

Inhaltsverzeichnis

Abbildungsverzeichnis	vii
Tabellenverzeichnis	viii
Abkürzungsverzeichnis	ix
Symbolverzeichnis	x
Liste der Algorithmen	xii
1 Einleitung	1
2 Literaturüberblick	3
3 Theoretische Grundlagen	6
3.1 Einführung und Überblick	6
3.2 Markow-Theorie der Entscheidungsprozesse	9
3.2.1 Markow-Prozesse	9
3.2.2 Markow-Belohnungsprozesse	12
3.2.3 Markow-Entscheidungsprozesse	15
3.3 Value-based Reinforcement Learning	19
3.3.1 Dynamic Programming	19
3.3.2 Monte-Carlo-Methoden	20
3.3.3 Temporal-Difference-Methoden	21
3.3.4 Value-based Deep Reinforcement Learning	22
3.3.5 Nachteile von <i>value-based</i> -Algorithmen	25
3.4 Policy-based Reinforcement Learning	25
3.4.1 Stochastische Policy-Gradient-Methoden	25
3.4.2 Monte-Carlo Policy-Gradient	29
3.4.3 Actor-Critic Policy-Gradient	30
3.4.4 Deterministische Policy-Gradient-Methoden	31
3.4.5 Deep Deterministic Policy-Gradient	32
3.4.6 Weitere Policy-Gradient-Algorithmen	34
4 Experiment	35
4.1 Das autonome Fahrzeug	35
4.2 Aufbau des Experiments	36
4.2.1 Die verwendete Simulationsumgebung	36
4.2.2 Das verwendete <i>Reinforcement Learning</i> -Framework	38
4.2.3 Die verwendete Fahrzeugsensorik und -aktorik	41
4.2.4 Die verwendeten Belohnungsfunktionen	44
4.2.5 Die verwendeten <i>Reinforcement Learning</i> -Algorithmen	47
4.2.6 Die verwendeten Performanzmetriken	47

INHALTSVERZEICHNIS

4.2.7	Die Untersuchungsgegenstände des Experimentes	49
4.2.8	Die Implementation und Experimentdurchführung	50
5	Ergebnisse und Interpretation	53
5.1	Ergebnisauswertung DDPG-Agent	53
5.1.1	Training und Test des DDPG-Agenten im <i>Practice</i> -Modus	53
5.1.2	Training des DDPG-Agenten im <i>Quick Race</i> -Modus	58
5.2	Vergleich verschiedener Belohnungsfunktionen	60
5.3	Vergleich DDPG-Agent mit DQN-Agent	62
5.4	Zusammenfassung und Erkenntnisse im Vergleich	64
6	Ausblick	66
6.1	Weiterführende Forschungsfragen	66
6.2	Ethische Aspekte und Sicherheitsfragen	67
7	Fazit	68
Literaturverzeichnis		68
Appendix		76
A	Taxonomien zu autonomen Fahrzeugen	77
B	Literaturvergleich	78
C	Aufbau der neuronalen Netze	79
D	Darstellung des Bremsvorganges	84
E	Testergebnisse DDPG-Agent im <i>Practice</i>-Modus	85
F	Darstellung von Überholmanövern	89
G	Trainingsergebnisse des DDPG-Agenten im <i>Quick Race</i>-Modus	90
H	Vergleich der Belohnungsfunktionen	92
I	Vergleich DQN- mit DDPG-Agent	93
J	Datenträger mit Implementierung	97

