FOM Hochschule für Oekonomie & Management

Hochschulzentrum Bonn

Bachelor-Thesis
im Studiengang Wirtschaftsinformatik

zur Erlangung des Grades eines

Bachelor of Science (B.Sc.)

über das Thema

Reinforcement Learning: Eine Auswahl, Implementation und Analyse verschiedener Reinforcement Learning Algorithmen für hochdimensionale Probleme

von

Andreas Kretschmer

Erstgutachter: Dr. rer. Pol. Julian Schröter M.Sc.

Matrikelnummer: 463466

Abgabedatum: 12.04.2021

Inhaltsverzeichnis

Abbildungsverzeichnis ..... **III**

Tabellenverzeichnis ..... **IV**

Algorithmen Verzeichnis ..... **V**

Symbolverzeichnis ..... **VI**

Formelverzeichnis ..... **VIII**

Abkürzungsverzeichnis ..... **IX**

1. Einleitung **1**

1.1 Motivation **1**

1.2 Zielsetzung **2**

1.3 Vorgehen **2**

2. Grundlagen **3**

2.1 Reinforcement Learning **3**

2.2 Markov Entscheidungsprozess **4**

2.3 Funktionen zum Verbessern des Verhaltens **6**

2.3.1 Regelfunktion 6

2.3.2 Wertfunktion 7

2.3.3 Wertfunktions-Algorithmus: Q-Learning 9

2.3.4 Regelfunktions-Algorithmus: REINFORCE 10

2.4 Erkundung oder Ausnutzung **12**

2.5 Deep Reinforcement Learning **13**

2.6 Neuronale Netze **14**

2.6.1 Neuron 14

2.6.2 Tiefes Neuronales Netz 15

2.6.3 Faltende Neuronale Netze 16

2.7 Deep Q Network **18**

2.8 Asynchronous Advantage Actor Critic **20**

3. Experiment **22**

3.1 Aufbau des Experiments **23**

3.2 Vorverarbeitung **25**

3.3 Implementation **28**

3.3.1 Main 30
3.3.2 DQN Algorithmus 30
3.3.3 A3C Algorithmus 33
3.3.4 Hyperparameter 36

4. Evaluation 38
4.1 Trainingsergebnisse 39
4.2 Probleme 42
4.3 Evaluationsergebnisse 43
4.4 Bewertung 44

5. Limitation 45

6. Zusammenfassung und Ausblick 47

Quellenverzeichnis i

Abbildungsverzeichnis

Abbildung 2.1: Interaktion des RL Agenten mit der Umgebung 3

Abbildung 2.2: Tiefes Neuronales Netz 16

Abbildung 2.3: Beispielhafte Faltungsoperation 17

Abbildung 3.1: Vergleich originales Bild und erste Vorverarbeitung 26

Abbildung 3.2: Vorverarbeitetes Bild in Graustufe in der Größe 84x84x1 Pixel 27

Abbildung 3.3: Komponenten der Implementation 29

Abbildung 3.4: DQN Klassenmodell 30

Abbildung 3.5: A3C Klassenmodell 33

Abbildung 4.1: Trainingsverlauf DQN Agenten 39

Abbildung 4.2: Trainingsverlauf A3C Agenten 41

Abbildung 4.3: Überanpassung des A3C Modells 42

Tabellenverzeichnis

Tabelle 3.1: Aktionsraum von Breakout 24

Tabelle 3.2: allgemeine Hyperparameter 36

Tabelle 3.3: DQN Hyperparameter 37

Tabelle 3.4: A3C Hyperparameter 38

Tabelle 4.1: Evaluationsergebnisse 43

Algorithmen Verzeichnis

Algorithmus 2.1: Q-Learning Pseudocode 10

Algorithmus 2.2: REINFORCE Pseudocode 12

Algorithmus 2.3: DQN Pseudocode 19

Algorithmus 2.4: A3C Pseudocode für jeden Agenten 22

Symbolverzeichnis

s	Zustand
a	Aktion
r	Belohnung
S	Menge aller möglichen Zustände
A	Menge aller möglichen Aktionen
$A(s)$	Menge der mögliche Aktionen in Zustand s
R	Menge der mögliche Belohnungen
θ	Menge an Funktionsparametern
t	diskreter Zeitschritt
T	terminierender Zeitschritt einer Episode
S_t	Zustand zum Zeitpunkt t
A_t	Aktion zum Zeitpunkt t
R_t	Belohnung zum Zeitpunkt t abhängig von A_{t-1} und S_{t-1}
π	stochastische Regelfunktion
μ	deterministische Regelfunktion
$\pi(a\|s)$	Wahrscheinlichkeit der Aktion a im Zustand s unter der stochastischen Regelfunktion π
$\pi(a\|s,\theta)$	Wahrscheinlichkeit der Aktion a im Zustand s unter der stochastischen Regelfunktion π mit den Funktionsparametern θ
$\mu(s)$	ausgewählte Aktion im Zustand s unter der deterministische Regelfunktion μ
$\mu(s,\ \theta)$	ausgewählte Aktion im Zustand s unter der deterministische Regelfunktion μ mit den Funktionsparametern θ
$p(s', r\|s, a)$	Wahrscheinlichkeit aus dem Zustand s mit der Aktion a in Zustand s' mit der Belohnung r zu wechseln
$\upsilon_\pi(s)$	Wert des Zustandes s unter der Regelfunktion π (erwartete Belohnung)
$\upsilon^*(s)$	Wert des Zustandes s unter der optimalen Regelfunktion π

$Q_\pi(s,a)$	Wert des Ausführens der Aktion a in Zustand s unter der Regelfunktion π
$Q_\pi(s,a,\theta)$	Wert des Ausführens der Aktion a in Zustand s unter der Regelfunktion π mit den Funktionsparametern θ
$Q^*(s,a)$	Wert des Ausführens der Aktion a in Zustand s unter der optimalen Regelfunktion
γ, α	Lernrate/ Rabattfaktor
ε	Wahrscheinlichkeit eine zufällige Aktion auszuwählen (gilt nur für den ε-greedy Algorithmus)
$g(\cdot)$	Aktivierungsfunktion eines Neuron
$\theta_{i,j}$	gewichtete Verbindung zwischen Neuron i und Neuron j
a_i	Ausgabewert des Neurons i
I	Eingabematrix der diskreten Faltung mit der Größe $m \times n$
K	Filterkernmatrix der diskreten Faltung mit der Größe $m \times n$
$S(i,j)$	Ausgabewert der diskreten Faltung an der Stelle i,j
$L_i(\theta_i)$	Verlustfunktion für die Iteration i mit den Funktionsparametern
$E(\cdot)$	Erwartet Wert
$\triangle_\theta J(\pi_\theta)$	Gradient der Funktionsparameter unter der Regelfunktion π die abhängig von den Funktionsparametern θ ist
$A(s,a)$	Vorteilsfunktion für den Zustand s nach dem Ausführen der Aktion a

Formelverzeichnis

(2.1) Markov Eigenschaft 5

(2.2) Belohnungsfunktion 5

(2.3) stochastische Regelfunktion 6

(2.4) deterministische Regelfunktion 6

(2.5) stochastische Regelfunktion mit Parametern 7

(2.6) deterministische Regelfunktion mit Parametern 7

(2.7) Zustand-Wertfunktion 8

(2.8) optimale Zustand-Wertfunktion 8

(2.9) Aktion-Wertfunktion 8

(2.10) optimale Aktion-Wertfunktion 8

(2.11) optimale Aktion-Wertfunktion aus optimaler Zustand-Wertfunktion 9

(2.12) Aktualisierungsfunktion des Q-Wertes 9

(2.13) Gradientenaktualisierung des REINFORCE Algorithmus 11

(2.14) Gradientenfunktion des REINFORCE Algorithmus 11

(2.15) ε-Greedy Regel 13

(2.16) Ausgabefunktion eines Neurons 15

(2.17) diskrete Faltung 17

(2.18) Verlustfunktion des Deep Q Networks 18

(2.19) Advantage Funktion 20

(2.20) Gradientenaktualisierung des A3C Algorithmus 21

Abkürzungsverzeichnis

A3C ..... Asynchronous Advantage Actor Critic

DL ..... Deep Learning

DRL ..... Deep Reinforcement Learning

DQN ..... Deep Q Network

fNN ..... faltendes Neuronales Netz

MEP ..... Markov Entscheidungsprozess

ML ..... Maschinelles Lernen

NN ..... Neuronales Netz

ReLU ..... Rectifier Linear Unit

RGB ..... Rot-Grün-Blau

RL ..... Reinforcement Learning

Std.abw. ..... Standardabweichung

1. Einleitung

1.1 Motivation

Einer der spannendsten Bereiche der Informatik bildet in den letzten Jahren das Teilgebiet Künstliche Intelligenz. Vor allem in dem Teilbereich Machine Learning und dessen Subgebieten gelingen in den letzten Jahren Fortschritte. Viele Methoden, die früher zu viele Ressourcen benötigt hätten, werden durch die steigende Rechenleistung erneut aufgegriffen.[1] In Anwendungen, wie zum Beispiel der Analyse von Bildern, der Analyse von Sprache, Prognosen in der Medizin, der Steuerung von Robotern[2] oder dem Management von Kunden, eröffnet dies neue Möglichkeiten.

Die Teilbereiche Überwachtes und Unüberwachtes Lernen sind für das Training der Modelle auf große Datenmengen mit ausreichender Qualität und gegebenenfalls auf Expertenwissen angewiesen. Durch die steigende Komplexität der Probleme erhöht sich auch die Anzahl der verschiedenen zu betrachtenden Dimensionen und somit die Datenmenge.[3]

Beim Reinforcement Learning (RL) oder auch Bestärkendem Lernen hingegen, benötigt man diese nicht. Hierbei erlernt das Modell durch iterative Simulation des Problems selbstständig eine Lösung, indem es eine Aktion in einem Umfeld anwendet und das Ergebnis beobachtet. Somit erstellt das Modell die Testdaten selbst.[4]

So konnte die Google-Tochter DeepMind im Jahr 2013 ein Modell entwickeln, dass verschiedene ATARI-Spiele meistern und menschliche Ergebnisse übertreffen konnte, ohne dabei das Modell auf die einzelnen und zum Teil sehr unterschiedlichen Probleme zu optimieren.[5]

Reinforcement Learning ist besonders bei Spielen beliebt, da diese eine überschaubare Anzahl an Zuständen haben, klar definierte Regeln haben und doch sehr komplex sein können. Ein weiteres großes Anwendungsgebiet ist die Robotik. Durch die steigende Anzahl an Robotern in der Wirtschaft wird RL auch in der realen Arbeitswelt an Relevanz gewinnen.

---

[1] Vgl. Döbel, I. u. a. (2018), S. 9.
[2] Vgl. S. Gu u. a. (2017), S. 3389.
[3] Vgl. Döbel, I. u. a. (2018), S. 25.
[4] Vgl. ebd., S. 28.
[5] Vgl. Mnih, V. u. a. (2013), S. 1.

Da es bereits viele unterschiedliche RL Algorithmen gibt, sollen in dieser Arbeit verschiedene Algorithmen aufgezeigt, implementiert und miteinander verglichen werden.

1.2 Zielsetzung

Ziel der Arbeit soll es sein, Reinforcement Learning und die dahinterstehenden Konzepte zu erläutern, die ausgewählten Algorithmen in Python zu implementieren und in Bezug auf das ATARI-Spiel Breakout mithilfe eines Experimentes zu vergleichen und daraus gegebenenfalls eine Empfehlung für ähnliche Probleme, wie beispielsweise andere ATARI Spiele, geben zu können.
Folgende Forschungsfragen sollen dabei versucht werden zu beantworten:

- Welcher RL Algorithmus erreicht die höchste Punktzahl in Breakout?
- Kann man die Algorithmen unter anderen Kriterien vergleichen, wie zum Beispiel der Trainingszeit oder der verwendeten Ressourcen?
- Lassen sich die Ergebnisse auch auf andere Probleme projizieren?

1.3 Vorgehen

Die Forschungsfragen sollen mithilfe der Experimente, Auswertung, der daraus resultierenden Ergebnisse und Statistiken beantwortet werden.
Zuerst soll ein genereller Überblick über Reinforcement Learning und die dahinterstehenden Konzepte gegeben werden. Hierbei wird Reinforcement Learning in das Themengebiet Machine Learning eingeordnet. Anschließend wird die Testumgebung Breakout genau beschrieben und eine Auswahl der Algorithmen getroffen. Danach folgen die einzelnen Tests der Algorithmen. Danach werden die Ergebnisse der Tests miteinander verglichen und evaluiert. Zum Schluss erfolgt eine Limitation der Arbeit und ein Ausblick.

2. Grundlagen

In diesem Kapitel wird der Fokus auf die zentralen Entscheidungsprozesskonzepte und auf die Elemente gelegt, die für die Implementierung und die Experimente notwendig sind.

2.1 Reinforcement Learning

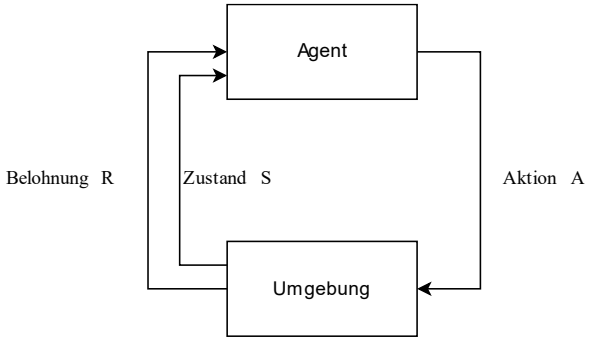

Abbildung 2.1: Interaktion des RL Agenten mit der Umgebung in Anlehnung an Sutton und Barto (2018)[6]

Reinforcement Learning ist ein Lernstil des Maschinellen Lernens (ML).[7] Man behandelt hierbei das Problem eine Vorhersage zu erlernen.[8] Das Standardmodell besteht dabei aus einem Agenten, der versucht ein Verhalten zu erlernen, um ein Ziel zu erreichen, indem er mit einer dynamischen Umgebung interagiert.[9] Die Interaktion wählt der Agent aus einer Menge von möglichen Aktionen über mehrere in sich abgeschlossene Zeitintervalle aus.[10] Bei jeder Interaktion erhält der Agent von der Umgebung eine Belohnung, die der Agent benutzt um die Einschätzung der Aktion zu bewerten und seine Entscheidungen zu verbessern.[11] Die Belohnung kann dabei positiv oder negativ ausfallen.[12] Der Agent versucht dabei zu lernen, die zukünftigen Belohnungen zu maximieren und somit die bestmögliche Aktion für einen gewissen Zustand auszuführen.[13] Die ausgeführte Aktion verändert anschließend die Umgebung, in der der Agent agiert und erzeugt einen neuen Zustand.[14] Der beispielhafte Prozess ist in Abbildung 2.1 abgebildet.

---

[6] Vgl. Sutton, R. S., Barto, A. (2018), S. 54.
[7] Vgl. Döbel, I. u. a. (2018), S. 28.
[8] Vgl. Sutton, R. S. (1988), S. 9.
[9] Vgl. Kaelbling, L. P. u. a. (1996).
[10] Vgl. Watkins, C. J.C.H., Dayan, P. (1992), S. 280.
[11] Vgl. Rummery, G. A., Niranjan, M. (1994), S. 4.
[12] Vgl. Gatti, C. (2015), S. 9.
[13] Vgl. Mnih, V. u. a. (2015), S. 529.
[14] Vgl. Kaelbling, L. P. u. a. (1996), S. 238.

Der Zustand kann beispielsweise eine Position auf einem Spielfeld, verschiedene Sensordaten oder ein Bild sein.

Allgemein kann man zwischen modellfreien Agenten und modellbasierten Agenten unterscheiden. Bei modellbasierten Agenten ist der Agent speziell für ein Problem modelliert. Bei modellfreien Agenten ist das Modell für verschiedene Probleme gedacht.[15]

Die Optimierung des Agenten wird über eine vorgegebene Anzahl an Iteration, welche auch als Experimente oder Episoden bezeichnet werden, ausgeführt.[16] Jede Episode ist die Summe der Schritte zwischen dem ersten Zustand und dem letzten oder auch terminierenden Zustand.[17]

Probleme, die sich sequenziell lösen lassen, werden oft als Markov Entscheidungsprozess definiert.[18] Diese werden im folgenden Kapitel genau definiert.

2.2 Markov Entscheidungsprozess

Reinforcement Learning-Algorithmen basieren auf Markov Entscheidungsprozessen (kurz MEP).[19] Diese wurden von Bellman (1957) vorgestellt.[20] Ein MEP ist ein statistisches Modell, zum Beschreiben von sequenziellen Entscheidungsproblemen und eine Erweiterung der Markov Kette.[21] In RL spiegelt ein MEP das Modell der Umgebung wider, mit der der Agent interagiert. Es wird definiert durch das Tupel $(S, A, P, R(s), p_0, \gamma)$.[22] Dabei gilt:

- S ist eine endliche nicht leere Menge von zuständen s[23]
- A ist eine endliche nicht leere Menge von möglichen Aktionen a[24]
- P ist eine Funktion für die Übergangswahrscheinlichkeit von einem Zustand in den nächsten Zustand[25]

---

[15] Vgl. Degris, T. u. a. (2012), S. 2182.
[16] Vgl. Sewak, M. (2019), S. 2.
[17] Vgl. Wang, Z. u. a. (2015), S. 3.
[18] Vgl. Leemon Baird (1995), S. 30.
[19] Vgl. K. Arulkumaran u. a. (2017), S. 28.
[20] Bellman, R. (1957).
[21] Vgl. Riedmiller, M. (2005), S. 318.
[22] Vgl. Schulman, J. u. a. (2015a), S. 1889.
[23] Vgl. Szepesvári, C. (2010), S. 15.
[24] Vgl. Schulman, J. u. a. (2015a), S. 1889.
[25] Vgl. Watkins, Christopher John Cornish Hellaby (1989).

