. Hierfür gibt es die Langzeit-Memory „ltmemory“ und die Kurzzeit-Memory „stmemory“. Die stmemory beinhaltet keine Belohnungen r, da sie während dem Spiel angelegt wird und die Belohnungen erst nach dem Spiel zur Verfügung stehen. Nach jedem Spiel werden die Zustands-Aktions-Paare der stmemory inklusive deren Belohnungen in die ltmemory in Form von $sars$-Tupeln gespeichert und die stmemory geleert.

Mensch gegen Agent

Um gegen das NN zu spielen, gibt es die Funktion „model_eval(player, goes_first = 0)“, dabei gibt der zweite Parameter an, welcher Spieler beginnt.

Agent gegen Agent

Mithilfe der Funktion „playMatches(player1, player2, episodes, turns_until_not_random, memory = None, goes_first = 0)“ können zwei Agenten mehrfach gegeneinander spielen. Dabei wird die Memory für das spätere Training aufgebaut und die Spielergebnisse übergeben.

Training

Die Funktion „run(old_memory = False, old_model = False)“ wird in Kapitel 4.3.4 detailliert beschrieben.

6 Ergebnisse

Die Abbilung 6.1 visualisiert die Gesamt-Loss-Funktion sowie deren gewichtete Einzelsummanden aus Kapitel 4.3.3. Auf der x-Achse sind die Anzahl der Trainingsiterationen aufgetragen, was für den verwendeten Parametersatz (vgl. Abbildung 4.3) gleichbedeutend mit 150-180 Spielen ist.

- Value-Loss:
  Mit den verwendeten Hyperparametern werden zunächst 15 Spiele gespielt, um die Memory aufzubauen, woraufhin das NN über 20 Iterationen trainiert wird. Das wechselnde Aufbauen der Memory und Trainieren des NNs sorgt im Graphen alle 20 Iterationen für Spitzen. Da die Memory für die ersten Trainingsiterationen nicht groß genug ist, kann die Bewertung der Zustände nicht gut verallgemeinert werden. Dadurch schlägt die Loss-Funktion, durch Hinzunahme neuer $sars$-Tupel, höher aus. Dieses Verhalten lässt mit größer werdender Memory nach und die Spitzen werden kleiner.

- Policy-Loss:
  Im Vergleich zum Value-Loss hat der Policy-Loss einen größeren Einfluss auf den Gesamt-Loss, was den absoluten Werten der y-Achse geschuldet ist. Das hat zur Folge, dass der Value-Loss global gesehen konstant erscheint. Im Policy-Loss ist hingegen nach einer kurzen Einschwingphase eine Tendenz nach unten zu erkennen.

- Regularisierungs-Loss:
  Der Regularisierungs-Loss dient der Vermeidung von Überanpassung an die Daten und hat keinen großen Einfluss auf den Gesamt-Loss.

Insgesamt kann im Gesamt-Loss eine klare Tendenz nach unten erkannt werden. Zusätzlich ist das trainierte NN in der Lage das untrainierte, zufällige NN nach 200 Trainingsiterationen zu schlagen.