Abbildungsverzeichnis

3.1	Interaktionsschema Agent-Umgebung im <i>Reinforcement Learning</i>	7
3.2	Taxonomie <i>Reinforcement Learning</i> -Agenten	8
3.3	Darstellung eines beispielhaften Markow-Prozesses	11
3.4	Darstellung eines beispielhaften Markow-Entscheidungsprozesses	16
4.1	Darstellung der Simulationsumgebung TORCS	37
4.2	Darstellung der TORCS-Rennstrecken	38
4.3	Darstellung der Schnittstelle zu TORCS	39
4.4	Darstellung der Sensorik eines autonomen Fahrzeug	41
4.5	Darstellung der Sensorik des TORCS-Fahrzeuges	43
4.6	Darstellung des in TORCS verwendeten Fahrzeuges	44
4.7	Beispiel einer Performanzmetrik für das Experiment	48
5.1	Trainingsergebnisse DDPG-Agent auf der Strecke CG Speedway	53
5.2	Trainingsergebnisse DDPG-Agent auf der Strecke E-Road	54
5.3	Trainingsergebnisse DDPG-Agent auf der Strecke Forza	55
A.1	Vergleich AD-Taxonomien	77
B.1	Vergleich Publikationen	78
C.1	Darstellung neuronales Netz des DQN-Critic	79
C.2	Darstellung neuronales Netz des DDPG-Actor	81
C.3	Darstellung neuronales Netz des DDPG-Critic	82
D.1	Darstellung des Bremsvorganges in TORCS mit dem DDPG-Agenten	84
E.1	Testergebnisse DDPG-Agent auf der Strecke CG Speedway	86
E.2	Testergebnisse DDPG-Agent auf der Strecke E-Road	87
E.3	Testergebnisse DDPG-Agent auf der Strecke Forza	88
F.1	Darstellung von Überholmanövern im <i>Quick Race</i> -Modus	89
G.1	Trainingsergebnisse DDPG-Agent im <i>Quick Race</i> -Modus	91
H.1	Vergleich der Belohnungsfunktionen mit dem DDPG-Agenten	92
I.1	Vergleich DDPG- und DQN-Agent auf der Strecke CG Speedway	94
I.2	Vergleich DDPG- und DQN-Agent auf der Strecke E-Road	95
I.3	Vergleich DDPG- und DQN-Agent auf der Strecke Forza	96

Tabellenverzeichnis

4.1	Tabelle zur Erläuterung der Fahrzeugsensorik	42
4.2	Tabelle zur Erläuterung der Fahrzeugaktorik	44
4.3	Diskretisierung der Wertebereiche der Fahrzeugaktorik für den DQN-Agenten	51
C.1	Hyperparamtereinstellungen des DQN-Agenten	80
C.2	Hyperparametereinstellungen des DDPG-Agenten	83

Abkürzungsverzeichnis

A2C Advantage Actor-Critic

A3C Asynchronous Advantage Actor-Critic

BASit Bundesanstalt für Straßenwesen

CNN Convolutional Neural Network

DDAC Deep Determinisitic Actor-Critic

DDPG Deep Deterministic Policy Gradient

DQN Deep Q Network

LSTM Long Short-Term Memory Reccurent Neural Network

MC-Methoden Monte-Carlo-Methoden

MDP Markow-Entscheidungsprozess

MP Markow-Prozess

MRP Markow-Belohnungsprozess

MSE Mean Square Error

NHTSA National Highway Traffic Saftey Administration

OU Ornstein-Uhlenbeck-Prozess

PPO Proximal Policy Optimization

SAE Society of Automotive Engineers

TD-Methoden Temporal-Difference-Methoden

TORCS The Open Racing Car Simulator

TRPO Trust Region Policy Optimization

Symbolverzeichnis

Wichtige Symbole in dieser Arbeit ab ihrer erstmaligen Verwendung.

Allgemein

\mathbb{N}	Menge der natürlichen Zahlen $\{1, 2, 3, \dots\}$
\mathbb{N}_0	$\mathbb{N} \cup \{0\}$
\mathbb{R}	Menge der reellen Zahlen
\mathbb{R}^n	Menge der geordneten n -Tupel reeller Zahlen