- $R(s)$ ist eine Belohnungs- oder Verlustfunktion[26]
- p_0 ist die Verteilung des initialen Zustands[27]
- γ ist ein Rabattfaktor im Intervall (0,1), der die Prioritäten von frühen Belohnungen im Vergleich zu späteren Belohnungen steuert[28]

Damit ein Problem als MEP bezeichnet werden kann, muss es die Markov Eigenschaft erfüllen. Bei einem endlichen MEP kann die Markov Eigenschaft formal durch

$$p(s', r|s, a) = Pr\{R_{t+1} = r, S_{t+1} = s'|S_t = s, A_t = a\} \quad (2.1)$$

definiert werden. Pr beschreibt dabei die Wahrscheinlichkeit den folgenden Zustand s' vom Zustand s zu erreichen, wenn man die Aktion a ausführt.[29] Dabei gilt $s \in S$, $s' \in S$, $a \in A$. Der aktuelle Zustand s zum Zeitpunkt t ist dabei vollkommen unabhängig von allen vorherigen Zuständen.[30]

Wenn der MEP einen terminierenden Status T hat, spricht man von einem endlichen MEP, indem gilt $t \in [1, \dots, T]$. Wenn kein terminierender Zustand existiert, bezeichnet man das MEP als unendliches MEP.[31]

Die Belohnungsfunktion $R(s)$ gibt eine Bewertung der zuletzt ausgeführten Aktion a für den Zustand s an den Agenten zurück. Die Belohnung hängt dabei vom aktuellen Zustand s ab und kann positiv oder auch negativ ausfallen.[32] Die Belohnungsfunktion kann formal durch

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k} \quad (2.2)$$

definiert werden.[33] Durch den konstanten Rabattfaktor $0 < \gamma \leq 1$ können zukünftige Belohnungen anders gewichtet werden, um somit die Relevanz von späteren Zuständen für den aktuellen Zustand zu verringern.[34]

---

[26] Vgl. Riedmiller, M. (2005), S. 318.
[27] Vgl. Schulman, J. u. a. (2015a), S. 1889 f.
[28] Vgl. Brendan O'Donoghue u. a. .
[29] Vgl. Sutton, R. S., Barto, A. (2018).
[30] Vgl. Leemon Baird (1995), S. 30.
[31] Vgl. Levine, S. u. a. (2015), S. 5.
[32] Vgl. D. Silver u. a. (2014)., S 388.
[33] Vgl. Li, Y. (2018b), S. 9.
[34] Vgl. Szepesvári, C. (2010), S. 17.

2.3 Funktionen zum Verbessern des Verhaltens

Im folgenden Kapitel wird beschrieben, wie das Verhalten des Agenten beeinflusst und verbessert werden kann und formal spezifiziert. Von einem Agenten wird erwartet, dass er zunehmend höhere Belohnungen pro Episode sammelt und somit aktiv durch die Bestärkung lernt. Jedem Zustand folgt eine Aktion, die zu einem neuen Zustand und dazugehöriger Belohnung führt. Daraus folgt die simple Aufgabe für einen Agenten, die bestmögliche Aktion für jeden einzelnen Zustand zu wählen.

2.3.1 Regelfunktion

Um ein RL Problem zu lösen muss man das Verhalten des Agenten trainieren. Das Verhalten des Agenten wird gesteuert durch eine Regelfunktion, welche auf lange Sicht darauf abzielen soll, die größtmögliche Belohnung pro Episode zu erhalten.[35] Die Regel kann man auch als Strategie verstehen, die die nächste Aktion in einem gewissen Zustand bestimmt.[36]

Die Regel kann entweder stochastisch[37] oder deterministisch[38] sein. Wenn die Regel stochastisch ist, dann ist diese eine Wahrscheinlichkeitsverteilung von Aktionen zu einem bestimmten Zustand.[39] In diesem Fall wird die Regel durch π gekennzeichnet:

$$\pi(a_t|s_t), \qquad a_t \in A \quad s_t \in S \tag{2.3}$$

[40]

Eine deterministische Regel ordnet jedem Zustand eine Aktion zu, die garantiert ausgeführt wird.[41] In diesem Fall wird die Regel formal durch μ dargestellt:

$$a_t = \mu(s_t), \qquad a_t \in A \quad s_t \in S \tag{2.4}$$

[42]

Die Regel kann auch durch zusätzliche Parameter θ ergänzt werden.[43] Diese können beispielsweise Gewichte und Filterkerne eines Neuronalen Netzes sein (siehe Kapitel

---

[35] Vgl. Sutton, R. S., Barto, A. (2018)., S. 75.
[36] Vgl. Sewak, M. (2019), S. 16.
[37] Vgl. Degris, T. u. a. (2012), S. 2177 f.
[38] Vgl. D. Silver u. a. (2014), S. 387.
[39] Vgl. Lillicrap, T. P. u. a. (2015), S. 2.
[40] Vgl. Szepesvári, C. (2010), S. 20.
[41] Vgl. D. Silver u. a. (2014), S. 387.
[42] Vgl. ebd., S. 387.
[43] Vgl. Mnih, V. u. a. (2016), S. 1930.

2.6).[44] In diesem Fall wird die formale Notation um die Parameter erweitert, sodass eine stochastische Regel durch

$$\pi_\theta(a_t|s_t) \tag{2.5}$$

definiert wird und eine deterministische Regel durch

$$a_t = \mu_\theta(s_t) \tag{2.6}$$

definiert wird.[45]

Die beiden Varianten unterscheiden sich darin, dass die stochastische Regel Zustands- und Aktionsraum integriert, wobei die deterministische Regel nur den Zustandsraum einbezieht. Wenn der Aktionsraum viele Dimensionen beinhaltet, wird der Rechenaufwand bei einer stochastischen Regel aufwendiger, da pro Kombination aus Zustand und Aktion Beispiele benötigt werden.[46]

Theoretisch ist die Regel, die die höchste summierte Belohnung für eine bestimmte Umgebung erhält, die optimale Regel π^*.[47] Darüber hinaus kann eine Regel auch als optimal bezeichnet werden, wenn diese die Eigenschaft besitzt, dass egal welcher erste Zustand und welche erste Aktion ausgeführt wurden, die darauffolgenden Aktionen die größtmögliche Belohnung erhalten.[48] Dies wird auch als Bellmanns „Prinzip der Optimalität“[49] bezeichnet.

2.3.2 Wertfunktion

Neben der Regelfunktion gibt es noch die Wertfunktionen, um das Verhalten des Agenten zu verbessern. Die Wertfunktion ermittelt, wie gut der aktuelle Zustand ist, indem sich der Agent befindet oder wie gut es ist, eine gewählte Aktion in einem gegebenen Zustand auszuführen.[50] Wie gut ein Zustand oder eine Kombination aus Zustand und Aktion ist, wird definiert durch die summierten, erwarteten, rabattierten, zukünftigen Belohnungen (Formel (2.2)).[51]

---

[44] Vgl. Levine, S. u. a. (2015), S. 5.
[45] Vgl. Wu, Y. u. a. (2017).
[46] Vgl. D. Silver u. a. (2014), S. 387.
[47] Vgl. Gatti, C. (2015), S. 10.
[48] Vgl. Bellman, R., Kalaba, R. E. (1965), S. 73.
[49] Vgl. ebd., S. 73.
[50] Vgl. Sutton, R. S., Barto, A. (2018), S. 70.
[51] Vgl. Li, Y. (2018b), S. 9.

Da die zukünftigen Belohnungen von den ausgeführten Aktionen abhängig sind, ist die Zustand-Wertfunktion von der Regel beeinflusst. Der Wert eines Zustands s nach der Regel π wird deklariert durch $v_\pi(s)$.[52] Für ein MEP wird v_π formal durch

$$v_\pi(s) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \middle| s_0 = s\right], \qquad s \in S \tag{2.7}$$

definiert, wobei $\mathbb{E}_\pi[\cdot]$ der erwartete Wert einer zufälligen Variablen für einen Agenten, welcher der Regel π folgt, kennzeichnet.[53] Die sogenannte Zustand-Wertfunktion hängt nur vom Zustand s ab.[54] Die optimale Zustand-Wertfunktion wird deklariert durch $v^*(s)$ und definiert durch

$$v^*(s) = \max_\pi v_\pi(s), \qquad s \in S \tag{2.8}$$

für alle $s \in S$.[55] v^* gibt dabei den maximalen Zustandswert an, der von einer Regel für Zustand s erreicht werden kann.[56]

Ähnlich zu der Zustand-Wertfunktion kann man die Aktion-Wertfunktion für das Ausführen der Aktion a im Zustand s nach der Regel π als $Q_\pi(s,a)$ für die erwartete Belohnung deklarieren.[57] Formal wird die Aktion-Wertfunktion $Q_\pi(s,a)$ durch

$$Q_\pi(s,a) = \mathbb{E}_\pi\left[\sum_{t=0}^{\infty} \gamma^t R_{t+1} \middle| s_0 = s, a_0 = a\right], \qquad s \in S, a \in A \tag{2.9}$$

definiert.[58]

Die optimale Aktion-Wertfunktion wird durch $Q^*(s,a)$ deklariert, für die gilt

$$Q^*(s,a) = \max_\pi Q_\pi(s,a), \qquad s \in S, a \in A \tag{2.10}$$

für alle (s,a).[59] Dabei gibt $Q^*(s,a)$ das Maximum eines Aktionswertes an, der von einer Regel für den Zustand s und der Aktion a erreicht werden kann.[60]

---

[52] Vgl. Sutton, R. S., Barto, A. (2018), S. 70.
[53] Vgl. Szepesvári, C. (2010), S. 21.
[54] Vgl. I. Grondman u. a. (2012), S. 1292.
[55] Vgl. Sutton, R. S., Barto, A. (2018), S. 75.
[56] Vgl. Li, Y. (2018a), S. 14.
[57] Vgl. Sutton, R. S., Barto, A. (2018), S. 70.
[58] Vgl. Szepesvári, C. (2010), S. 21.
[59] Vgl. Brendan O'Donoghue u. a. , S. 3.
[60] Vgl. Li, Y. (2018a), S. 14.

Die optimale Aktion- und Zustand-Wertfunktion lassen sich in Beziehung zueinander setzen durch die Gleichung: [61]

$$Q^*(s,a) = \mathbb{E}[R_{t+1}\gamma v^*(s_{t+1})|s_0 = s, a_0 = a] \quad (2.11)$$

Sobald man die optimale Zustand Wertfunktion v^* für alle s hat, kann man eine optimale Regel definieren, indem man zu jedem Zustand s die Aktion mit dem höchsten Wert für den nächsten Zustand auswählt.[62] Wenn man die optimale Aktion-Wertfunktion $Q^*(s,a)$ hat, muss man zu dem Zustand s die Aktion nehmen, die Q^* maximiert, sodass man im Gegensatz zu v^*, den nächsten Status s_{t+1} nicht mit in Betracht ziehen muss.[63]

2.3.3 Wertfunktions-Algorithmus: Q-Learning

Der von Watkins (1989) [64] vorgestellte Q-Learning Algorithmus ist einer der bekanntesten RL Algorithmen. Mithilfe des Q-Learning Algorithmus wird versucht einen realen Wert Q von Zuständen und Aktionen zu ermitteln. Dabei ist $Q(s,a)$ die erwartete, rabattierte, zukünftige Belohnung für das Ausführen von Aktion a im Zustand s und folgend optimal handelnd.[65] Diesen Wert nennt man auch Q-Wert. Die Q-Funktion ist dabei die Aktions-Wertfunktion aus Kapitel 2.3.2. Die Aktualisierung von $Q(s,a)$ wird definiert durch: [66]

$$Q(s,a) \leftarrow Q(s,a) + \alpha\left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a_{t+1}) - Q(s,a)\right], s \in S, a \in A \quad (2.12)$$

wobei α ein Lernfaktor ($0 < \alpha \leq 1$)ist, der bestimmt, mit welcher Rate der alte Q-Wert mit dem Delta zwischen altem und neuem Q-Wert überschrieben wird. γ ist ein Rabattfaktor ($0 < \gamma \leq 1$), um zukünftige Q-Werte geringer gewichten zu können.[67] Die Aktions-Wertfunktion Q wird schrittweise aktualisiert und nähert sich so der optimalen Aktions-Wertfunktion Q^* an.[68] Die einzige Voraussetzung, damit Q sich Q^* annähert ist, dass die Q-Werte aller Zustands Aktions-Paare aktualisiert werden.[69] Ein großer Vorteil von Q-Learning ist die Einfachheit, da man nicht auf eine Regel angewiesen ist, sondern

---

[61] Vgl. Szepesvári, C. (2010), S. 21.
[62] Vgl. Sutton, R. S., Barto, A. (2018), S. 77.
[63] Vgl. ebd., S. 77.
[64] Watkins, Christopher John Cornish Hellaby (1989).
[65] Vgl. R. S. Sutton u. a. (1992), S. 20.
[66] Vgl. Sutton, R. S., Barto, A. (2018), S. 157.
[67] Vgl. Watkins, C. J.C.H., Dayan, P. (1992), S. 281.
[68] Vgl. Sutton, R. S., Barto, A. (2018), S. 157.
[69] Vgl. Melo, F. S. (2001), S. 3.

die Aktionen mit dem höchsten Q-Wert auswählt.[70] Dies nennt man auch eine gierige Regel (Kapitel 2.4).

Der Q-Learning Algorithmus ist beispielhaft in Algorithmus 2.1 als Pseudocode dargestellt und funktioniert wie folgt. Zuerst werden für alle möglichen Kombinationen von Zuständen und Aktionen $S \times A$ die Q-Werte zufällig initialisiert und der Trainingsprozess beginnt. In jeder neuen Episode wird zuerst der Startzustand initialisiert. Für jeden Schritt in einer Episode wählt der Agent eine Aktion für den aktuellen Zustand anhand seiner Regel (zum Beispiel eine ε-greedy Regel), führt diese Aktion aus und beobachtet die erhaltene Belohnung und den neuen Zustand. Anschließend wird der Q-Wert nach der Formel (2.12) aktualisiert. Zuletzt wird der Zustand für den nächsten Schritt mit dem neuen beobachteten Zustand aktualisiert.

Algorithmus 2.1: Q-Learning (Pseudocode)

```
1 INITIALISIERE Q(S,A) willkürlich
2 WIEDERHOLE für jede Episode:
3     INITIALISIERE S
4     WIEDERHOLE für jeden Schritt in Episode
5           WÄHLE A von S mithilfe von Q abgeleiteter Regel (ε-greedy)
6           FÜHRE Aktion A aus, beobachte R und S‘
7           Q(S,A) ← Q(S,A) + α [R + γ max_a Q(S‘,a) - Q(S,A)]
8           S ← S‘
9     BIS S ist Terminalzustand
```

Algorithmus 2.1: Q-Learning Pseudocode in Anlehnung an Sutton und Barto (2018)[71]

2.3.4 Regelfunktions-Algorithmus: REINFORCE

Der REINFORCE Algorithmus ist ein Beispiel für eine Regel Gradienten Funktion. Das Ziel dieser Algorithmen ist, eine stochastische Regelfunktion $\pi(a|s)$ zu trainieren. Wie in Kapitel Regelfunktion beschrieben, bildet die Regel eine Strategie für die Auswahl von Aktionen in einem MEP. Im Gegensatz zu Wertfunktions-Algorithmen benutzen Regel

[70] Vgl. Szepesvári, C. (2010), S. 62.
[71] Vgl. Sutton, R. S., Barto, A. (2018), S. 158.

Gradienten Funktionen aber nicht unbedingt eine Wertfunktion.[72] Die Regel ist dabei abhängig von einer Sammlung von Parametern θ und die Regel wird somit durch $\pi_\theta(a|s)$ definiert.[73] Die Basisidee bei der Suche der Regel ist, die Parameter θ so anzupassen, dass die Aktion mit der höchsten erwarteten Belohnung ausgewählt wird.[74] Wie in Q-Learning müssen die Parameter der Regelfunktion regelmäßig aktualisiert werden. Die Aktualisierung der Parameter findet mithilfe eines Gradienten $\triangle_\theta J(\pi_\theta)$ statt, sodass sich für die Anpassung der Parameter folgende Gleichung ergibt:

$$\theta_{t+1} = \theta_t + \gamma \triangle_\theta J(\pi_\theta), \tag{2.13}$$

wobei γ ein Lernfaktor ($0 < \gamma \leq 1$) ist und θ die Parameter für den Zeitpunkt t beziehungsweise $t+1$ sind.[75] Der Regel Gradient wird durch jede durchlaufene Episoden aktualisiert, die mithilfe der entsprechenden Regel simuliert werden. Aus diesen Episoden werden Daten extrahiert, um den Gradienten der Regel zu berechnen.[76] Dieses Vorgehen wurde von Willimas 1992 [77] als REINFORCE Algorithmus beschrieben. Der Regelgradient ist hierbei durch die Gleichung

$$\triangle_\theta J(\pi_\theta) = y^t R_t \triangle_\theta log\pi_\theta(a_t|s_t), \tag{2.14}$$

definiert, wobei y^t ein Rabattfaktor ist und R_t die Belohnung für die gesamte Episode vom Zustand s aus. π_θ ist die parametrisierte Regel des Agenten zum Zustand s und der Aktion a jeweils für den Zeitpunkt t.[78] In Algorithmus 2.2 ist ein Pseudocode für einen allgemeinen REINFORCE Algorithmus gegeben. Dieser funktioniert wie folgt. Zuerst werden die Regel Parameter zufällig initialisiert. Anschließend wird bis zum Ende des Trainings des Agenten eine Episode durchgeführt, indem Aktionen durch die aktuelle Regel des Agenten ausgewählt werden. Nachdem die Episode beendet wurde, wird für jeden Schritt in der Episode die gesamte Belohnung berechnet und die Parameter der Regel mithilfe der Formel (2.14) aktualisiert.

---

[72] Vgl. ebd., S. 10.
[73] Vgl. D. Silver u. a. (2014), S. 388.
[74] Vgl. ebd., S. 388.
[75] Vgl. I. Grondman u. a. (2012), S. 1294.
[76] Vgl. Sewak, M. (2019), S. 134.
[77] Williams, R. J. (1992).
[78] Vgl. Sewak, M. (2019), S. 137.

Algorithmus 2.2: REINFORCE (Pseudocode)

```
1 INITIALISIERE Parameter θ der Regel willkürlich
2 WIEDERHOLE
3     ERSTELLE eine Episode nach Regel π_θ(s_1,a_1, s_2,a_2,…,s_t,a_t)
4     WIEDERHOLE für jeden Schritt t= 0,…,T in Episode:
5           BERECHNE gesamte Belohnung R_T für Episode
6           θ ← θ+αy^t R_t Δ_θ log π_θ(a_t|s_t)
7 BIS Ende des Trainings
```

Algorithmus 2.2: REINFORCE Pseudocode in Anlehnung an Sewak(2019)[79]

2.4 Erkundung oder Ausnutzung

Das Dilemma zwischen Erkundung und Ausbeutung des gesammelten Wissens ist ein bekanntes Problem in RL und in Künstlicher Intelligenz allgemein aber auch in anderen Domänen.[80] So muss man sich beispielsweise auch bei der Wahl eines Restaurants entscheiden, ob man in sein Lieblingsrestaurant geht oder ein neues ausprobiert.