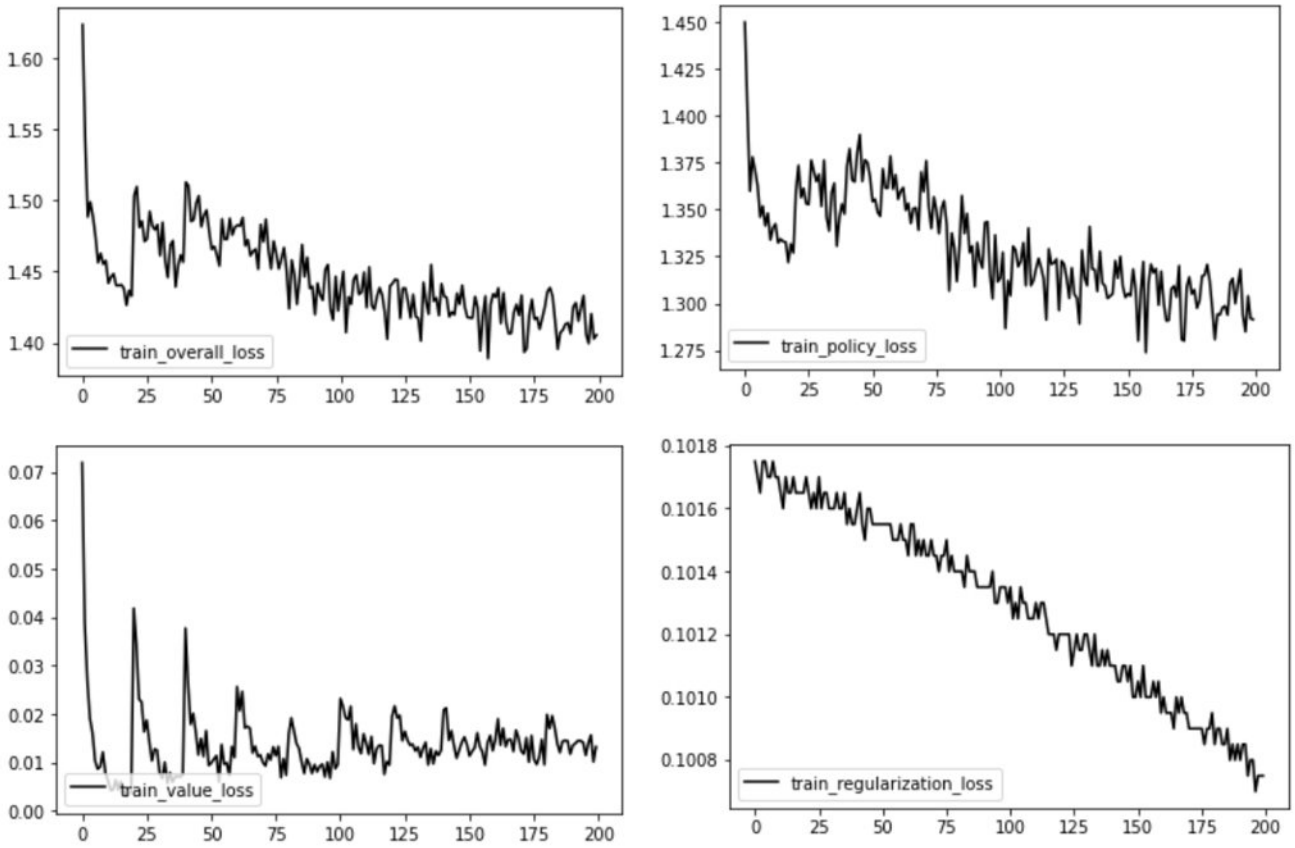

Abbildung 6.1: Graphen der Loss-Funktionen über 200 Trainingsiterationen: Gesamt-Loss, Policy-Loss, Value-Loss, Regularisierungs-Loss

7 Fazit

7.1 DQN

Für die Optimierung der Hyperparameter sowie das Training des DQNs konnte nicht viel Zeit investiert werden, da der Schwerpunkt auf den MCTS-Algorithmus gelegt wurde. Dennoch hat das Datenbank-Training gezeigt, dass das NN konkrete Züge von Schachgroßmeistern, insbesondere in der Eröffnungsphase, wiedergeben kann. Aufgrund des riesigen Zustandsraumes ist das Netzwerk nach kurzem Training nicht in der Lage alle komplexen Zusammenhänge zu lernen. Hierfür sind jedoch weiterer Untersuchungen nötig.

7.2 MCTS

Für die Optimierung der Hyperparameter ist bereits eine Menge Zeit investiert worden. Da es sich jedoch um ein sehr komplexes Modell mit vielen verstellbaren Parametern handelt, sind weitere Änderungen notwendig, um dieses zu verbessern. Des Weiteren ist zu untersuchen, ob eine Anpassung des Modells sinnvoll ist. Hierfür könnte beispielsweise die L2-Regularisierung durch einen Dropout ersetzt oder der Input-Tensors durch vergangene Zustände erweitert werden. Das größte Verbesserungspotential sehen wir in der verwendeten Hardware. Die Ergebnisse wurden mithilfe einer „RTX 2070 Super“-Grafikkarte, 16GB Arbeitsspeicher und einem „Intel Core i5-9600k“-Prozessor berechnet. Das Training des NNs aus Kapitel 6 hat bereits eine Woche in Anspruch genommen. Insgesamt ist bei diesem Trend im Vergleich zu ähnlichen NNs davon auszugehen, dass die gesamte Trainingsdauer etwa 6-12 Monate beträgt, bis Leistungen menschlicher Durchschnittsspieler erwartet werden können.