3.2.1, 3.2.2, 3.2.3

$X \sim p$	Zufallsvariable X mit Wahrscheinlichkeitsfunktion $p(x) = \mathbb{P}(X = x)$
$(X_t)_{t \in I}$	Stochastischer Prozess $(X_t, X_{t+1}, \dots, X_n)$ der Zufallsvariable X
$\mathbb{E}(X)$	Erwartungswert der Zufallsvariable X
\mathcal{S}	Menge aller Zustände $\{s_1, \dots, s_n\}$ einer Umgebung
R_t	Belohnung zum Zeitschritt t
γ	Diskontierungsfaktor für die Summe der Belohnungen
G_t	Der Ertrag als Summe diskontierter Belohnungen
\mathcal{A}	Menge aller Aktionen $\{a_1, \dots, a_m\}$ in einer Umgebung
\mathcal{R}_s^a	Belohnung für Aktion a in Zustand s von einer Umgebung
$\mathcal{P}_{ss'}^a$	Übergangswahrscheinlichkeit von s nach s' für Aktion a von einer Umgebung
$\pi(a s)$	Stochastische <i>Policy</i> -Funktion
π	$ \mathcal{S} \times \mathcal{A} $ - <i>Policy</i> -Matrix
$v_\pi(s)$	Langfristig zu erwartender Ertrag unter <i>Policy</i> π ausgehend von Zustand s
$q_\pi(s, a)$	Langfristig zu erwartender Ertrag unter <i>Policy</i> π ausgehend von Zustand s und Aktion a
$\pi_*(a s) \sim a_*$	Optimale stoch. <i>Policy</i> -Funktion mit Ausgabe der in s optimalen Aktion a_*
π_*	Optimale $ \mathcal{S} \times \mathcal{A} $ - <i>Policy</i> -Matrix
$v_*(s)$	Optimale <i>State-Value</i> -Funktion für Zustand s
$q_*(s, a)$	Optimale <i>Action-Value</i> -Funktion für Zustand s und Aktion a
$\arg \max_x f(x)$	Argument x für den $\max f(x)$

3.3.2

ε	Verfallfaktor $\varepsilon \in [0; 1]$ einer ε - <i>Greedy Policy</i>
α	Schrittgröße $\alpha \in [0; 1]$
$q_k(s, a)$	<i>Action-Value</i> -Funktion für Zustand s und Aktion a nach k Episoden

3.3.3

δ_t	TD-Error im Zeitschritt t
$q_t(s, a)$	<i>Action-Value</i> -Funktion für Zustand s und Aktion a nach t Zeitschritten
$\mu(a s)$	Zu optimierende, stoch. <i>Policy</i> bei <i>off-policy</i> -TD-Algorithmen
$\pi(a^\circ s)$	Stoch. <i>Policy</i> bei <i>off-policy</i> -TD-Algo., die a° für das TD-Target vorgibt

TABELLENVERZEICHNIS

3.3.4

w	Gewichtsvektor $\mathbf{w} = (w_1, \dots, w_n)^T$ von <i>value-based Deep Reinforcement Learning</i> -Algorithmen
$\hat{v}(s, \mathbf{w})$	Parametrierte <i>State-Value</i> -Funktion als Funktionsapproximator
$\hat{q}(s, a, \mathbf{w})$	Parametrierte <i>Action-Value</i> -Funktion als Funktionsapproximator
$E(\mathbf{w})$	Fehlermaß-Funktion eines Gewichtsvektors \mathbf{w}
$\nabla f(\mathbf{w})$	Gradient der Funktion f bzgl. \mathbf{w} , im Kontext etwa $\nabla E(\mathbf{w})$

3.4.1

θ	Gewichtsvektor $\boldsymbol{\theta} = (\theta_1, \dots, \theta_n)^T$ von <i>Policy Gradient Methods</i>
$J(\boldsymbol{\theta})$	Performanzmaß eines Gewichtsvektors
$\Lambda(s)$	$\Lambda(s) := \sum_a \nabla \pi(a s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a);$ Summe des Produktes aus Gradientvektor und Skalierungsfaktor über alle Aktionen
$\eta(s)$	$\eta(s) := \sum_{k=0}^{\infty} \mathbb{P}(s \rightarrow s', k, \pi_{\boldsymbol{\theta}})$ Wahrscheinlichkeit von s nach s' in k Schritten unter $\pi_{\boldsymbol{\theta}}$ zu übergehen, hier als Grenzwert
$\psi(s)$	Proportionalitätskonstante des Gradienten des Performanzmaßes $\nabla J(\boldsymbol{\theta})$
$\exp(x) = e^x$	Natürliche Exponentialfunktion mit $e \approx 2,71828$