Dabei stellt sich die Frage, ob man bekannte Informationen ausnutzen sollte, wie zum Beispiel ob man einer bekannten Regel mit einer hohen Belohnung folgen sollte, oder ob man neue Zustände erkunden sollte, um eine bessere Regel auf lange Sicht zu finden[81]

In RL gibt es das besondere Problem, dass ein Agent bekannte Aktionen, die eine hohe Belohnung garantieren, bevorzugen muss, um eine hohe Gesamtbelohnung zu erreichen. Um diese Aktionen zu finden, muss der Agent allerdings die Aktionen ausprobieren und die Umgebung erkunden, damit sich der Entscheidungsprozess für zukünftige Aktionen verbessert.[82]

Eine einfache Methode, um einen Kompromiss für den Agenten zwischen der Erkundung und dem Ausbeuten zu schaffen, ist der ε-greedy Algorithmus.[83] Er bildet eine Strategie, um eine Aktion auszuwählen. Dafür sei eine Aktion-Wertfunktion $Q(s,a)$ gegeben,

[79] Vgl. ebd., S. 136.
[80] Vgl. Azoulay-Schwartz, R. u. a. (2004), S. 3.
[81] Vgl. Berry, D. A., Fristedt, B. (1985), S. 2.
[82] Vgl. Sutton, R. S., Barto, A. (2018), S. 3.
[83] Vgl. Li, Y. (2018a), S. 14.

wobei die beste Aktion a mit einer Wahrscheinlichkeit von $(1-\varepsilon), \varepsilon \in [0,1]$ ausgewählt wird. Die beste Aktion ist die Aktion, bei der die Aktion-Wertfunktion für einen gegeben Zustand s am höchsten ist.[84] Mit einer Wahrscheinlichkeit von ε wählt der Agent, anstatt der besten möglichen Aktion, eine zufällige mögliche Aktion $a \in A(s)$ aus.[85] Anders ausgedrückt erkundet der Agent die Umgebung mit der Wahrscheinlichkeit ε und nutzt die aktuelle Aktion-Wertfunktion mit der Wahrscheinlichkeit $1-\varepsilon$ aus.[86] Wenn ε gleich Null ist, dann wird der Agent immer die bestmögliche Aktion basierend auf $Q(s,a)$ auswählen. Falls ε gleich eins ist, wird der Agent die Umgebung erkunden, ohne die Aktion-Wertfunktion zu berücksichtigen.[87] Dies wird formal durch

$$\pi(s) \begin{cases} wähle zufällige Aktion\ a \in A(s), wenn\ zufällige\ Zahl[0,1] < \varepsilon \\ \underset{a}{\operatorname{argmax}}\, Q(s,a)\,, ansonsten \end{cases} \tag{2.15}$$

definiert, wobei $a \in A(s)$, $s \in S$ gilt. Argmax steht dabei für den Wert von a, bei dem der Wert von $Q(s,a)$ maximiert wird.[88]

Eine andere Strategie ist ε über die Zeit zu verringern. Dies nennt man auch einen verringernden ε-greedy Algorithmus. Hierbei startet man beispielsweise mit einem hohen ε, welches sich mit jedem Zeitschritt verringert. Somit erkundet der Agent gerade zum Anfang eines Trainingsprozesses die ausgewählte Umgebung. Je länger die Trainingsphase dauert, umso häufiger wählt der Agent Aktionen basierend auf der Aktion-Wertfunktion aus.[89]

2.5 Deep Reinforcement Learning

Das Anwenden von Reinforcement Learning Algorithmen für Entscheidungsfindungsprobleme bringt einige Schwierigkeiten mit sich. Vor allem in unendlichen MEPs. Die zuvor genannten Beispiele für Algorithmen sind tabellarische Lösungsansätze, in denen jedes Paar aus Zustand und Aktion eine Dimension in einer Tabelle ausmachen.[90]

---

[84] Vgl. Sutton, R. S., Barto, A. (2018), S. 34 f.
[85] Vgl. Caelen, O., Bontempi, G. (2008), S. 58.
[86] Vgl. Li, Y. (2018a), S. 14.
[87] Vgl. Caelen, O., Bontempi, G. (2008), S. 58.
[88] Vgl. Tokic, M. (2010), S. 205.
[89] Vgl. ebd., S. 206.
[90] Vgl. Sharma, S. u. a. (2017), S. 1.

Informationen zu den Zustands-Aktionswerten werden während der Laufzeit gespeichert und aktualisiert. Dadurch können diese Methoden bei komplexeren Problemen nicht praktikabel sein, durch Grenzen im Speicher oder in der Zeit.[91] Vor allem bei hochdimensionalen Problemen, wie zum Beispiel eine Bild oder ein Film, bei der eine Beobachtung oder ein Zustand mehrere Tausend oder mehr Dimensionen haben kann, kann dies zu Problemen führen.[92] Das Problem wird auch als „Fluch der Dimensionalität“[93] beschrieben.

Um diesem Problem entgegenzuwirken kann man, anstelle von Tabellen, andere Deep Learning (DL) Konzepte verwenden. Eines der bekanntesten Konzepte ist das tiefe Neuronale Netz (NN). Mithilfe von NN lassen sich, hochdimensionale Beobachtungen in eine komprimierte Repräsentationen ableiten.[94] Die Kombination aus DL und RL Methoden bezeichnet man auch als Deep Reinforcement Learning (DRL).

Im folgenden Kapitel werden Grundlagen von NN aufgezeigt und zwei Algorithmen beschrieben, die NN für die Funktionsapproximation für die Wertfunktion und/ oder die Regelfunktion verwenden.

2.6 Neuronale Netze

Neuronale Netze bestehen aus künstlichen Neuronen. Diese werden über gewichtete Verbindungen miteinander verknüpft.[95] Das Kapitel beschreibt den Aufbau eines Neurons und beschreibt verschiedene Arten von Neuronalen Netzen und deren Konzepte.

2.6.1 Neuron

Ein Neuron ist die kleinste Einheit in einem NN. Jedes Neuron erhält eine oder mehrere Eingabewerte, welche mit dazugehörigen Gewichten gewertet werden. Aus den Eingabewerten wird ein Ausgabewert berechnet. Diesen berechnet eine Aktivierungsfunktion.[96]

Die Berechnung des Ausgabewertes eines Neurons wird formal definiert durch

---

[91] Vgl. Li, Y. (2018a), S. 18 f.
[92] Vgl. Donoho, D. L., others (2000), S. 1.
[93] Vgl. ebd., S. 1.
[94] Vgl. K. Arulkumaran u. a. (2017), S. 26 f.
[95] Vgl. Hopfield, J. J. (1982), S. 2554 f.
[96] Vgl. Kim, P. (2017), S. 21 f.

$$a_j = g\left(\sum_{i=0}^{n} \theta_{i,j} a_i\right), \tag{2.16}$$

wobei $a_j$die Ausgabe des betrachteten Neurons ist. a_i ist die Ausgabe des Neurons an der Stelle i aus der vorherigen Schicht. $g(\cdot)$ ist die Aktivierungsfunktion und $\theta_{i,j}$ ist die gewichtete Verbindung zwischen Neuron i und j. n ist die Anzahl an Neuronen aus der vorherigen Schicht [97] Für jedes Neuron wird also die gewichtete Summe aller Eingabesignale berechnet. Die Ausgabe des Neurons entspricht dem Ergebnis der Aktivierungsfunktion der gewichteten Summe.[98]

Die Aktivierungsfunktion entscheidet, ob ein Neuron aktiviert wird oder nicht. Dafür gibt es einen Schwellenwert, von der die Aktivierung abhängt.[99]

Es gibt verschiedene Aktivierungsfunktionen. Die Rectifier Linear Unit oder kurz ReLU eignet sich besonders für das Trainieren von tiefen NN.[100] Diese kann formal durch $g(x) = max(0, x)$ dargestellt werden. Alle negativen Eingabewerte werden mit Null zusammengefasst. Für alle positiven Eingabewerte wird ein Ausgabewert proportional zur Eingabe ausgegeben. Die Berechnung während des Trainings ist dabei einfach und Ressourcen schonend.[101]

2.6.2 Tiefes Neuronales Netz

Ein NN erhält man, indem man viele Neuronen miteinander verbindet. Das NN wird dabei in mehrere Schichten unterteilt. Es gibt eine Eingabe- und eine Ausgabeschicht. Dazwischen können beliebig viele unsichtbare Schichten existieren.[102] Bei NN mit zwei oder mehr unsichtbaren Schichten, spricht man von einem tiefen NN.[103] Jede Schicht kann eine unterschiedliche Anzahl an Neuronen haben und unterschiedlich stark miteinander verknüpft sein.[104] In Abbildung 2.2 ist ein Modell eines tiefen NN abgebildet.

---

[97] Vgl. Hansen, L. K., Salamon, P. (1990), S. 994.
[98] Vgl. Kim, P. (2017), S. 22.
[99] Vgl. Hopfield, J. J. (1982), S. 2555.
[100] Vgl. Glorot, X. u. a. (2011), S. 315.
[101] Vgl. ebd., S. 318.
[102] Vgl. Specht, D. F., others (1991), S. 572 f.
[103] Vgl. Kim, P. (2017), S. 23.
[104] Vgl. Goodfellow, I. u. a. (2016), S. 164 f.

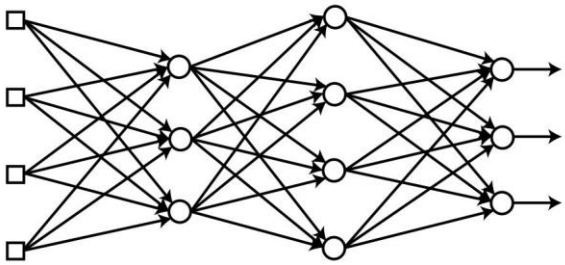

Abbildung 2.2: Tiefes Neuronales Netz von Kim (Hg.) 2017, S. 24[105]

Hierbei ist zu beachten, dass die Verbindungen zwischen den Neuronen gerichtet sind und dass keine rückläufigen Verbindungen existieren.

Allgemein verwendet man ein Neuronales Netz, um eine Funktion f zu bestimmen, wie zum Beispiel eine Funktion $y = f(x, \theta)$, die eine Eingabe x zu einer bestimmten Kategorie y zuordnet, indem die Parameter θ erlernt werden.[106] Im Falle von RL kann y beispielsweise eine Aktion sein, die der Agent ausführen soll.

Um das NN zu trainieren und sich der Funktion anzunähern, kann man verschiedene Gradientenverfahren verwenden. Ein Gradientenverfahren ist RMSProp[107]. Dabei wird versucht die Differenz zwischen einer gewünschten Ausgabe und dem tatsächlichen Ausgabewert zu minimieren. Eine Verlustfunktion berechnet dabei diese Differenz. Die gewichteten Verbindungen der unsichtbaren Schichten werden auf Basis des Verlustwertes durch einen Gradienten des Lernverfahrens aktualisiert.[108]

2.6.3 Faltende Neuronale Netze

Ein faltendes Neuronales Netz (fNN) ist eine besondere Form von NN, welche sich besonders dazu eignen hochdimensionale Daten zu verarbeiten, die ein Rastermuster aufweisen. Dies können beispielsweise Bilder sein.[109] Dafür werden zwei neue Arten von Schichten eingeführt, die das NN erweitern. Zum einen die Faltungsschicht und zum anderen die Pooling-Schicht.[110]

Die Faltungsschicht faltet die Eingabematrix mithilfe eines Filterkerns und berechnet so eine Ausgabewertmatrix. Die diskrete Faltung kann formal durch

---

[105] Kim, P. (2017), S. 24.
[106] Vgl. Goodfellow, I. u. a. (2016), S. 164.
[107] Vgl. Hinton, G. (o.J.)., S. 29.
[108] Vgl. Rumelhart, D. E. u. a. (1986), S. 533.
[109] Vgl. Goodfellow, I. u. a. (2016), S. 326.
[110] Vgl. Kim, P. (2017), S. 124.

$$S(i,j) = \sum_m \sum_n I_{(i+m,j+n)} K_{(m,n)} \tag{2.17}$$

definiert werden. Dabei ist I eine $n \times m$ Eingabematrix, K ist ein $n \times m$ Filterkern und $S(i,j)$ ist der Wert der Ausgabematrix an der Stelle i,j. Jede Faltungsschicht kann mehrere Filterkerne besitzen.[111]

Durch die diskrete Faltung wird mithilfe des Filterkerns eine gewichtete Summe aus Werten einer Matrix in einem bestimmten Bereich gebildet. Der Filterkern legt dabei die Gewichte fest. Diese werden durch den Trainingsprozess kontinuierlich angepasst, ähnlich dem Anpassen der Verbindungsgewichte in einem NN (siehe Kapitel 2.6.2).[112] Simpler ausgedrückt, wird der Filterkern auf jeden Wert in der Eingabematrix gelegt, beziehungsweise auf nur jeden n-ten Wert. Die Werte der Matrix, die sich mit dem Filterkern überschneiden, werden mit den entsprechenden Werten aus dem Filterkern multipliziert. Die Produkte der einzelnen Werte werden summiert und bilden den neuen Wert der Ausgabematrix an der Stelle der betrachteten Eingabematrix. In Abbildung 2.3 ist eine beispielhafte Faltungsoperation abgebildet.

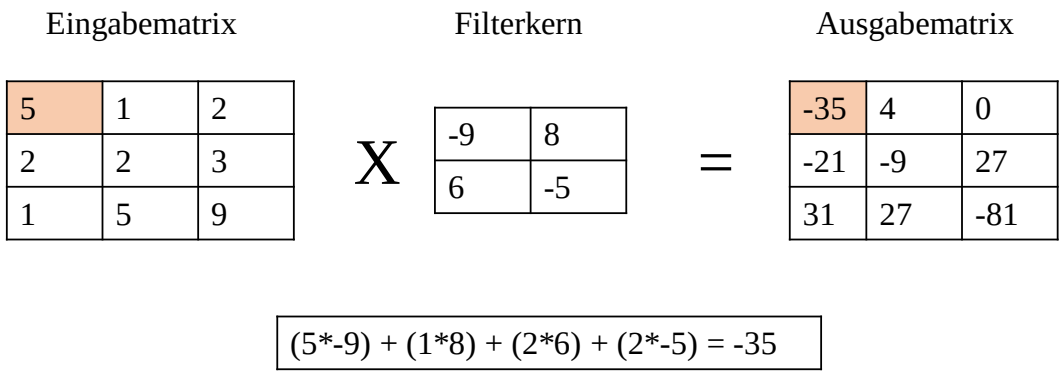

Abbildung 2.3: Beispielhafte Faltungsoperation in Anlehnung an Kim, P.(2017), S. 126 f.[113]

Die Pooling-Schicht ersetzt die Ausgabe an einer bestimmten Position im NN. Dabei verkleinert das Pooling die Ausgabe, indem es Bereiche in der Eingabematrix mit gewissen Methoden zusammenfasst. Dies ist vor allem hilfreich, wenn man herausfinden möchte, ob eine bestimmte Eigenschaft vorhanden ist, aber man nicht wissen muss wo

[111] Vgl. Goodfellow, I. u. a. (2016), S. 328.
[112] Vgl. Kim, P. (2017), S. 125 f.
[113] Vgl. ebd., S. 125 f.

genau.[114] Es gibt verschiedene Methoden für das Pooling. Eine ist beispielsweise das Max-Pooling, wobei aus einem betrachteten benachbarten Bereich einer Eingabematrix der Maximale Wert an die Ausgabematrix weitergegeben wird.[115] Die Pooling-Schicht ist optional und kann durch die Schrittweite des Filterkerns ersetzt werden.

Ein fNN lässt sich in zwei Komponenten aufteilen. Zum Ersten der Eigenschaftendetektor, bestehend aus gegebenenfalls mehreren abwechselnden Faltungs- und Pooling-Schichten. Dies führt zu einer abstrakten Repräsentation der Eingabe mit geringeren Dimensionen. Darauf folgt die zweite Komponente. Diese ist ein Klassifikator, bestehend aus einem (tiefen) NN, das die Eingabe beispielsweise einer auszuführenden Aktion zuordnet (siehe Kapitel 2.6.2). Das Training des Netzes kann, wie beim NN, über ein Gradientenverfahren erfolgen.[116]

2.7 Deep Q Network

Ein Algorithmus, um Deep Learning Methoden mit Reinforcement Learning zu verbinden, ist das Deep Q Network (DQN), welches von Minh u. a. (2013)[117] vorgestellt wurde. Der Algorithmus verbindet eine Variante von Q-Learning mit einem fNN. Die Eingabewerte für das fNN stammen aus einem Bild des Zustandes eines Spiels. Der Ausgabewert ist eine Aktion-Wertefunktion, die zukünftige Belohnungen prognostiziert.[118]

Im Gegensatz zum normalen Q-Learning wird versucht die Parameter des fNN anzupassen und keine Tabelle für jede Kombination aus Zustand und Aktion aufzubauen. Das fNN wird durch ein Gradientenverfahren trainiert. Die dafür benötigte Verlustfunktion versucht man dadurch zu minimieren. Die Verlustfunktion für ein DQN wird durch $L_i(\theta_i)$ definiert und formal durch

$$L_i(\theta_i) = \mathbb{E}_{s_{t+1},a_{t+1}}[(y_i - Q(s_i, a_i; \theta_i))^2] \tag{2.18}$$

beschrieben. Dabei ist $y_i = r_{t+1} + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}; \theta_{i-1})$ der Zielwert und $Q(s_i, a_i; \theta_i)$ ist der aktuelle Wert für das Zustands-Aktionspaar für die Iteration i. Die

[114] Vgl. Goodfellow, I. u. a. (2016), S. 336.
[115] Vgl. Kim, P. (2017), S. 130 f.
[116] Vgl. Koutník, J. u. a. (2014), S. 542.
[117] Mnih, V. u. a. (2013).
[118] Vgl. ebd., S. 1.

Parameter für das Ziel θ_{i-1} sind für die Aktualisierung der Verlustfunktion fixiert. Dies geschieht durch ein zweites fNN, welches auch als Zielnetzwerk bezeichnet wird.[119] Der Pseudocode für den DQN Algorithmus ist in Algorithmus 2.3 zu sehen.
Eine zusätzliche Änderung zum klassischen Q-Learning stabilisiert das Training des Agenten. Dies geschieht durch Erfahrungswiederholung von Lin (1992)[120].