Literaturverzeichnis

[1] NOVUSTAT: *Künstliches neuronales Netz einfach erklärt: Lernen im Data Mining*. https://novustat.com/statistik-blog/kuenstliches-neuronales-netz-einfach-erklaert.html. – Aufgerufen am 29.07.2021

[2] Bui, Huy: *From Convolutional Neural Network to Variational Auto Encoder*. https://medium.com/analytics-vidhya/from-convolutional-neural-network-to-variational-auto-encoder-97694e86bb51. – Aufgerufen am 30.07.2021

[3] Saha, Sumit: *A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way*. https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53. – Aufgerufen am 30.07.2021

[4] MathWorks: *Reinforcement Learning Agents*. https://www.mathworks.com/help/reinforcement-learning/ug/create-agents-for-reinforcement-learning.html. – Aufgerufen am 30.07.2021

[5] LearnDataSci: *Q-learning*. https://en.wikipedia.org/wiki/Q-learning#Algorithm. – Aufgerufen am 09.06.2021

[6] Choudhary, Ankit: *A Hands-On Introduction to Deep Q-Learning using OpenAI Gym in Python*. https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/. – Aufgerufen am 26.07.2021

[7] Borges, Pedro: *AI: Monte Carlo Tree Search (MCTS).* https://medium.com/@pedrohbtp/ai-monte-carlo-tree-search-mcts-49607046b204. – Aufgerufen am 26.07.2021

[8] Rmoss92: *Monte Carlo tree search.* https://en.wikipedia.org/wiki/Monte_Carlo_tree_search. – Aufgerufen am 27.07.2021

[9] chess.com: *chess.com.* https://www.chess.com/analysis. – Aufgerufen am 27.07.2021

[10] Volodymyr, Mnih; Koray Kavukcuoglu; David Silver; Andrei A. Rusu; Joel Veness; Marc G. Bellemare; Alex Graves; Martin Riedmiller; Andreas K. Fidjeland; Georg Ostrovski; Stig Petersen; Charles Beattie; Amir Sadik; Ioannis Antonoglou; Helen King; Dharshan Kumaran; Daan Wierstra; Shane Legg; Demis H.: Human-level control through deep reinforcement learning. In: *nature* 518 (2015), 529–531 + Anhang. https://www.nature.com/articles/nature14236. – Aufgerufen am 27.07.2021

[11] Wilczok, Elke: *Maschinelles Lernen.* – Aufgerufen am 29.07.2021

[12] Wuttke, Laurenz: *Reinforcement Learning: Wenn KI auf Belohnungen reagiert.* https://datasolut.com/reinforcement-learning/#Was-ist-Reinforcement-Learning. – Aufgerufen am 30.07.2021

[13] Wilczok, Elke: *Bestärkendes Lernen/Reinforcement Learning.* – Aufgerufen am 30.07.2021

[14] Heinz, Sebastian: *Einführung in Reinforcement Learning – wenn Maschinen wie Menschen lernen.* https://www.statworx.com/de/blog/einfuehrung-in-reinforcement-learning-wenn-maschinen-wie-menschen-lernen/. – Aufgerufen am 09.06.2021

[15] Nicholson, Chris: *A Beginner's Guide to Deep Reinforcement Learning.* https://wiki.pathmind.com/deep-reinforcement-learning. – Aufgerufen am 26.07.2021

[16] Wang, Mike: *Deep Q-Learning Tutorial: minDQN.* https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc. – Aufgerufen am 26.07.2021

[17] Sharma, Sagar: *Monte Carlo Tree Search*. https://towardsdatascience.com/monte-carlo-tree-search-158a917a8baa. – Aufgerufen am 26.07.2021

[18] Nair, Surag: *A Simple Alpha(Go) Zero Tutorial*. https://web.stanford.edu/~surag/posts/alphazero.html. – Aufgerufen am 28.07.2021

[19] SH: *Schach in Zahlen*. https://schachlich.de/schach-in-zahlen/. – Aufgerufen am 27.07.2021

[20] Foster, David: *AlphaGo Zero Cheat Sheet*. https://medium.com/applied-data-science/alphago-zero-explained-in-one-diagram-365f5abf67e0. – Aufgerufen am 28.07.2021

[21] Foster, David: *How to build your own AlphaZero AI using Python and Keras*. https://medium.com/applied-data-science/how-to-build-your-own-alphazero-ai-using-python-and-keras-7f664945c188. – Aufgerufen am 28.07.2021

[22] Prasad, Aditya: *Lessons from AlphaZero (part 3): Parameter Tweaking*. https://medium.com/oracledevs/lessons-from-alphazero-part-3-parameter-tweaking-4dceb78ed1e5. – Aufgerufen am 28.07.2021

Eidesstattliche Erklärung

Hiermit versichern wir, die vorliegende Projektarbeit selbstständig und nur unter Verwendung der von uns angegebenen Quellen und Hilfsmittel verfasst zu haben. Sowohl inhaltlich als auch wörtlich entnommene Inhalte wurden als solche kenntlich gemacht. Die Arbeit hat in dieser oder vergleichbarer Form noch keinem anderem Prüfungsgremium vorgelegen.

Datum: 31.07.20 Unterschrift: T. Seby

Datum: 31.07.20 Unterschrift: S. Schramm