3.4.3

$b(s)$	Baseline bei <i>actor-critic</i> -Algo., um Varianz des Skalierungsfaktors zu verringern
$a_{\pi_{\boldsymbol{\theta}}}$	<i>Advantage</i> -Funktion zur Reduzierung der Varianz

3.4.4

$\tau(s, \boldsymbol{\theta}) = a$	Parametrierte, deterministische <i>Policy</i> -Funktion
$\boldsymbol{\theta}', \mathbf{w}'$	Target-Parameter für die Berechnung des TD-Target
\mathcal{D}	<i>Experience Replay Memory</i> des DDPG-Algorithmus

Liste der Algorithmen

1	Deep Q-Learning mit Experience Replay Memory	24
2	Monte-Carlo Policy-Gradient (REINFORCE)	29
3	One-Step Actor-Critic	30
4	Deep Deterministic Policy-Gradient	33
5	Programmaufbau für die OpenAI-gym-Schnittstelle	40
6	Pseudocode für die Berechnung der Belohnungsfunktion	45

1 Einleitung

Die Geschichte des autonomen Fahrens beginnt im Jahr 1478, nicht ganz 400 Jahre vor der Patentierung des ersten Viertaktmotors durch Nikolaus August Otto. Da vor dem 18. Jahrhundert wenig über thermodynamische Zusammenhänge bekannt war [Kie18, S. 11 f.], nutzte das autonome Fahrzeug des Jahres 1478 ein Federwerk als Antrieb, konnte aber, durch eine ausgefeilte Mechanik, im Vorhinein festgelegte Fahrmanöver während der Fahrt selbstständig ausführen. Genau genommen hat das hier beschriebene Fahrzeug seinerzeit nur als Entwurf auf dem Konzeptpapier seines Erfinders, Leonardo Da Vinci, existiert. Erste dokumentierte Experimente zur Fahrautomation von modernen Fahrzeugen fanden im Jahr 1926 statt, als die Houdina Radio Control Company das funkgesteuerte Fahrzeug „Linrican Wonder“ fahrerlos durch die verkehrsreichen Straßenschluchten New Yorks manövrierte. [Bim15]

Getragen von den technologischen Fortschritten der Folgejahrzehnte, wie der stetigen Miniaturisierung in der Mikroprozessortechnik und der Verbesserung von Algorithmen zur Sensordatenauswertung, fahren autonome Fahrzeuge, teilweise ohne Sicherheitsfahrer, von Waymo, Uber und etablierten Automobilherstellern heutzutage auf öffentlichen Straßen. Es wird erwartet, dass bis zum Jahr 2035 erste vollautonome Fahrzeuge der Öffentlichkeit angeboten und verkauft werden. [Bim15]

Einen großen Anteil an den jüngeren Fortschritten hat das moderne Maschinelle Lernen (engl. *machine learning*), das einem autonomen Fahrzeug erlaubt, seine Umgebung verlässlich wahrzunehmen und vorherzusagen. So kann ein autonomes Fahrzeug beispielsweise erkennen, wie die Fahrbahnmarkierungen verlaufen, welches Verkehrssignal eine Ampelanlage gerade anzeigt und wie der Pfad eines Fußgängers auf dem Fußgängerüberweg vermutlich verlaufen wird. In Kapitel 2 ist ein Überblick über die Forschungsergebnisse von *Machine Learning*-Algorithmen für autonome Fahrzeuge gegeben und die, im hiesigen Kontext, relevantesten Ideen werden dargestellt. In die Kategorie der *Machine Learning*-Algorithmen fallen auch Algorithmen, bei denen ein Agent durch die Interaktion mit seiner Umgebung die Erreichung eines vorher festgelegten Ziels anstrebt. Diese Unterkategorie von *Machine Learning*-Algorithmen werden *Reinforcement Learning*-Algorithmen (dt. Bestärkendes Lernen) genannt. *Reinforcement Learning*-Algorithmen sind, im Kontext des autonomen Fahrens, der Untersuchungsgegenstand der vorliegenden Arbeit.