Algorithmus 2.3: Deep Q Network (Pseudocode)

---

1 INITIALISIERE Erfahrungsspeicher D für Kapazität N
2 INITIALISIERE Aktionswertfunktion Q mit zufälligen Parametern
3 FÜR Episode = 1 bis M WIEDERHOLE
4 INITIALISIERE Startzustand s_1
5 FÜR Schritt t=1 bis Terminalzustand T WIEDERHOLE
6 WÄHLE Aktion a mit Wahrscheinlichkeit ε zufällig aus
7 ANDERNFALLS wähle a = $\max_a Q^*(s_t, a; \theta)$ aus
8 FÜHRE Aktion a in Umgebung aus und beobachte Belohnung r_t
UND neuen zustand s_{t+1}
9 SPEICHERE Übergang (s_t, a_t, r_t, s_{t+1}) in Erfahrungsspeicher D
10 HOLE Sammlung von Übergangsproben (s_j, a_j, r_j, s_{j+1}) aus D
11 SETZE

$$y_i \begin{cases} r_j, & \text{wenn t=Terminalzustand} \\ r_j + \gamma \max_{a_{j+1}} Q(s_{j+1}, a_{j+1}; \theta), & \text{wenn t nicht Terminalzustand} \end{cases}$$

12 FÜHRE Aktualisierung der Parameter aus

Algorithmus 2.3: DQN Pseudocode nach Minh (2013)[121]

Dabei werden Erfahrungenstupel $(s_t, a_t, s_{t+1}, a_{t+1})$, die der Agent während des Trainings gesammelt hat, in einem Erfahrungsspeicher gesammelt.[122] Beim Aktualisieren der Parameter des fNN wird anschließend nicht das aktuelle Ergebnis des aktuellen Zustandes verwendet. Anstelle wird eine zufällige Sammlung an Erfahrungen aus dem Erfahrungsspeicher entnommen und für das Parameterupdate verwendet. Dies hat zum

---

[119] Vgl. Mnih, V. u. a. (2013), S. 3.
[120] Vgl. L. Lin (1992).
[121] Vgl. Mnih, V. u. a. (2013), S. 5.
[122] Vgl. L. Lin (1992), S. 29.

einen den Vorteil, dass jede Erfahrung gegebenenfalls mehrfach für Aktualisierungen der Parameter verwendet wird. Zweitens sind aufeinanderfolgende Zustände stark von seinen Vorgängerzuständen abhängig. Das zufällige Auswählen von Proben bricht diese Kopplung.[123] Zum dritten kann die Erfahrungswiederholung helfen, sodass die Parameter nicht zu einem schlechten lokalen Minimum neigen.[124]

2.8 Asynchronous Advantage Actor Critic

Ein weiterer Algorithmus für DRL ist der Asynchronous Advantage Actor Critic (A3C) Algorithmus, welcher 2016 von Minh[125] vorgestellt wurde. Der Algorithmus verwendet im Gegensatz zum DQN andere Konzepte. Man trainiert einen Agenten mit verschiedenen Arbeitern parallel (asynchron) zueinander in verschiedenen Umgebungen und erhält so zu jedem Zeitpunkt eine Auswahl von Zuständen.[126]
Des Weiteren wird die Actor-Critic Methode verwendet. Bei dieser Methode kombiniert man das Training einer Regelfunktion, die die Regel $\pi_\theta(a_t|s_t)$ bereitstellt, und einer Zustand-Wertfunktion, die $v_\theta(s_t)$ bereitstellt. Die Regelfunktion nennt man in diesem Fall Actor, da sie die Aktionen auswählt. Die Zustand-Wertfunktion bezeichnet man als Critic, da sie die Aktionen der Regelfunktion bewertet.[127] Dabei besitzt jeder Arbeiter seine eigenen Parameter für die Regelfunktion und für die Wertfunktion.[128]
Als weiteres Konzept benutzt man die Vorteilsfunktion $A_{\theta,\theta_v}(s_t, a_t)$ (Advantage), welche formal durch

$$A_{\theta,\theta_v}(s_t, a_t) = \sum_{i=0}^{k-1} \gamma^i r_{t+1} + \gamma^k v_\theta(s_{t+k}) - v_\theta(s_t) \tag{2.19}$$

definiert wird. Dabei ist $\gamma^i r_{t+1}$ die rabattierte Belohnung zum Zeitpunkt t, $\gamma^k v_\theta(s_{t+k})$ das rabattierte Ergebnis der Zustand-Wertfunktion für den Zustand s_{t+k} und $v_\theta(s_t)$ ist das Ergebnis der Wertfunktion für den Zustand s_t. Dabei ist k limitiert.[129] Dies wird auch als angenommene Vorteilsfunktion bezeichnet, da der tatsächliche Aktions-Wert nicht

[123] Vgl. Mnih, V. u. a. (2013), S. 4 f.
[124] Vgl. L. Lin (1992), S. 29.
[125] Mnih, V. u. a. (2016).
[126] Vgl. ebd., S. 1928.
[127] Vgl. Sutton, R. S., Barto, A. (2018), S. 257.
[128] Vgl. Mnih, V. u. a. (2016), S. 1930 f.
[129] Vgl. ebd., S. 1931.

verfügbar ist und durch die Zustands-Wertfunktion nur angenommen wird.[130] Somit lässt sich ermitteln, welchen Mehrwert eine Aktion für den gegebenen Status hat.[131]
Die Parameter der Zustand-Wertfunktion und Regelfunktion werden nach jeder Episode wie im REINFORCE Algorithmus aktualisiert. Auch hier wird ein Gradient benutzt, der durch

$$\triangle_{\theta'} \, log\pi_{\theta'}(a_t|s_t)A_{\theta,\theta_v}(s_t,a_t) \tag{2.20}$$

definiert wird.[132] Die Parameter des Actors und des Crititcs werden in der Praxis teilweise geteilt und sind nicht komplett separat.[133] In Algorithmus 2.4 ist der Pseudocode für A3C aufgezeigt. In Teilen ähnelt er stark dem REINFORCE Algorithmus (siehe Algorithmus 2.2).

[130] Vgl. Schulman, J. u. a. (2015b), S. 4 f.
[131] Vgl. Wu, Y. u. a. (2017), S. 3.
[132] Vgl. Mnih, V. u. a. (2016), S. 1931.
[133] Vgl. ebd., S. 1931.

Algorithmus 2.4: A3C (Pseudocode)

```
// globale geteilte Parameter θ,θ_v und Zähler T =0 existieren
// Agentenspezifische Parameter θ',θ'_v exisiteren
1 INITIALISIERE Schrittzähler t
2 FÜR T = 1 bis max(T) WIEDERHOLE
3     SETZE Gradienten für θ,θ_v zurück
4     SYNCHRONISIERE agentenspezifische Parameter θ'=θ,θ'_v=θ_v
5     SETZE tstart = t
6     HOLE Zustand s_t
7     WIEDERHOLE bis Terminalzustand oder t-tstart = tmax
8           WÄHLE Aktion a_t nach der Regel π_θ(a_t|s_t) aus
9           FÜHRE Aktion a in Umgebung aus und beobachte Belohnung r_t
10          SETZE t = t+1
11          SETZE T = T+1
12    SETZE R={ 0, wenn s_t=Terminalzustand
              { V_θ(s_t), wenn s_t nicht Terminalzustand
13    FÜR i in {t-1,…,tstart} WIEDERHOLE
14          R = r_i+γR
15          SUMMIERE Gradienten für θ':dθ=dθ+Δ_θ'logπ_θ'(a_t|s_t)A_θ,θ_v(s_t,a_t)
16          SUMMIERE Gradienten für θ'_v:dθ_v=dθ_v+∂(R-V_θ'(s_i))^2/∂θ'_v
17    FÜHRE asynchrone Aktualisierung der Parameter θ,θ_v mit dθ,dθ_v
aus
```

Algorithmus 2.4: A3C Pseudocode für jeden Agenten in Anlehnung an Mnih(2016)[134]

3. Experiment

Als Methodik für diese wissenschaftliche Arbeit wurde ein Laborexperiment gewählt. Für die Auswahl der Methode wurde sich an der aktuellen Fachliteratur orientiert (siehe

[134] Vgl. Mnih, V. u. a. (2016), S. 1942.

[135],[136],[137],[138],[139],[140],[141],[142],). Darüber hinaus eignet sich ein Experiment zur Erhebung von Daten zum Vergleichen, da sie unter gleichen Umständen erstellt werden können. Des Weiteren beschreiben die Ergebnisse des Experimentes Zusammenhänge zwischen verschiedenen Variablen innerhalb des Experimentes.[143]

Für das Experiment sollen ein RL Agent das Spiel Breakout mit dem A3C und ein anderer Agent mit dem DQN Algorithmus spielen. Nach der Trainingsphase der Agenten, soll jeder der trainierten Agenten 1000 Spiele bestreiten. Die Ergebnisse jeder Partie werden quantitativ erfasst. Für die Trainingsphase der Agenten wird die erreichte Belohnung pro Episode beobachtet. Für den DQN Agenten wird zusätzlich der durchschnittliche maximale Q-Wert pro Episode aufgezeichnet. Für den A3C Agent werden die durchschnittliche maximale Wahrscheinlichkeit der Aktionen pro Episode der Regelfunktion aufgezeichnet. Um die Forschungsfragen zu beantworten, wird von allen Punkteständen der Durchschnitt pro Agenten berechnet und miteinander verglichen. Die Trainingszeit lässt sich zum einen durch die Anzahl der benötigten Episoden spezifizieren, die ein Agent benötigt, um eine bestimmte Genauigkeit zu erreichen oder die tatsächliche Zeit.[144] Die Implementierung des Experiments erfolgt in Python.

Im folgenden Kapitel soll der Aufbau des Experimentes genauer beschrieben werden und die verwendete Software kurz dargestellt werden. Anschließend erfolgt eine Beschreibung der Implementation des Experiments.

3.1 Aufbau des Experiments

Als Umgebung (oder auch Problemstellung) für den RL Agenten wird das ATARI Spiel Breakout verwendet. Dafür wird die bereits vorhandene Implementation von OpenAI[145] verwendet. Diese eignet sich für dieses Experiment, da sie speziell für einen

---

[135] Hado V. Hasselt (2010).
[136] Heess, N. u. a. (2013).
[137] Kaiser, L. u. a. (2019).
[138] Ong, H. Y. u. a. (2015).
[139] S. Gu u. a. (2017).
[140] Schulman, J. u. a. (2017).
[141] Schulman, J. u. a. (2015b).
[142] D. Silver u. a. (2014).
[143] Vgl. Baumgarth, C. u. a. (2009), S. 368.
[144] Vgl. Greg Brockman u. a. , S. 2.
[145] Vgl. ebd., S. 1 f.

Leistungsvergleich und als MEP erstellt wurde.[146] Bei dem Spiel geht es darum mit einem Schläger einen Ball auf Steine zu schlagen. Wenn der Ball einen Stein trifft, wird der Stein zerstört. Das Ziel des Spiels ist es, alle Steine mit dem Ball zu zerstören. Wenn alle Steine zerstört sind, startet die nächste Stufe, in der sich der Ball schneller bewegt. Pro Episode hat der Spieler fünf Leben. Man verliert ein Leben, wenn der Ball nicht von dem Schläger aufgefangen wird. Eine Episode endet, sobald alle Leben aufgebraucht sind.
Jeder Zustand des Spiels wird durch ein Bild des Spielfelds dargestellt. Das Bild hat eine Größe von 210 x 160 Pixel. Jeder Pixel hat einen Rot-Grün-Blau (RGB) Wert, der aus drei Zahlen besteht.[147] Der von der Umgebung zurückgemeldete Zustand hat die Form einer dreidimensionalen Liste. Für jeden Pixel gibt es einen Eintrag in dieser Liste mit drei natürlichen Zahlen, die den RGB Wert darstellen. Jede Zahl liegt dabei im Intervall [0,255]. Ein Beispiel für die Form des Zustands ist [[[200 72 72] [142 142 142] [0 0 0] ... [0 0 0] [200 72 72] [142 142 142]]]. Da jede Zahl eine Dimension des Zustandes ist, besteht ein Bild des Spiels aus $210 * 160 * 3 = 100.800$ Dimensionen.
Der Aktionsraum für jeden Zustand besteht aus genau vier Aktionen, die durch einen Index repräsentiert werden. Die möglichen Aktionen sind nichts machen, feuern, nach links oder nach rechts fahren. Eine Zuordnung der Aktionen ist in Tabelle 3.1 aufgezeigt. Jede ausgeführte Aktion wird zufällig für zwei, drei oder vier Bilder wiederholt.[148]

Tabelle 3.1: Aktionsraum von Breakout

Index	Aktion
0	Der Schläger macht nichts und bleibt auf der Stelle stehen.
1	Feuern. Diese Aktion startet eine Episode und feuert den Ball vom Schläger ab. Der Schläger bleibt auf der Stelle stehen.
2	Der Schläger fährt nach rechts.
3	Der Schläger fährt nach links.

Da ein Zustand immer aus einer festen Anzahl an Dimensionen besteht, wovon jede in einem festen endlichen Bereich liegt, ist der Zustandsraum endlich. Da es eine feste Anzahl an Aktionen gibt, die ebenfalls immer eine endliche Aktion ausführen, die in

[146] Vgl. Greg Brockman u. a., S. 1.
[147] Vgl. OpenAI (2021).
[148] Vgl. ebd.

einem endlichen Zeitraum abgeschlossen ist, ist der Aktionsraum ebenfalls endlich. Somit ist das Spiel Breakout ein endlicher MEP.
Die Belohnung des Agenten pro Episode wird durch den Spielstand repräsentiert. Pro zerstörten Stein erhöht sich der Spielstand.
Für das Experiment eignet sich das Spiel vor allem dadurch, dass es zum Ersten ein MEP ist, welches die Markov-Eigenschaft (Kapitel 2.2) erfüllt. Zweitens ist es ein hochdimensionales Problem. Drittens ist jeder vorangegangene Zustand für die Entscheidungsfindung in dem aktuellen Zustand egal.

3.2 Vorverarbeitung

Die Vorverarbeitung ist für viele ML-Projekte notwendig. Wie von Mnih (2013) beschrieben,[149] ist die Verarbeitung von den ursprünglichen Bildern des Zustandes aufgrund der hohen Dimensionszahl sehr Rechenaufwendig. Um das Training des Agenten zu verbessern werden in dieser Thesis als erstes Daten aus dem Bild entfernt, die der Agent nicht benötigt. Dies geschieht in den ursprünglichen Implementationen des DQN[150] und des A3C[151] Algorithmus nicht. Am oberen Bildrand sind beispielsweise ein freier Raum und der Spielstand abgebildet, welche nicht für das Erlernen des Spiels relevant sind. Somit reduziert sich die Anzahl der Dimensionen auf $176 * 160 * 3 = 84.480$ In Abbildung 3.1 ist der Unterschied vor und nach der Bearbeitung des Bildes aufgezeigt.

[149] Vgl. Mnih, V. u. a. (2013), S. 5 f.
[150] Vgl. ebd., S. 5 f.
[151] Vgl. Mnih, V. u. a. (2016), S. 1939 f.

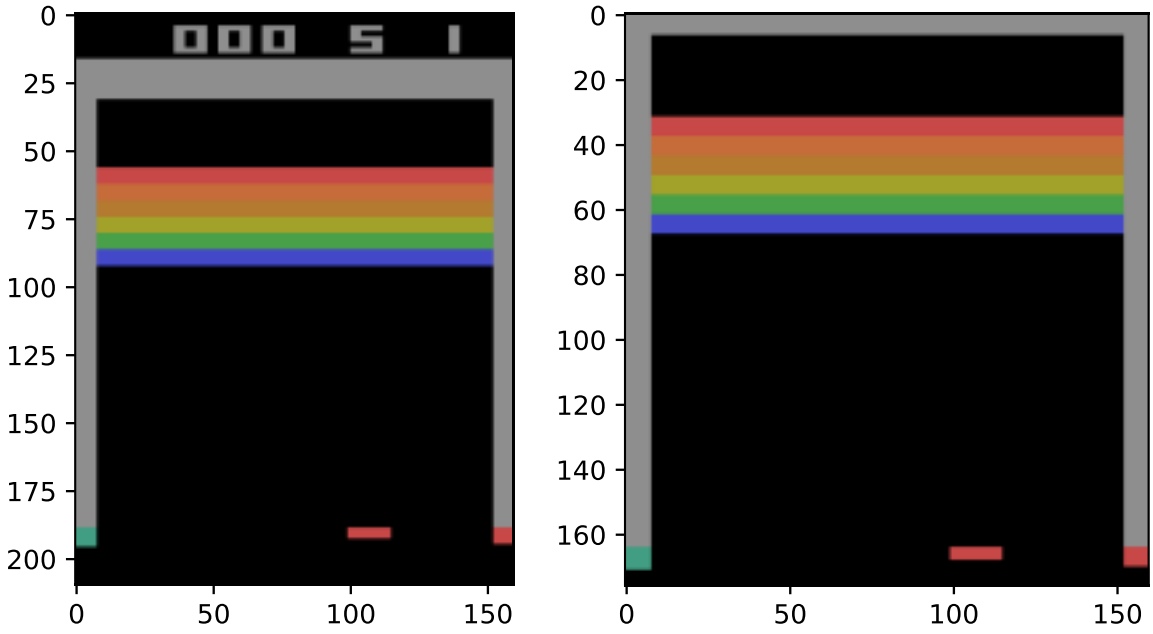

Abbildung 3.1: Vergleich originales Bild und erste Vorverarbeitung, (links) Originales Bild von Breakout-v0 mit der Größe 210x160x3 Pixel (100.800 Dimensionen), (rechts) bearbeitetes Bild ohne Spielstand und Freiraum am unteren und oberen Rand mit der Größe 176x160x3 (84.480 Dimensionen)

Um die zu verarbeitenden Dimensionen weiter zu verringern, wird das zuvor bearbeitete Bild auf die Größe 84 x 84 Pixel reduziert. Zusätzlich dazu wird das Bild von einem RGB Bild mit drei Dimensionen pro Pixel in ein Graustufenbild mit einer Dimension pro Pixel umgewandelt. Somit reduziert sich die Anzahl der Dimensionen von 100.800 auf $84 * 84 * 1 = 7.056$ Dimensionen. In Abbildung 3.2 ist das weiterverarbeitete Bild aufgezeigt.

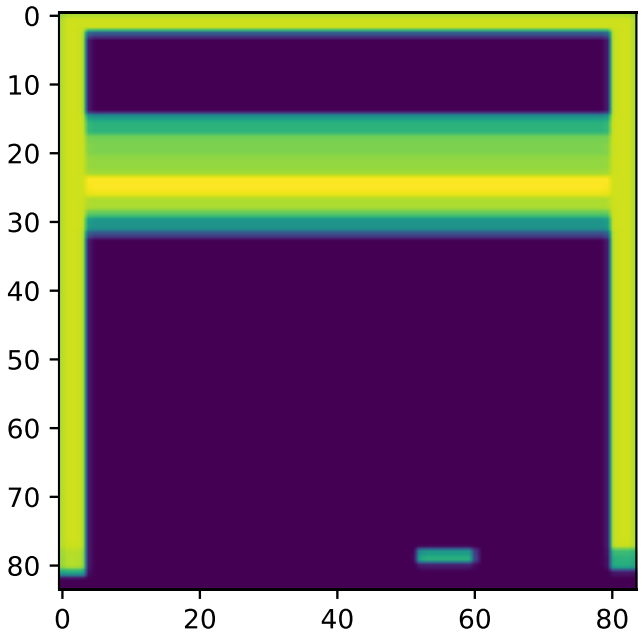

Abbildung 3.2: Vorverarbeitetes Bild in Graustufe in der Größe 84x84x1 Pixel

Eine Eigenheit der ATARI Spiele ist, dass manche Elemente des Spielzustandes nicht mit jedem Bild gerendert werden. Um dieses Problem zu umgehen, werden immer die zwei letzten beobachteten Zustände beziehungsweise Bilder miteinander verglichen. Aus den beiden Bildern wird ein kombiniertes Bild erstellt, indem Pixel für Pixel verglichen wird und jeweils das Maximum der beiden Pixel für das neue Bild verwendet wird.[152]
Diese Schritte bilden den ersten Schritt der Vorverarbeitung der Zustände.
Der zweite Schritt besteht darin, vier Bilder, die alle durch den ersten Schritt vorverarbeitet wurden, miteinander zu verknüpfen. Dies hat den Grund, da man aus einem Bild sich bewegende Elemente, wie zum Beispiel einen Ball, schwer von anderen sich nicht bewegenden Objekten unterscheiden kann. Zusätzlich dazu lässt sich so erkennen, in welche Richtung sich ein Objekt bewegt. So soll der Agent in dem Beispiel von Breakout unterscheiden können, ob der Ball sich nach oben beziehungsweise nach unten bewegt. Zu Beginn einer Episode wir der erste Zustand, wie im ersten Schritt beschrieben, vorverarbeitet und anschließend vier Mal übereinandergelegt. Somit entsteht eine Liste aus vier Bildern. Während einer Episode wird nach jeder durchgeführten Aktion der neue Zustand mit dem ersten Schritt vorverarbeitet und der Liste hinzugefügt. Das älteste Bild

[152] Vgl. Seita, D. (2016).

wird anschließend aus der Liste entfernt. Somit enthält die Liste immer die letzten vier betrachteten Bilder.

Ein weiteres Problem ist, dass nach dem Verkleinern des Bildes vier direkt aufeinander folgende Bilder nicht ausreichen würden, um Bewegungen erkennen zu können.[153] Um dies zu verbessern wird nur jedes vierte Bild betrachtet.[154] Um dies zu erreichen wählt der Agent eine Aktion, die für die nächsten vier Schritte wiederholt wird. Der Agent erhält als neuen Zustand das Bild, nachdem die Aktion vier Mal ausgeführt wurde. Um dieses Verhalten abzubilden wird von Breakout eine modifizierte Version verwendet. Diese ist „BreakoutDeterministic-v4“ [155] welche ebenfalls in der OpenAI Gym Bibliothek vorhanden ist. Die zurückgemeldete Belohnung für die ausgeführte Aktion ist die summierte Belohnung, die über die vier aufeinanderfolgenden Schritte erzielt wurde.

Neben dem Vorverarbeiten des Zustandes, um den Trainingsprozess zu optimieren, wird der Aktionsraum ebenfalls optimiert. In Tabelle 3.1 sind die Aktionen mit dem Index 0 und 1 fast gleich. Die Aktion 1 hat den Zusatz, dass sie den Ball abfeuert, wodurch eine Episode gestartet wird. Damit der Agent nicht erst noch lernen muss, wie man eine Episode startet, wird die Aktion 0 aus dem Aktionsraum entfernt.

Zuletzt wird die Belohnung der Umgebung begrenzt. Alle positiven Belohnungen, die der Agent erhält, werden auf 1 beziehungsweise alle negativen Belohnungen werden bei -1 gekappt und 0 bleibt unverändert. Dies hilft dabei die Verluste des Gradientenverfahrens zu limitieren und die Leistungsfähigkeit des Agenten zu verbessern, da dieser durch die Änderung der Belohnung nicht zwischen Belohnungen verschiedener Größe unterscheiden muss.[156]

3.3 Implementation

Im folgenden Kapitel wird die Implementation des DQN Agenten und A3C Agenten beschrieben. Die Implementation erfolgt in der Programmiersprache Python 3.7[157]. Für die Implementation der fNN wird die Bibliothek Keras[158] verwendet, welche auf dem

---

[153] Vgl. Seita, D. (2016).
[154] Vgl. Mnih, V. u. a. (2015), S. 534.
[155] OpenAI (2016).
[156] Vgl. Mnih, V. u. a. (2015), S. 534.
[157] van Rossum, G., Drake, F. L. (2009).
[158] Chollet, F., others (2015).

Framework TensorFlow[159] basiert. Keras ist eine Programmierschnittstelle, um NN und dazugehörige Parameter einfach zu definieren und zu erstellen.[160] TensorFlow ist eine Programmierschnittstelle, die die Erstellung von Neuronen und deren Verbindung abstrahiert und die Berechnung von verschiedenen Gradientenverfahren ermöglicht. Die Erweiterung TensorBoard von TensorFlow wird verwendet, um die Trainingsdaten und Modelle zu speichern und um den Trainingsprozess zu visualisieren und zu beobachten.[161]

Die Umgebung wird durch die in Kapitel 3.1 erwähnte Bibliothek Gym von OpenAI bereitgestellt.

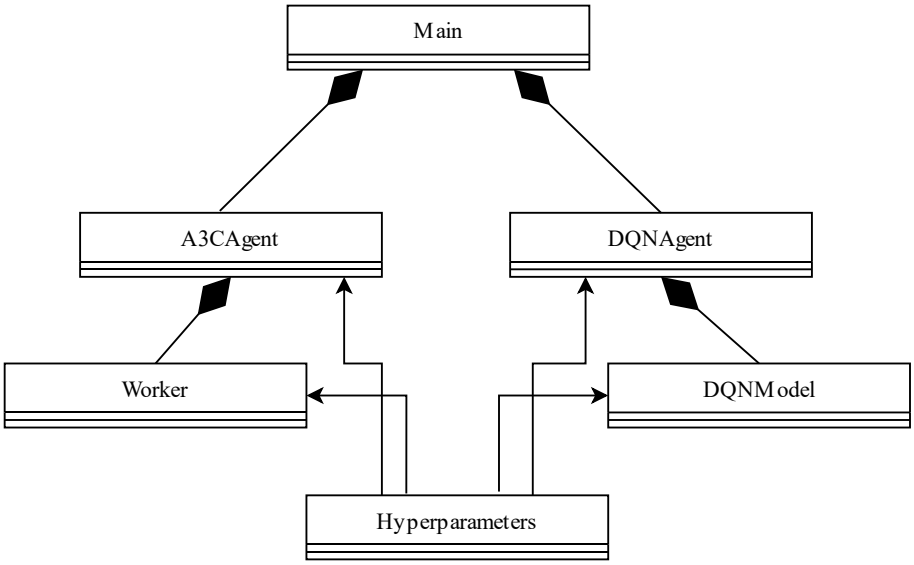

Abbildung 3.3: Komponenten der Implementation

In **Fehler! Verweisquelle konnte nicht gefunden werden.** ist ein Modell der K omponenten der Implementation dargestellt.

Bei der Implementation wurde die Funktionalität der Modularität der einzelnen Komponenten vorgezogen. Die Hauptfunktionen der einzelnen Komponenten werden innerhalb dieses Kapitels aufgezeigt und erläutert.[162]

---

[159] Abadi, M. u. a. (2016).
[160] Vgl. Chollet, F., others (2015).
[161] Vgl. Abadi, M. u. a. (2016), S. 1 f.
[162] Die komplette Implementation ist unter https://github.com/AndreasKretschmer/Deep_Reinforcement_Learning_Thesis einsehbar oder in den beigefügten Dateien mit vorhanden.

3.3.1 Main

Die Main Komponente implementiert die beiden Agenten. Mithilfe des Python Terminals lässt sich die Main Komponente mit verschiedenen Parametern aufrufen. Somit kann man das Training und die Evaluierung der Agenten starten. Die Parameter, die beim Aufruf mitgegeben werden können, sind das zu verwendende Modell und der zu verwendende Modus. Der Modell Parameter kann die Werte DQN oder A3C beinhalten. Je nach Wert wird ein Agent der Klasse DQNAgent oder A3CAgent instanziiert. Der Modus Parameter gibt an, ob das ausgewählte Modell trainiert oder evaluiert werden soll. Dafür kann der Parameter die Werte train oder eval beinhalten. Je nach Wert wird somit die Methode train() oder Evalualte() des jeweiligen Agenten aufgerufen.

3.3.2 DQN Algorithmus

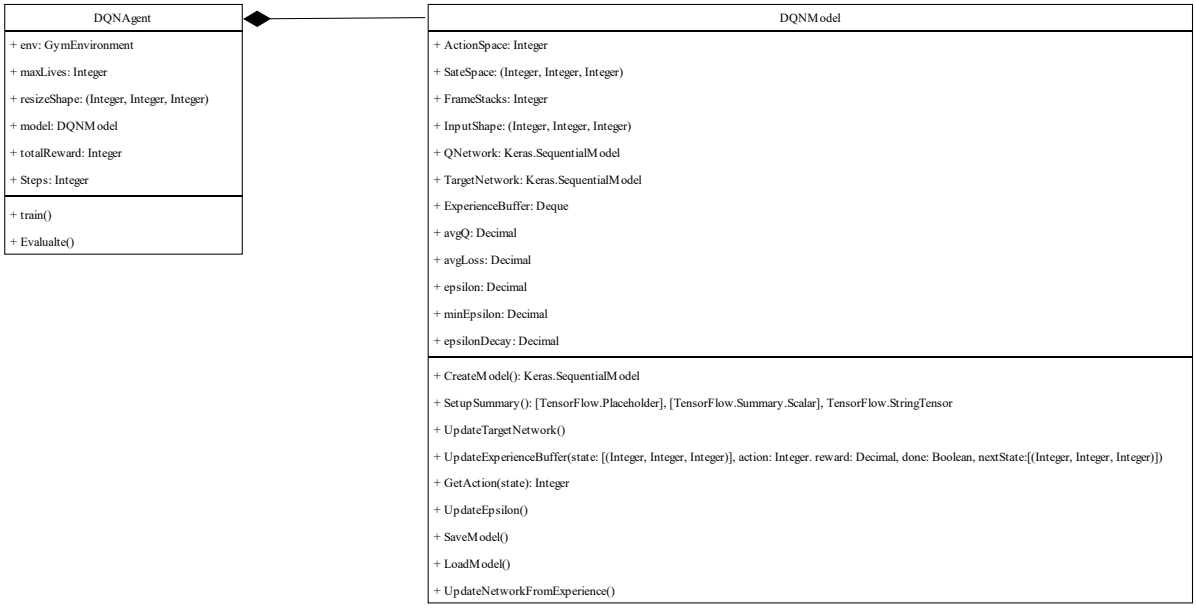

Abbildung 3.4: DQN Klassenmodell

Die Implementierung des DQN Algorithmus besteht aus zwei Klassen. Zum einen die Klasse des Agenten und zum anderen die Klasse des DQN Modells. Ein Klassenmodell der beiden Klassen ist in Abbildung 3.4 dargestellt. Der Übersichtlichkeit halber sind nur trainingsrelevante Klassenbestandteile im Klassenmodell enthalten.

Die Klasse DQNModel implementiert das Keras-Modell in Form eines fNN für das DQN und das Zielnetz. Beide NN haben den gleichen Aufbau. Zusätzlich dazu enthält die Klasse den Erfahrungsspeicher, sowie Informationen über das maximale und minimale

Epsilon und über die Abnahmegröße von Epsilon für den abnehmenden ε-greedy Algorithmus. Diese werden bei der Instanziierung der Klasse erstellt beziehungsweise initialisiert.

Das Erstellen des DQN Modells ist in der Methode CreateModel() implementiert. Die Eingabeschicht hat das Format 84x84x4, welches das vorverarbeitete Bild der Umgebung widerspiegelt. Darauf folgt die erste versteckte Faltungsschicht mit 32 Filterkernen. Jeder Filterkern hat das Format 8x8. Anstelle einer Pooling-Schicht, wird nach jedem Faltungsschritt eine 4x4 Matrix an Werten übersprungen. Anschließend folgt die zweite versteckte Faltungsschicht mit 64 Filterkernen mit dem Format 4x4 und einer Schrittweite von 2x2. Danach folgt die letzte Faltungsschicht mit 64 Filterkernen mit dem Format 3x3 und einer Schrittweite von 1x1. Darauf folgt eine vollvernetzte Schicht mit 512 Neuronen. Als Aktivierungsfunktion wird für alle Schichten ReLU verwendet. Zuletzt folgt die Ausgabeschicht mit 3 Neuronen, die die möglichen Aktionen des Agenten abbilden. Als Gradientenverfahren wird RMSProp verwendet. Dies eignet sich besonders für Aktualisierungen der Gewichte der Neuronen in einem Stapelverfahren, welches für das Training des DQN Modells verwendet wird.[163] Dieses Modell stammt von Mnih (2015) und es wurde gewählt, da es die besten Ergebnisse erzielt.[164]

Mit Hilfe der der Methode UpdateTargetNetwork() wird das Zielnetz mit den Gewichten des DQN aktualisiert. Die Methode UpdateExperienceBuffer() fügt einen neues Erfahrungstupel zum Erfahrungsspeicher hinzu. Eine auszuführende Aktion erhält man mithilfe der Methode GetAction() auf Basis eines Zustandes aus dem aktuellen Modell.

Die Aktualisierung des Modells wird durch die Methode UpdateNetworkFromExperience() gestartet. Dabei wird zu Beginn der aktuelle Epsilonwert durch die Methode UpdateEpsilon() aktualisiert. Anschließend werden zufällig eine Anzahl von Erfahrungstupel aus dem Erfahrungsspeicher ausgewählt. Für jedes Tupel wird anschließend ein Zielwert für die Verlustberechnung mithilfe des Zielnetzes berechnet. Danach wird im Stapelverfahren das DQN für alle Tupel aus dem Erfahrungsspeicher aktualisiert.

---

[163] Vgl. Hinton, G. (o.J.), S. 29 f.
[164] Vgl. Mnih, V. u. a. (2015), S. 534.

Zusätzlich dazu implementiert die Klasse Methoden für das Laden (LoadModel()) und speichern (SaveModel()) eines Modells und für das Protokollieren der Leistungsdaten während des Trainings (SetupSummary()).

Die Klasse DQNAgent implementiert den DQN Agenten, der mit der Umgebung agiert. Dafür instanziiert der Agent eine Klasse der OpenAI Umgebung und des DQN Modells. Zusätzlich dazu erhält der Agent Informationen über die Maximale Anzahl an Leben und die Form der zu verarbeitenden Bilder nach der Vorverarbeitung aus der Hyperparameter Klasse.
Mit der Methode train() lässt sich das Training des Agenten starten. Sie implementiert den DQN Pseudocode (Algorithmus 2.3). Der Algorithmus startet eine Schleife für eine festgelegte Anzahl an Episoden. Jede Iteration der Schleife startet eine neue Episode. Dafür wird die Umgebung zurückgesetzt. Anschließend wird die Aktion 1 (vergleiche Tabelle 3.1) mehrfach wiederholt. Die Anzahl der Wiederholungen wird zufällig im Bereich 0 bis 30 (vergleiche Tabelle 3.2) ausgewählt. Dies wird gemacht, damit der Agent die Episoden möglichst immer in unterschiedlichen Ausgangssituationen startet. Dies verhindert, dass der Agent eine Sequenz von Aktionen erlernt.[165]
Der daraus resultierende Zustand wird wie in Kapitel 3.2 vorverarbeitet und ist der erste beobachtete Zustand der Episode für den Agenten. Anschließend folgt eine Schleife, in der der Agent die folgenden Schritte so lange wiederholt, bis die Episode abgeschlossen ist. Als erstes wählt der Agent eine Aktion auf Basis des vorverarbeiteten Zustandes. Die ausgewählte Aktion wird in der Umgebung ausgeführt. Der Agent erhält von der Umgebung den neuen Zustand, die Belohnung, die Anzahl der Leben und die Information zurück, ob die Episode beendet wurde. Der neue Zustand wird ebenfalls wie in Kapitel 3.2 vorverarbeitet und dient dem Agenten für die nächste Iteration der Schleife als Basis für die Auswahl einer Aktion, falls die Episode nicht geendet hat. Anschließend wird der Erfahrungsspeicher wie in Kapitel 2.7 beschreiben aktualisiert. Darauf folgt die Aktualisierung des fNN mithilfe des Erfahrungsspeichers (vergleiche Kapitel 2.7). Zum Schluss wird das Zielnetz im vorgegebenen Intervall aktualisiert und die Leistungsdaten des Agenten für die ausgeführte Episode protokolliert.

[165] Vgl. Marlos C. Machado u. a. (2017), S. 14.

Die Evaluierung des DQN Modells wird mithilfe der Methode Evaluate() gestartet. Diese hat den gleichen Ablauf, wie die train() Methode. Allerdings wird das Modell nicht mehr aktualisiert und es werden 1000 Episoden durchlaufen. Zum Beginn der Methode wird ein bereits trainiertes Modell geladen, welches evaluiert werden soll.

3.3.3 A3C Algorithmus

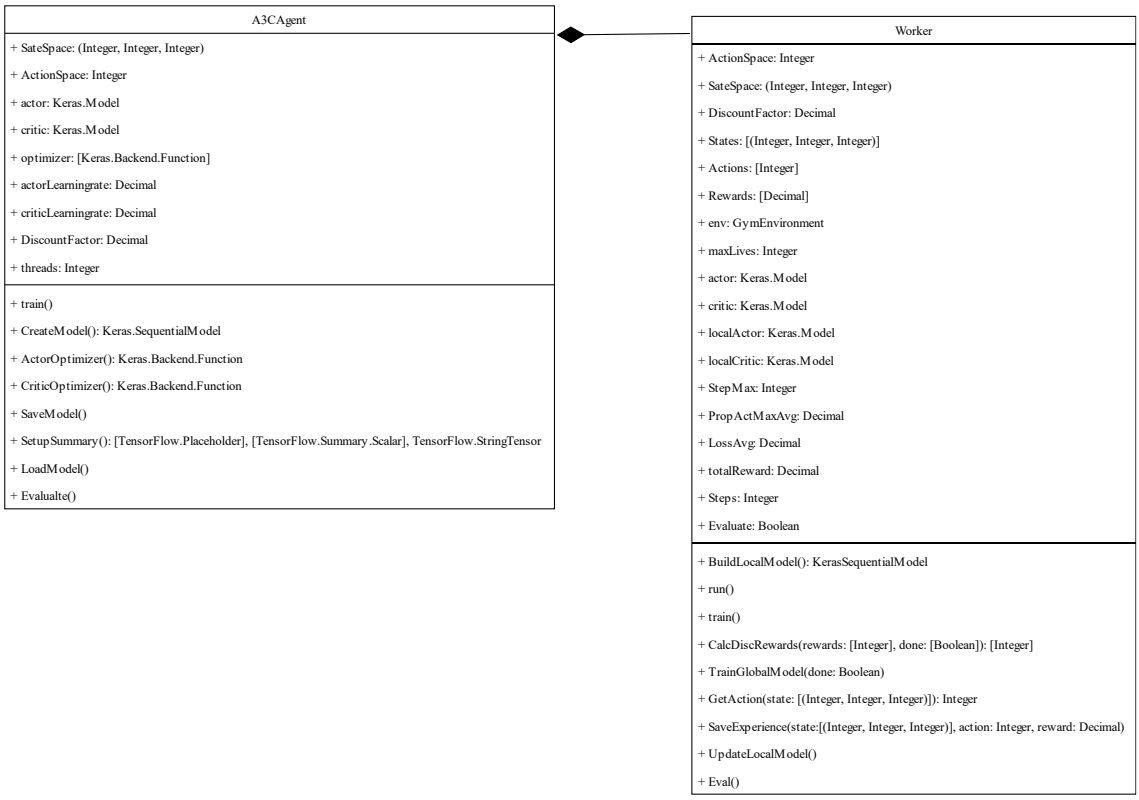

Abbildung 3.5: A3C Klassenmodell

Die Implementierung des A3C Algorithmus besteht aus zwei Klassen. Zum einen die Klasse für einen globalen A3C Agenten und zum anderen eine Klasse für die parallellaufenden Arbeiter (Worker). Ein Klassenmodell der beiden Klassen ist in Abbildung 3.5 zu sehen. Aus Übersichtlichkeitsgründen sind nur Trainingsrelevante Klassenbestandteile im Modell enthalten.

Die A3CAgent-Klasse implementiert ein globales Keras-Modell, welches aus einem Model für den Actor und einem Model für den Critic besteht. Die beiden Modelle sind bis auf die Ausgabeschicht gleich. Dabei teilen sich die beiden Modelle alle Gewichte zwischen den Neuronen, wobei nur die Ausgabeschicht für die einzelnen Modelle getrennt sind.

Als Eingabe dient das Bild der Umgebung, welches wie beim DQN Agenten vorverarbeitet ist. Die Eingabe hat auch das Format 84x84x4. Die erste Faltungsschicht hat 16 Filterkerne mit dem Format 8x8. Die Schrittweite beträgt 4x4 und wird anstelle einer Pooling-Schicht verwendet. Als nächstes folgt eine weitere Faltungsschicht mit 32 Filterkernen mit dem Format 4x4 und einer Schrittweite von 2x2. Die Ergebnisse der zweiten Faltungsschicht werden anschließend an die erste vollvernetzte tiefe Schicht übergeben. Diese besteht aus 256 Neuronen. Die genannten Schichten verwenden alle ReLU als Aktivierungsfunktion. Als letztes folgt für das Actor-Modell die Ausgabeschicht mit 3 Neuronen für die Anzahl der Aktionen. Als Aktivierungsfunktion wird dafür Softmax verwendet. Diese ist notwendig, um für jede Aktion eine Wahrscheinlichkeit definieren zu können, welche Aktion der Agent präferiert.[166] Die Summe aller Wahrscheinlichkeiten der Aktionen ergibt 1.

Die Ausgabeschicht des Critic-Modells besteht aus einem Neuron, welches das Ergebnis der Wertfunktion widerspiegelt. Dafür wird die lineare Aktivierungsfunktion verwendet. Dabei ist das prognostizierte Ergebnis gleich dem Ausgabewert. Die Modelle werden durch die Methode CreateModel() erstellt. Das Modell wurde von Max Jaderberg (2016) übernommen[167]. Allerdings wird auf das lange Kurzzeitgedächtnis Netzwerk verzichtet. Im Vergleich zum DQN Modell ist eine Faltungsschicht weniger vorhanden. Es wurde dieses Modell gewählt, da es durch die weniger komplexe Architektur auch weniger Ressourcen benötigt und trotzdem gute Ergebnisse in dem Forschungsprojekt von Jaderberg lieferte.[168]

Zusätzlich beinhaltet die Klasse zwei Keras-Funktionen, die für die Berechnungen des Verlustes und der Gradienten für das Gradientenverfahren benötigt werden. Die Keras-Funktion für den Critic und den Actor erstellt die Methode CriticOptimizer()[169] beziehungsweise ActorOptimizer()[170]. Die Optimierungsmethoden wurden aus dem GitHub-Projekt Deep-RL-Keras von Hugo Germain übernommen. Als Gradientenverfahren wird auch hier RMSProp verwendet, da wie beim DQN Modell das A3C Modell in kleinen Stapeln aktualisiert wird.

---

[166] Vgl. Sutton, R. S., Barto, A. (2018), S. 258.
[167] Max Jaderberg u. a. (2016).
[168] Vgl. ebd., S. 12.
[169] Germain, H. (2020b).
[170] Germain, H. (2020a).

Die train()-Methode erstellt eine festgelegte Anzahl an Instanzen der Klasse Worker und ruft die Methode train() jeder angelegten Instanz der Klasse Worker auf. Dies startet den Trainingsprozess für jeden einzelnen Arbeiter. Zusätzlich implementiert die A3CAgent-Klasse Methoden für das Speichern (SaveModel()) und Laden (LoadModel()) des globalen Modells und für das Protokollieren der Trainings- und Evaluierungsergebnisse (SetupSummary()). Die Evaluierung des Modells wird über die Methode Evaluate() gestartet. Diese ruft die Methode Eval() der Klasse Worker auf.

Die Klasse Worker implementiert die Spielumgebung, diverse Hyperparameter werden initialisiert und es wird ein Speicher für die Zustände, Aktionen und Belohnungen initialisiert. Darüber hinaus hat jede Instanz der Klasse ein Modell des globalen A3C Agenten und ein lokales Modell, welches für diese Instanz eigenständig ist. Das lokale Modell des Arbeiters hat das gleiche Format wie das globale Modell hat. Das lokale Modell wird über die Methode BuildLocalModel() erstellt. Zusätzlich dazu implementiert die Klasse verschiedene Variablen für das Erfassen von Werten, um den Lernprozess zu überwachen.
Die Methode train() startet den Trainingsprozess für eine Instanz der Worker Klasse. Die Methode implementiert den A3C Pseudocode (Algorithmus 2.4). Die Funktion ist ähnlich aufgebaut, wie die Methode train() des DQN Agenten (Kapitel 3.3.2). Sie unterscheiden sich nur in der Optimierungsfunktion für das Modell und dem Erfahrungsspeicher. Für den A3C Agenten werden für jeden Schritt in einer Episode die beobachteten Zustände, die für den jeweiligen Zustand ausgeführte Aktionen und die daraus resultierende Belohnung für die jeweilige Episode gespeichert. Dies implementiert die Methode SaveExperience(). Beim Start einer neuen Episode wird der Speicher gelöscht. Der Speicher dient nur dazu, die Vorteilsfunktion zu berechnen, welche für das Gradientenverfahren verwendet wird. Das Gradientenverfahren wird in einem festgelegten Intervall an Schritten innerhalb einer Episode oder zum Ende einer Episode ausgeführt. Dabei wird immer nur das globale Modell trainiert. Dies wird durch die Methode TrainGlobalModel() implementiert. Zu Beginn wird mithilfe der Methode CalcDiscountedRewards() für jeden Zustand die rabattierte Belohnungssumme berechnet (ein Teil der Formel (2.19)). Anschließend werden für alle gespeicherten Zustände die Ergebnisse der Zustands-Wertfunktion mithilfe des Critics des lokalen Modells

berechnet. Danach wird aus den Ergebnissen und den rabattierten Belohnungen die Vorteilsfunktion berechnet (Formel (2.19)). Mithilfe der Optimierungsfunktionen der A3C Agenten Klasse wird anschließend das globale Modell aktualisiert. Zum Schluss werden die Gewichte des lokalen Modells mit den Gewichten des globalen Modells ersetzt.
Die Methode Evaluate() startet die Evaluierung des Modells. Dabei werden 1000 Episoden durchlaufen. Die Methode ist genauso aufgebaut, wie die train() Methode ohne die Aktualisierung des Modells und zu Beginn wird ein bereits vorhandenes Modell geladen.

3.3.4 Hyperparameter

Die Komponente Hyperparameter besteht aus der Klasse Hyperparameter. Diese beinhaltet alle konstanten Trainingsparameter, wie zum Beispiel der Rabattfaktor, Epsilon und zusätzliche Parameter, wie beispielsweise Exportpfade oder Modellnamen für die Protokolldateien, der beiden Agenten.

Tabelle 3.2: allgemeine Hyperparameter

Hyperparameter	Wert	Beschreibung
Rabattfaktor	0.99	Spezifiziert den Rabattfaktor γ für beide Agenten.
Anzahl mache nichts Schritte	30	Anzahl der maximalen Schritte zu Beginn einer Episode, in denen der Agent nichts unternimmt.
Aktionsraum Größe	3	Gibt die Anzahl der verfügbaren Aktionen an.
Lernrate	0.00025	Gibt die Lernrate α des Agenten an.

Die allgemeinen Parameter gelten für beide Agenten. Diese sind in Tabelle 3.2 einzusehen. Parameter, die für den eigentlichen Trainingsprozess irrelevant sind (wie zum Beispiel Exportpfade) sind in dieser Übersicht nicht vorhanden.
Für die Werte der Hyperparameter dienen die Publikationen von Mnih (2015)[171] und Mnih (2016)[172] der verwendeten Algorithmen als Vorlage.

[171] Vgl. Mnih, V. u. a. (2015), S. 538.
[172] Vgl. Mnih, V. u. a. (2016), S. 1939 f.

Der Wert für den Rabattfaktor wurde so gewählt, um zukünftige Belohnungen stark mit in die Bewertung einfließen zu lassen. Dies ist vor allem hilfreich, wenn der Agent in mehreren Schritten einen Zustand vorbereitet.
Der Wert für die Schritte zu Beginn einer Episode, in denen der Agent nichts machen soll, wurde so gewählt, dass der Agent möglichst oft in unterschiedlichen Situationen eine Episode startet. Bei dem gewählten Wert verliert der Agent gegebenenfalls ein oder zwei Leben, bevor er aktiv die Aktionen auswählt. Somit lernt der Agent keine Sequenz von Aktionen für eine Episode. Die Lernrate ist für alle Agenten gleich gewählt. Der Wert wurde niedrig festgelegt, um lokale Minima schnell zu finden und um diese schneller zu überspringen.

Tabelle 3.3: DQN Hyperparameter

Hyperparameter	Wert	Beschreibung
Erfahrungsspeichergröße	400000	Gibt die Anzahl der maximalen Einträge im Erfahrungsspeicher an.
Anzahl der Trainingsbeispiele	32	Gibt die Anzahl an Trainingsbeispiele für die Aktualisierung des NN an.
Explorationsrate zum Start	1	Gibt den Initialwert für ε für den ε-greedy Algorithmus an.
Minimale Explorationsrate	0,1	Gibt den finalen ε-Wert für den ε-greedy Algorithmus an.
Abnahmeschritte für die Explorationsrate	1000000	Gibt die Anzahl an Schritten an, über die der Initialwert von ε linear reduziert wird, bis zu seinem Finalwert.
Minimale Anzahl an Erfahrungsspeichereinträgen	50000	Gibt die minimale Anzahl an Trainingsbeispielen für den Erfahrungsspeicher an.
Zielnetz Aktualisierungsfrequenz	10000	Gibt die Anzahl an Schritten an, nachdem das Zielnetzwerk mit dem Q-Netzwerk aktualisiert werden soll.

Die Hyperparameter für den DQN Agenten sind in der Tabelle 3.3 aufgelistet. Der Erfahrungsspeicher wurde auf den Wert begrenzt, da er für jeden Erfahrungseintrag

Hauptspeicher benötigt. Mit 400000 Einträgen kann man die benötigten Ressourcen etwas limitieren. Die Explorationsrate zum Beginn einer Episode liegt bei 1, um am Anfang den Agenten die Umgebung erkunden zu lassen und um möglichst unterschiedliche Erfahrungen zu sammeln. Über die ersten Millionen Schritte soll die Rate linear abnehmen, um den Agenten die Aktionen immer mehr durch das NN auswählen zu lassen. Die minimale Explorationsrate liegt dann bei 0,1 um auch später, wenn der Agent bereits Erfahrungen gesammelt hat, die Umgebung gelegentlich zu erkunden, um unbekannte Zustände zu erkunden. Dies ist vor allem hilfreich bei Zuständen, die der Agent erst spät im Trainingsprozess erreicht, um auch dort gegebenenfalls neue Entdeckungen zu machen. Die minimale Anzahl an Erfahrungsspeichereinträgen ist so gewählt, sodass ausreichend Einträge für die Aktualisierung des NN zur Verfügung stehen und das Modell nicht über mehrere Episoden mit den gleichen Erfahrungen trainiert wird.

Tabelle 3.4: A3C Hyperparameter

Hyperparameter	Wert	Beschreibung
Anzahl Arbeiter	8	Gibt die Anzahl an Arbeitern (Threads) an, die parallel ausgeführt werden.
Actor-Critic Aktualisierungsfrequenz	20	Gibt die Anzahl an Schritten pro Episode an, nachdem ein Training des Modells stattfinden soll.

Die Hyperparameter des A3C Agenten sind in Tabelle 3.4 aufgelistet. Die Anzahl an parallelen Arbeitern ist auf 8 festgelegt, da die verwendeten Ressourcen nicht mehr zulassen. Die Aktualisierungsfrequenz des Actors und Critics wurde auf den Wert 20 festgelegt, damit in einer Episode nach dieser Anzahl an Schritten eine Aktualisierung des Netzwerkes stattfindet. Da eine Episode bereits zum Start des Trainings 100 bis 200 Schritte benötigt, wird die Aktualisierung mehrfach pro Episode ausgeführt. Dies wird gemacht, um das Modell nicht nach jeder Aktion zu trainieren, sondern in kleinen Stapeln. Auf eine Optimierung der Hyperparameter wurde in dieser Arbeit verzichtet.

4. Evaluation

Das Training und die Evaluation der beiden Modelle erfolgte auf einem Rechner mit einer NVIDIA GeForce GTX 1050 Grafikkarte und einem Intel i7-8650U Prozessor. Im

folgenden Kapitel werden die Ergebnisse des Trainings und der Evaluation beschrieben und erläutert.[173]

4.1 Trainingsergebnisse

Für den DQN Agenten wurde zum einen die Belohnung pro Episode während des Trainingsprozesses protokolliert. Diese wurde gewählt, um die Leistungsfähigkeit der einzelnen Agenten miteinander vergleichen zu können. Zum anderen wurde der durchschnittliche maximale Q-Wert der möglichen Aktionen für das Training erfasst. Dieser bildet das Ergebnis der Aktion-Wertfunktion ab und wurde gewählt, um zu überprüfen, welche rabattierte Belohnung das Modell für einen gegebenen Zustand erwartet. Darüber ist der maximale Q-Wert ausschlaggebend für die Auswahl der Aktion. Das Training für das DQN Modell benötigte knapp 135 Stunden. Dabei hat der Agent circa 9,6 Millionen Bilder über 43000 Episoden betrachtet. Die maximale Belohnung, die der Agent in einer Episode erreicht hat, beträgt 18. Die kleinste Belohnung in einer Episode beträgt 0. Der maximale durchschnittliche Q-Wert in einer Episode beträgt 0,2313 und der kleinste maximale durchschnittliche Q-Wert ist 0. Der Q-Wert 0 kann allerdings nur zustande kommen, wenn der Agent in einer Episode nur zufällige Aktionen auswählt, da dann kein Q-Wert berechnet wird.

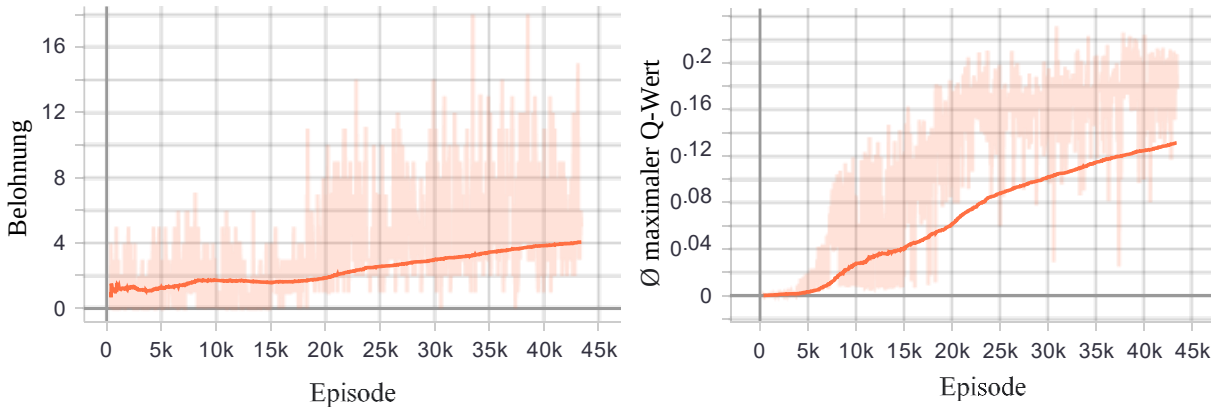

Abbildung 4.1: Trainingsverlauf des DQN Agenten, (links) der gleitende Mittelwert für die Belohnung pro Episode des DQN Agenten, (rechts) der gleitende Mittelwert für den durchschnittlichen maximalen Q-Wert pro Episode

[173] Die kompletten Protokolle der Trainingsläufe und Evaluierungsläufe sind unter https://github.com/AndreasKretschmer/Deep_Reinforcement_Learning_Thesis/tree/master/logs inklusiver der trainierten Modelle und der Rohdaten mithilfe von TensorBoard oder in den beigefügten Dateien einsehbar.

Abbildung 4.1 zeigt links den gleitenden Mittelwert für die Belohnung pro Episode und rechts den gleitenden Mittelwert für den durchschnittlichen maximalen Q-Wert der möglichen Aktionen pro Episode. Gezeigt sind die Messungen aus dem besten Trainingsergebnis von drei Versuchen. Dabei ist zu erkennen, dass der maximale Q-Wert viel stetiger größer wird als die Belohnung. Wenn man die Kurve der Belohnungen betrachtet, sieht es fast so aus, als würde der Agent gar nicht lernen. Erst nach knapp 17000 Episoden steigt der Mittelwert der Belohnung pro Episode stetig. Allgemein sind die Belohnungen pro Episode sprunghaft. Das liegt vor allem daran, dass gerade zu Beginn des Trainings viele Aktionen zufällig ausgewählt werden. Dazu kommt, dass kleine Änderungen an den Gewichten des Modells große Auswirkungen auf die Auswahl der Aktionen haben können, die der Agent auswählt. Betrachtet man aber in dem gleichen Zeitraum die Q-Werte, kann man sehen, dass diese stetig steigen und stabiler wirkt als die Belohnung. Dies liegt vor allem an den kleinen Anpassungen der Gewichte und der Limitation der Belohnung auf -1 bis 1. Dadurch werden die Gewichte des NN immer nur geringfügig angepasst. Dies führt allerdings wiederrum zu einer erhöhten Trainingszeit.
Das Training des DQN Modells wurde nach der angegebenen Zeit aus zeitlichen Gründen beendet, obwohl der Agent sich vermutlich danach noch weiter verbessern würde.

Für den A3C Agenten wurde ebenfalls die Belohnung pro Episode protokolliert, um die Leistung der beiden Agenten vergleichen zu können. Da für den A3C Agenten die Aktion-Wertfunktion nicht zur Verfügung steht, wurde die maximale durchschnittliche Wahrscheinlichkeit der möglichen Aktionen pro Episode zusätzlich erfasst. Diese wurde gewählt, da die Wahrscheinlichkeit ausschlaggebend für die Wahl der Aktionen ist, parallel zum Q-Wert des DQN Agenten.
Das Training des A3C Agenten dauerte knapp 44 Stunden. Dabei betrachtete der Agent circa 23 Millionen Bilder in 35000 Episoden. Die maximale Belohnung in einer Episode beträgt 321. Die geringste Belohnung beträgt 0.

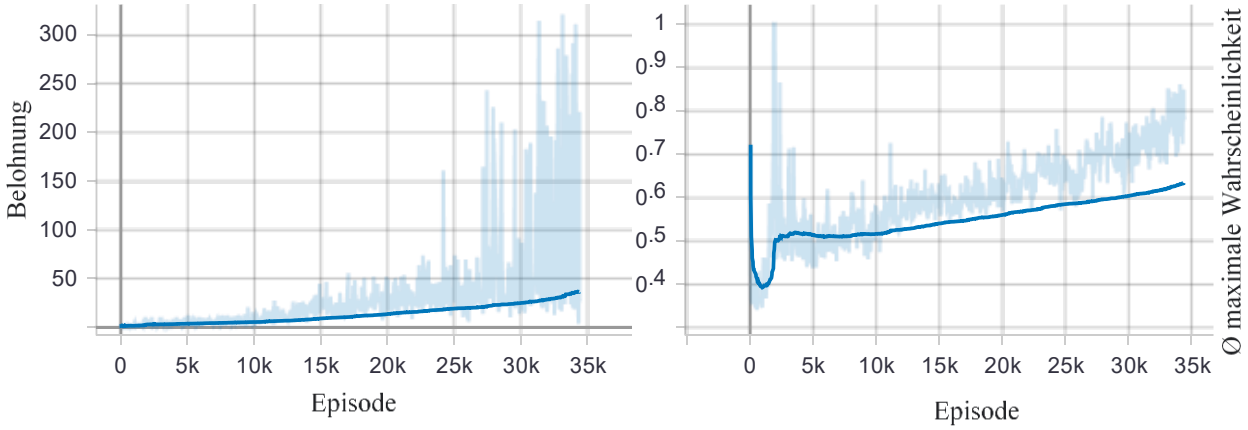

Abbildung 4.2: Trainingsverlauf A3C Agenten (links) der gleitende Mittelwert der Belohnung pro Episode des A3C Agenten, (rechts) der gleitende Mittelwert für die durchschnittliche maximale Wahrscheinlichkeit der möglichen Aktionen pro Episode.

Abbildung 4.2 zeigt links den gleitenden Mittelwert der Belohnung pro Episode und rechts den gleitenden Mittelwert für dir durchschnittliche maximale Wahrscheinlichkeit der möglichen Aktionen pro Episode. Gezeigt ist der beste Versuch aus fünf Trainingsdurchläufen. Wenn man die Belohnungen pro Episode des A3C Agenten betrachtet, fällt auf, dass dieser zu Beginn konstant lernt und stetig mehr Belohnungen erhält. Ab circa 27000 Episoden streuen die Belohnungen stark, sodass der Agent teilweise eine Belohnung von über 200 oder höher erreicht und anschließend eine Belohnung von 10 erzielt. Dies kann zum einen mit der kleinen Modellgröße zusammenhängen. Da es nicht so viele trainierbare Parameter gibt, wird das Training im späteren Verlauf instabil, da bereits kleinere Änderungen an den wenigen Gewichten große Auswirkungen haben können. Zum anderen hängt dies auch damit zusammen, dass falls ein Agent der acht eine hohe Belohnung erzielt, die anderen sieben Agenten immer noch die Gewichte des vorherigen Durchlaufs gespeichert haben und die neu gewonnen Erfahrungen noch nicht mit einbeziehen. Betrachtet man hingegen die durchschnittliche maximale Wahrscheinlichkeit der möglichen Aktionen pro Episode so steigt diese durchgehend konstant und gleichmäßig. Nur zu Beginn schlägt die Kurve aus, was durch die zufällige Initialisierung der Gewichte des Modells zu erklären ist. Da der Agent zu Beginn des Trainings die Umgebung erkundet und viele Aktionen zufällig zu Beginn auswählt, verteilt sich die Wahrscheinlichkeit gleichmäßig auf die drei möglichen Aktionen.

4.2 Probleme

Bei dem Training sind einige Probleme und Hindernisse in den Modellen aufgekommen. Ein Hindernis ist die Vorverarbeitung der Zustände. Ohne die Vorverarbeitung (Kapitel 3.2) sind die Modelle nicht in der Lage in einem absehbaren Zeitraum die Ergebnisse aus der Evaluation (Kapitel 4.3) zu erreichen. In der Vorverarbeitung steckt Domänenwissen, welches bei anderen Problemen nicht immer zur Verfügung steht. Dazu kommt, dass durch das Abschneiden der Ränder und das darauffolgende herunterskalieren des Bildes einige Informationen verloren gehen können.

Ein weiteres Problem ist die Instabilität der fNN. Diese sind für hochdimensionale Probleme, die meist viele mögliche unterschiedliche Zustände haben, gedacht. Hierfür ist oft eine große Menge an Episoden notwendig, um möglichst viele Zustände zu trainieren. Unter diesen Umständen sind kleine Modelle mit wenig zu trainierenden Gewichten sprunghaft in den Ergebnissen. Das Modell des A3C Agenten neigt zu einem solchen verhalten und darüber hinaus dazu die Gewichte zu überanpassen, sodass der Agent immer die gleichen Aktionen nach circa 38000 Episoden in einer Sequenz zu Beginn einer Episode ausführt. Ein Beispiel ist in **Fehler! Verweisquelle konnte nicht gefunden w erden.** zu sehen. Dabei ist zu sehen, dass das Modell zum Ende hin wesentlich schlechtere Ergebnisse erzielt und das Modell für jeden Zustand eine eindeutige Aktion mit einer Wahrscheinlichkeit von nahezu 100% berechnet. Der Agent erreichte zu diesem Zeitpunkt immer die Belohnung 11 pro Episode. Dieses Modell ist somit nicht mehr verwendbar.

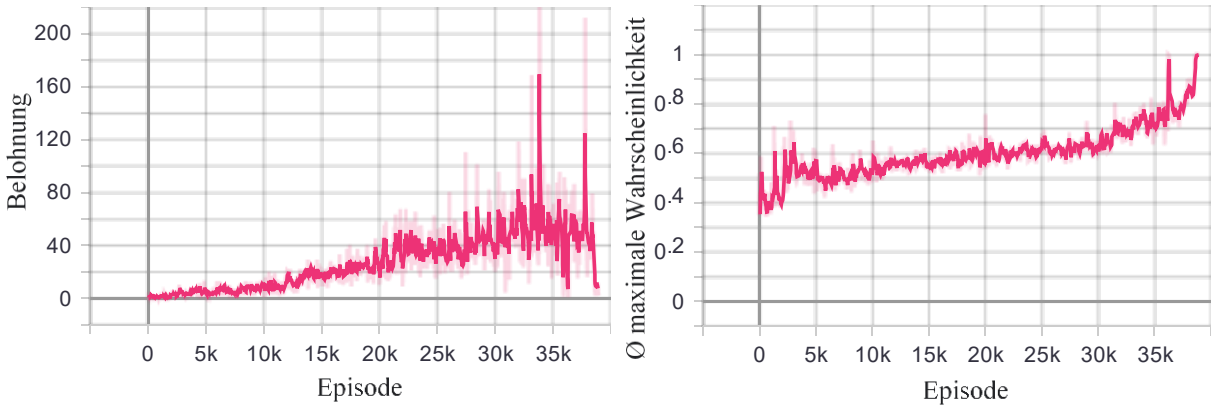

Abbildung 4.3: Überanpassung des A3C Modells. (links) der gleitende Mittelwert der gesamten Belohnung pro Episode, (rechts) der gleitende Mittelwert der maximalen durchschnittlichen Wahrscheinlichkeit pro Episode

Ein weiteres Problem ist die Abhängigkeit von verschiedenen zufällig ausgewählten Parametern, wie zum Beispiel die Gewichte der fNN oder die Auswahl von zufälligen Proben aus dem Erfahrungsspeicher des DQN Modells, auf den Trainingsprozess des Agenten. Gerade zu Beginn des Trainings hat dies Auswirkungen. Hat der DQN Agent anfangs zufällig mehrere gute Episoden gespielt, die anschließend für die Aktualisierung der Gewichte verwendet werden, ist dies für den Lernprozess hilfreich. Der Gegensatz ist eher hinderlich für den Trainingsprozess. Dies gleicht sich im Verlauf des Trainings meist aus, allerdings ist es hinderlich für die Nachstellbarkeit der aufgezeigten Ergebnisse.
Zuletzt ist zu erwähnen, dass die Trainingsdauer für fNN sehr schnell sehr groß wird, was dazu führte, dass mit den zur Verfügung stehenden Ressourcen, das Training des DQN Modells nicht abgeschlossen werden konnte.

4.3 Evaluationsergebnisse

Um die beiden Modelle zu evaluieren, wurden jeweils 1000 Episoden des Spiels gespielt. Dabei wurde die Belohnung pro Episode erfasst und der Mittelwert daraus berechnet. Es wurden die trainierten Modelle verwendet, die in Kapitel 4.1 vorgestellt sind. Zusätzlich zu den beiden Agenten, die in dieser Thesis vorgestellt werden, sind auch der Mittelwert eines menschlichen Agenten, welche von Mnih (2015)[174] stammen, und der Mittelwert eines Agenten, der die Aktionen rein zufällig auswählt, der Übersicht beigefügt. Die Resultate sind in **Fehler! Verweisquelle konnte nicht gefunden werden.** zu sehen. V erglichen werden der Mittelwert der Belohnung pro Episode von 1000 Episoden. Für den DQN und A3C Agenten ist ebenfalls die Standardabweichung (Std.abw.) mit aufgeführt.

Tabelle 4.1: Evaluationsergebnisse

Umgebung	DQN (± Std.abw.)	A3C (± Std.abw.)	Zufällig	Mensch
Breakout	7.5 (± 3.42)	77.1 (± 67.04)	1.3	31.8

Die Implementation des A3C Algorithmus schneidet insgesamt am besten ab. Die Implementation des DQN Algorithmus erreicht im Vergleich dazu wesentlich schlechtere Ergebnisse. Dieser ist besser als ein rein zufälliger Agent aber schlechter als ein menschlicher Agent. Des Weiteren fällt auf, dass das A3C Modell im Verhältnis zum

[174] Vgl. Mnih, V. u. a. (2015), S. 539.

DQN Modell weitaus instabiler ist als das DQN Modell, da die Standardabweichung im Verhältnis weitaus größer ist. Dies liegt vor allem an dem kleineren fNN Modell mit viel weniger trainierbaren Gewichten als beim DQN fNN Modell.
Das Ergebnis des DQN Modells schneidet durch das frühzeitige Beenden des Trainings schlechter ab. Mit mehr Trainingszeit sollte das Ergebnis auch besser sein.

4.4 Bewertung

Ordnet man die Evaluationsergebnisse in den aktuellen Forschungsstand ein, schneiden die vorgestellten Ergebnisse eher schlecht ab. Die originalen Implementationen der genannten Algorithmen schneiden mit einer durchschnittlichen Belohnung von 168 (DQN)[175] und 681 (A3C)[176] wesentlich besser ab. Neuere Algorithmen wie der Actor Critic Kronecker-Factored Trust (ACKTR) Region Algorithmus von Wu (2017) schließt mit einer Belohnung von 735,7 [177] noch einmal besser ab. Die durchgeführten Änderungen am Modell des A3C Agenten und an der Vorverarbeitung der Zustände können den Agenten in der gezeigten Implementation nicht verbessern.

Mit den Ergebnissen der Evaluation können die in Kapitel 1.2 formulierten Forschungsfragen beantwortet werden. Die erste Forschungsfrage, welcher Algorithmus die höhere Punktzahl im Spiel Breakout erzielt, kann man eindeutig mit dem A3C Algorithmus beantworten. Allerdings ist die Antwort kritisch zu betrachten, da in der Implementation des DQN Algorithmus entweder ein Fehler vorhanden ist oder das Training des Modells weitaus mehr Zeit in Anspruch genommen hätte. Daher ist das Ergebnis des DQN Algorithmus nicht aussagekräftig genug, um die Frage klar zu beantworten.
Die zweite Forschungsfrage ist, ob die Algorithmen auch unter anderen Kriterien verglichen werden können. In Bezug auf die Trainingszeit des DQN und A3C Modells kann man eindeutig sagen, dass der A3C Algorithmus schneller ist als der DQN Algorithmus. Dies liegt vor allem an der Aufteilung des Trainingsprozesses auf mehrere parallel arbeitenden Arbeiter. Wohingegen beim DQN Algorithmus nur ein einziger

[175] Vgl. Mnih, V. u. a. (2013), S. 8.
[176] Vgl. Mnih, V. u. a. (2016), S. 1947.
[177] Vgl. Wu, Y. u. a. (2017), S. 6.

Agent das Modell trainiert. In Bezug auf die verwendeten Ressourcen schneidet der DQN Algorithmus mit der in dieser Thesis beschriebenen Implementation etwas schlechter ab als der A3C Algorithmus. Dies liegt hauptsächlich an der Größe des Erfahrungsspeichers und der Größe des fNN. Das fNN des A3C Algorithmus ist kleiner als das fNN des DQN Algorithmus. Der Erfahrungsspeicher des A3C Algorithmus hat nur die Schritte einer Episode gespeichert. Aber auch hier muss die Antwort kritisch betrachtet werden, da mit steigender Anzahl an parallel arbeitenden Agenten die benötigten Ressourcen ebenfalls steigen würden, aber die Trainingszeit verkürzt werden könnte.
Die letzte Forschungsfrage lautet, ob die Ergebnisse der beiden Algorithmen auch auf andere Probleme projiziert werden können. Allgemein kann man die Frage anhand der in dieser Thesis aufgezeigten Ergebnisse nicht beantworten. Gerade die Vorverarbeitung der Zustände besteht komplett aus Domänenwissen aus den ATARI Spielen. Es gibt hundert weitere ATARI Spiele.[178] Für diese Probleme könnten die Ergebnisse ähnlich ausfallen. Für andere Probleme aus der realen Welt beispielsweise sind die Implementationen ungeeignet, da die Zustände und Aufgaben des Agenten dort viel komplexer ausfallen. Die aufgeführten Modelle sind allerdings speziell auf die ATARI Umgebungen zugeschnitten.

5. Limitation

Ein Manko dieser Arbeit ist die Implementation der beiden Algorithmen. Viele Funktionalitäten sind für beide Agenten gleich und könnten wiederverwendet werden. Ein Beispiel dafür wäre die Vorverarbeitung der Zustände. Diese ist derzeit getrennt für jeden Agenten implementiert. Dies könnte man in einer weiteren Klasse zusammenfassen. Dazu kommen die ausgewählten Netzstrukturen der NN. Da zwei verschiedene Modelle verwendet werden, ist die Aussagekraft der Ergebnisse nicht eindeutig definierbar. Ein Vergleich von zwei gleichen Modellen wäre für die Forschungsfragen von Vorteil gewesen.
Das Optimieren der Hyperparameter für beide Algorithmen würde den Rahmen der Arbeit sprengen. Daher wurde darauf verzichtet. Diese bieten allerdings Potenzial die Leistungsfähigkeit der Algorithmen zu verbessern und den Trainingsprozess zu stabilisieren und gegebenenfalls weniger Zeitaufwendig zu machen. Ebenso wären

[178] Vgl. Bellemare, M. G. u. a. (2013), S. 253.

andere Gradientenverfahren denkbar. Die Implementation verwendet nur das RMSProp Verfahren.

Was in dieser Thesis ebenfalls nicht thematisiert wird, sind lange Kurzzeitgedächtnis Netzwerke. Diese können den Trainingsprozess von Neuronalen Netzen im Allgemeinen verbessern und stabilisieren. Diese bestehen aus einem rückgekoppelten NN.[179]

Ebenfalls werden Optimierungen des DQN Algorithmus nicht mitberücksichtigt. Dazu zählen zum einen der doppelte DQN Algorithmus von Hasselt (2015)[180] und zum anderen der priorisierte Erfahrungsspeicher von Schaul (2015) [181] . Der doppelte DQN Algorithmus tauscht den Max-Operator des Ziel Q-Wertes durch die Kombination aus Aktionsauswahl und Aktionsbewertung mithilfe des Zielnetzwerkes aus.[182] Dies hat den Grund, dass der ursprüngliche DQN Algorithmus Zustände zu hoch bewertet. Der doppelte DQN Algorithmus kann diese Überbewertung verringern und somit den Trainingsprozess robuster machen und dadurch bessere Ergebnisse erzielen. [183] Der priorisierte Erfahrungsspeicher sorgt dafür, dass bei der Auswahl der Trainingsbeispiele gewisse Erfahrungen priorisiert werden, sodass entscheidende Erfahrungen öfters trainiert werden und der Trainingsprozess effizienter wird.[184] Der Trainingsprozess kann so um den Faktor zwei beschleunigt werden.[185]

Die implementierte Vorverarbeitung der Zustände basiert auf dem Wissen des Entwicklers. Neuere Ansätze, wie der von Ayberk Aydın, Elif Surer (2020)[186], verwendet ein NN, um die Objekte in dem Zustand zu erkennen. Die Ergebnisse des ersten NN werden anschließend als Eingabe für das NN des RL Algorithmus verwendet.[187] Dies könnte einen RL Algorithmus unabhängiger von dem zu lösenden Problem machen. Es wäre somit auch nicht das Wissen des Entwicklers über das Problem notwendig.

---

[179] Vgl. Mnih, V. u. a. (2016), S. 1933.
[180] van Hasselt, H. u. a. (2015).
[181] Tom Schaul u. a. (2016).
[182] Vgl. van Hasselt, H. u. a. (2015), S. 4.
[183] Vgl. ebd., S. 7.
[184] Vgl. Tom Schaul u. a. (2016), S. 1.
[185] Vgl. ebd., S. 9.
[186] Ayberk Aydın, Elif Surer (2020).
[187] Vgl. ebd., S. 1.

Eine weitere Methode, um hochdimensionale Probleme zu lösen bieten neuroevolutionäre Ansätze. Diese werden im Laufe dieser Arbeit nicht in Betracht gezogen. Lösungsmöglichkeiten bieten Hausknecht (2014)[188] oder Salimans (2017)[189] mit verschiedenen neuroevolutionären Algorithmen. Die Algorithmen bestehen aus Neuronalen Netzen. Diese werden trainiert, indem die Gewichte der Verbindungen zwischen den Neuronen durch Mutation und Kreuzung angepasst werden. Zusätzlich dazu können manche Algorithmen die Topologie des NN anpassen.[190] Ein Vorteile dabei ist, dass kein Gradientenverfahren, welches sehr Rechenaufwendig ist, notwendig ist. Zusätzlich ist die Methode stark parallelisierbar, stabil und besitzt zusätzlich ein verbessertes Erkundungsverhalten.[191] Diese Möglichkeiten bieten eine Alternative, die ähnliche Ergebnisse liefern können.

6. Zusammenfassung und Ausblick

Im Zuge dieser Thesis werden Grundlagen des RL aufgezeigt und erläutert. Anschließend wird ein Experiment aufgebaut. Dafür werden zwei verschiedene RL Algorithmen, DQN und A3C, für das hochdimensionale Problem Breakout implementiert und deren Implementation erläutert. Die beiden Algorithmen werden trainiert und die jeweiligen Trainings- und Evaluationsergebnisse gegenübergestellt und interpretiert. Mit Hilfe der Ergebnisse des Experiments können die am Anfang formulierten Forschungsfragen beantwortet werden. Dabei stellt sich heraus, dass der A3C Algorithmus bessere Ergebnisse, als der DQN Algorithmus erzielt und das in einer geringeren Trainingszeit. Diese Ergebnisse können aber nur durch Vorwissen über die Umgebung erzielt werden.

Da diese Arbeit nur eine Simulation beziehungsweise ein Spiel betrachtet, könnten zukünftige Arbeiten sich damit befassen, die hier gezeigten Modelle auf reale Probleme zu übertragen. Dafür könnten sich alle Probleme eignen, die als MEP modelliert werden können. Besonders könnte man aber Probleme betrachten, für die sehr wenige bis keine oder qualitativ schlechte Test- beziehungsweise Trainingsdaten vorhanden sind, da diese für RL Algorithmen nicht notwendig sind.

---

[188] Vgl. M. Hausknecht u. a. (2014), S. 355.
[189] Vgl. Tim Salimans u. a. , S. 1.
[190] Vgl. M. Hausknecht u. a. (2014), S. 357 f.
[191] Vgl. Tim Salimans u. a. , S. 1 f.

Weitere Möglichkeiten für andere Arbeiten bietet die Vorverarbeitung der Zustände. Die in dieser Thesis gezeigte Vorverarbeitung könnte ersetzt werden durch ein NN welches Objekte in den Zustandsbildern erkennt oder man verringert den betrachteten Bereich eines Zustandes, um die RL Algorithmen mit weniger Dimensionen zu trainieren und somit das Training zu beschleunigen.

Quellenverzeichnis

Abadi, M., et al. (2016): TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. URL: https://arxiv.org/pdf/1603.04467.pdf

Ayberk Aydın, Elif Surer (2020): Using Generative Adversarial Nets on Atari Games for Feature Extraction in Deep Reinforcement Learning 2020. URL: https://arxiv.org/pdf/2004.02762.pdf

Azoulay-Schwartz, R., Kraus, S., Wilkenfeld, J. (2004): Exploitation vs. exploration: choosing a supplier in an environment of incomplete information, in: Decision support systems, 38. Jg., Nr. 1, S. 1–18

Baumgarth, C., Eisend, M., Evanschitzky, H. (2009): Empirische Mastertechniken - Eine anwendungsorientierte Einführung für die Marketing- und Managementforschung, Wiesbaden 2009. ISBN: 978-3-8349-1572-6

Bellemare, M. G., et al. (2013): The Arcade Learning Environment: An Evaluation Platform for General Agents, in: Journal of Artificial Intelligence Research, 47. Jg., S. 253–279. DOI: 10.1613/jair.3912

Bellman, R. (1957): A Markovian decision process, in: Journal of mathematics and mechanics, S. 679–684

Bellman, R., Kalaba, R. E. (1965): Dynamic programming and modern control theory 1965

Berry, D. A., Fristedt, B. (1985): Bandit problems: sequential allocation of experiments (Monographs on statistics and applied probability), in: London: Chapman and Hall, 5. Jg., Nr. 71-87

Brendan O'Donoghue, et al.: Combining policy gradient and Q-learning 2017

Caelen, O., Bontempi, G. (2008): Improving the Exploration Strategy in Bandit Algorithms, in: Maniezzo, Vittorio u.a. (Hrsg.), Learning and Intelligent Optimization, Berlin, Heidelberg 2008, S. 56–68

Chollet, F.; others (2015): Keras. URL: https://keras.io, Abrufdatum: 13.03.2021

D. Silver, et al. (2014): Deterministic Policy Gradient Algorithms, ICML 2014, S. 387–395

Degris, T., Pilarski, P. M., Sutton, R. S. (2012): Model-Free reinforcement learning with continuous action in practice, 2012 American Control Conference (ACC 2012).

Montreal, Quebec, Canada, 27 - 29 June 2012, Piscataway, NJ 2012, S. 2177–2182. DOI: 10.1109/ACC.2012.6315022

Döbel, I., et al. (2018): Maschinelles Lernen-Kompetenzen, Anwendungen und Forschungsbedarf, in: Sankt Augustin: Fraunhofer-Gesellschaft (IAS, IMW, Zentrale)

Donoho, D. L., others (2000): High-dimensional data analysis: The curses and blessings of dimensionality, in: AMS math challenges lecture, 1. Jg., Nr. 2000, S. 1–32

Gatti, C. (2015): Design of Experiments for Reinforcement Learning, Cham 2015. ISBN: 978-3-319-12196-3

Germain, H. (2020a): Actor Optimizer. URL: https://github.com/germain-hug/Deep-RL-Keras/blob/master/A3C/actor.py#L28, Abruf am 18.03.2021

Germain, H. (2020b): Critic Optimizer. URL: https://github.com/germain-hug/Deep-RL-Keras/blob/master/A3C/critic.py#L27, Abruf am 18.03.2021

Glorot, X., Bordes, A., Bengio, Y. (2011): Deep sparse rectifier neural networks, Proceedings of the fourteenth international conference on artificial intelligence and statistics 2011, S. 315–323

Goodfellow, I., Bengio, Y., Courville, A. (2016): Deep learning 2016

Greg Brockman, et al.: OpenAI Gym 2016. URL: https://arxiv.org/pdf/1606.01540.pdf

Hado V. Hasselt (2010): Double Q-learning, in: J. D. Lafferty u.a. (Hrsg.), Advances in Neural Information Processing Systems 23 2010, S. 2613–2621

Hansen, L. Kai, Salamon, P. (1990): Neural network ensembles, in: IEEE transactions on pattern analysis and machine intelligence, 12. Jg., Nr. 10, S. 993–1001

Heess, N., Silver, D., Teh, Y. Whye (2013): Actor-critic reinforcement learning with energy-based policies, European Workshop on Reinforcement Learning 2013, S. 45–58

Hinton, G. (2012): Neural networks for Machine Learning. URL: http://www.cs.toronto.edu/~tijmen/csc321/slides/lecture_slides_lec6.pdf, Abruf am 16.03.2021

Hopfield, J. J. (1982): Neural networks and physical systems with emergent collective computational abilities, in: Proceedings of the national academy of sciences, 79. Jg., Nr. 8, S. 2554–2558

I. Grondman, et al. (2012): A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients, in: IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews), 42. Jg., Nr. 6, S. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595

K. Arulkumaran, et al. (2017): Deep Reinforcement Learning: A Brief Survey, in: IEEE Signal Processing Magazine, 34. Jg., Nr. 6, S. 26–38. DOI: 10.1109/MSP.2017.2743240

Kaelbling, L. Pack, Littman, M., Moore, A. (1996): Reinforcement Learning: A Survey, in: Journal of Artificial Intelligence Research 4, 1996. Jg., Nr. 4, S. 237–283

Kaiser, L., et al. (2019): Model-Based Reinforcement Learning for Atari. URL: https://arxiv.org/pdf/1903.00374v4.pdf

Kim, P. (2017): MATLAB deep learning, Berkeley, CA, Apress. ISBN: 978-1-4842-2844-9

Koutník, J., Schmidhuber, J., Gomez, F. (2014): Evolving deep unsupervised convolutional networks for vision-based reinforcement learning, in: Arnold, Dirk V.; Igel, Christian (Hrsg.), Proceedings and companion publication of the 2014 Genetic and Evolutionary Computation Conference, July 12 - 16, 2014, Vancouver, BC, Canada ; a recombination of the 23rd International Conference on Genetic Algorithms (ICGA) and the 19th Annual Genetic Programming Conference (GP) ; one conference - many mini-conferences ; [and co-located workshops proceedings], New York, NY 2014, S. 541–548. DOI: 10.1145/2576768.2598358

L. Lin (1992): Reinforcement learning for robots using neural networks 1992

Leemon Baird (1995): Residual Algorithms: Reinforcement Learning with Function Approximation, in: Armand Prieditis; Stuart Russell (Hrsg.), Machine Learning Proceedings 1995, San Francisco (CA) 1995, S. 30–37. DOI: 10.1016/B978-1-55860-377-6.50013-X

Levine, S., et al. (2015): End-to-End Training of Deep Visuomotor Policies. URL: https://arxiv.org/pdf/1504.00702v5.pdf

Lillicrap, T. P., et al. (2015): Continuous control with deep reinforcement learning. URL: https://arxiv.org/pdf/1509.02971v6.pdf

Li, Y. (2018): Deep Reinforcement Learning. URL: https://arxiv.org/pdf/1810.06339v1.pdf

M. Hausknecht, et al. (2014): A Neuroevolution Approach to General Atari Game Playing, in: IEEE Transactions on Computational Intelligence and AI in Games, 6. Jg., Nr. 4, S. 355–366. DOI: 10.1109/TCIAIG.2013.2294713

Marlos C. Machado, et al. (2017): Revisiting the Arcade Learning Environment: Evaluation Protocols and Open Problems for General Agents, in: CoRR. URL: https://arxiv.org/pdf/1709.06009v2.pdf

Max Jaderberg, et al. (2016): Reinforcement Learning with Unsupervised Auxiliary Tasks, in: CoRR. URL: https://arxiv.org/pdf/1611.05397.pdf

Melo, F. S. (2001): Convergence of Q-learning: A simple proof, in: Institute Of Systems and Robotics, Tech. Rep, S. 1–4

Mnih, V., et al. (2013): Playing Atari with Deep Reinforcement Learning. URL: http://arxiv.org/pdf/1312.5602v1.pdf

Mnih, V., et al. (2015): Human-level control through deep reinforcement learning, in: Nature, 518. Jg., Nr. 7540, S. 529–533

Mnih, V., et al. (2016): Asynchronous Methods for Deep Reinforcement Learning, in: Maria Florina Balcan; Kilian Q. Weinberger (Hrsg.), Proceedings of The 33rd International Conference on Machine Learning, New York, New York, USA 2016, S. 1928–1937

Ong, H. Yi, Chavez, K., Hong, A. (2015): Distributed Deep Q-Learning. URL: https://arxiv.org/pdf/1508.04186.pdf

OpenAI (2016): Breakout Deterministic-v4. URL: https://github.com/openai/gym/blob/master/gym/envs/__init__.py#L636, Abruf am 09.02.21

OpenAI (2021): Breakout-v0. URL: https://gym.openai.com/envs/Breakout-v0/, Abruf am 09.02.21

R. S. Sutton, A. G. Barto, R. J. Williams (1992): Reinforcement learning is direct adaptive optimal control, in: IEEE Control Systems Magazine, 12. Jg., Nr. 2, S. 19–22

Riedmiller, M. (2005): Neural Fitted Q Iteration - First Experiences with a Data Efficient Neural Reinforcement Learning Method, in: Gama, João u.a. (Hrsg.), Machine Learning: ECML 2005, Berlin, Heidelberg 2005, S. 317–328

Rumelhart, D. E., Hinton, G. E., Williams, R. J. (1986): Learning representations by back-propagating errors, in: Nature, 323. Jg., Nr. 6088, S. 533–536

Rummery, G. A., Niranjan, M. (1994): On-line Q-learning using connectionist systems 1994

S. Gu, et al. (2017): Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates, 2017 IEEE International Conference on Robotics and Automation (ICRA) 2017, S. 3389–3396. DOI: 10.1109/ICRA.2017.7989385

Schulman, J., et al. (2015a): Trust region policy optimization, International conference on machine learning 2015, S. 1889–1897

Schulman, J., et al. (2015b): High-Dimensional Continuous Control Using Generalized Advantage Estimation. URL: https://arxiv.org/pdf/1506.02438v6.pdf

Schulman, J., et al. (2017): Proximal Policy Optimization Algorithms. URL: https://arxiv.org/pdf/1707.06347

Seita, D. (2016): Frame Skipping and Pre-Processing for Deep Q-Networks on Atari 2600 Games. URL: https://danieltakeshi.github.io/2016/11/25/frame-skipping-and-preprocessing-for-deep-q-networks-on-atari-2600-games/, Abruf am 01.03.2021

Sewak, M. (2019): Deep Reinforcement Learning, Singapore, Springer Singapore; Imprint: Springer. ISBN: 978-981-13-8284-0

Sharma, S., Srinivas, A., Ravindran, B. (2017): Learning to Repeat: Fine Grained Action Repetition for Deep Reinforcement Learning. URL: https://arxiv.org/pdf/1702.06054.pdf

Specht, D. F., others (1991): A general regression neural network, in: IEEE transactions on neural networks, 2. Jg., Nr. 6, S. 568–576

Sutton, R. S. (1988): Learning to predict by the methods of temporal differences, in: Machine Learning, 3. Jg., Nr. 1, S. 9–44

Sutton, R. S., Barto, A. (2018): Reinforcement learning - An introduction, Second edition, Cambridge, MA et al. 2018. ISBN: 9780262039246

Szepesvári, C. (2010): Algorithms for Reinforcement Learning, in: Synthesis Lectures on Artificial Intelligence and Machine Learning, 4. Jg., Nr. 1, S. 1–103. DOI: 10.2200/S00268ED1V01Y201005AIM009

Tim Salimans, et al.: Evolution Strategies as a Scalable Alternative to Reinforcement Learning 2017. URL: https://arxiv.org/pdf/1703.03864.pdf

Tokic, M. (2010): Adaptive epsilon-greedy exploration in reinforcement learning based on value differences, Annual Conference on Artificial Intelligence 2010, S. 203–210

Tom Schaul, et al. (2016): Prioritized Experience Replay 2016. URL: https://arxiv.org/pdf/1511.05952.pdf

van Hasselt, H., Guez, A., Silver, D. (2015): Deep Reinforcement Learning with Double Q-learning. URL: https://arxiv.org/pdf/1509.06461.pdf

van Rossum, G., Drake, F. L. (2009): Python 3 Reference Manual, Scotts Valley, CA 2009. ISBN: 1441412697

Wang, Z., et al. (2015): Dueling Network Architectures for Deep Reinforcement Learning. URL: https://arxiv.org/pdf/1511.06581.pdf

Watkins, Christopher John Cornish Hellaby (1989): Learning from delayed rewards

Watkins, C. J. C. H., Dayan, P. (1992): Q-learning, in: Machine Learning, 8. Jg., Nr. 3-4, S. 279–292. DOI: 10.1007/BF00992698

Williams, R. J. (1992): Simple statistical gradient-following algorithms for connectionist reinforcement learning, in: Machine Learning, 8. Jg., Nr. 3-4, S. 229–256. DOI: 10.1007/BF00992696

Wu, Y., et al. (2017): Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation. URL: https://arxiv.org/pdf/1708.05144.pdf

Ehrenwörtliche Erklärung

Hiermit versichere ich, dass die vorliegende Arbeit von mir selbstständig und ohne unerlaubte Hilfe angefertigt worden ist, insbesondere, dass ich alle Stellen, die wörtlich oder annähernd wörtlich aus Veröffentlichungen entnommen sind, durch Zitate als solche gekennzeichnet habe. Ich versichere auch, dass die von mir eingereichte schriftliche Version mit der digitalen Version übereinstimmt. Weiterhin erkläre ich, dass die Arbeit in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde / Prüfungsstelle vorgelegen hat. Ich erkläre mich damit einverstanden/~~nicht einverstanden~~, dass die Arbeit der Öffentlichkeit zugänglich gemacht wird. Ich erkläre mich damit einverstanden, dass die Digitalversion dieser Arbeit zwecks Plagiatsprüfung auf die Server externer Anbieter hochgeladen werden darf. Die Plagiatsprüfung stellt keine Zurverfügungstellung für die Öffentlichkeit dar.

Leubsdorf, 12.04.2021 

(Ort, Datum) (Eigenhändige Unterschrift)