Wie Sutton und Barto in [SB17, S. 11 ff.] darstellen, haben *Reinforcement Learning*-Algorithmen eine bis in die 1950er-Jahre zurückreichende Historie und erfahren seit ihrer Kombination mit *Deep Learning*-Algorithmen durch Mnih et al. in [MKS⁺13] besonders viel Aufmerksamkeit innerhalb der Forschung zum Maschinellen Lernen. Im *Reinforcement Learning* befindet sich ein Agent in einer Umgebung, kann Aktionen innerhalb dieser Umgebung ausführen und erhält eine positive, neutrale oder negative Belohnung von der Umgebung für seine Aktionen. Diese Interaktion von Agent und Umgebung kann über viele Zeitschritte andauern. Ziel des Agenten ist das Erlernen einer Strategie (engl. *policy*), die die Summe der Belohnungen aus den einzelnen Zeitschritten maximiert. Für das Erlernen einer solchen Strategie nutzt der Agent einen *Reinforcement Learning*-Agenten-Algorithmus, der algorithmisch vielfältig ausgestaltet werden kann. Die *Reinforcement Learning*-Agenten-Algorithmen lassen sich grob in zwei Kategorien einteilen: *Value-based*-Agenten-Algorithmen leiten aus einer *Action-Value*-Funktion indirekt eine Strategie ab, während *policy-based*- und *actor-critic*-

Agenten-Algorithmen die Strategie direkt bestimmen. Ein bekannter Vertreter der *value-based*-Agenten-Algorithmen ist der *Deep Q-Network*-Algorithmus (DQN) aus [MKS⁺13]. Ein bekannter Vertreter von *actor-critic*-Agenten-Algorithmen ist der *Deep Deterministic Policy-Gradient*-Algorithmus (DDPG) aus [LHP⁺16]. In der vorliegenden Arbeit werden diese beiden Agenten-Algorithmen genutzt, um einem Agenten in der Rennsimulation *The Open Racing Car Simulator* (TORCS), eine Strategie für die Quer- und Längsführung eines Fahrzeugs beizubringen. Der Zustand der Umgebung wird dabei durch die Sensorik des Fahrzeugs erfasst und der Agenten-Algorithmus muss eine belohnungsreiche Abbildung der Sensorwerte auf die Fahrdynamiken Lenkung, Beschleunigung und Bremse erlernen. Genau diese Abbildung ist dann die Strategie des Agenten und kann mittels Performanzmetriken bewertet werden.

Der Agent erhält zwar eine Belohnung von der Umgebung, jedoch gibt die Umgebung nicht direkt vor, wie die Belohnung zu berechnen ist. So sind verschiedene Berechnungsweisen der Belohnung denkbar: Soll ein Agent möglichst schnell fahren, kann ausschließlich eine hohe Geschwindigkeit belohnt werden. Soll ein Agent schnell und möglichst fahrbahnmittig fahren, kann zusätzlich die Abweichung von der Fahrbahnmitte bestraft werden. Wie genau die Belohnung berechnet ist, wird in der Belohnungsfunktion festgelegt. Die Ausgestaltung der Belohnungsfunktion ist in der Literatur zum autonomen Fahren mit *Reinforcement Learning*-Algorithmen häufiger Untersuchungsgegenstand und wird *Reward Shaping* genannt. Somit ist nicht nur die Auswahl des Agenten-Algorithmus im *Reinforcement Learning* bedeutend, sondern auch die Ausgestaltung der Belohnungsfunktion der Umgebung.

Ziel der vorliegenden Arbeit ist deshalb (1.) die Auswertung von Performanzmetriken eines DDPG-Agenten auf verschiedenen Rennstrecken der Simulationsumgebung TORCS, (2.) die Untersuchung der Auswirkung verschiedener Belohnungsfunktionen auf den DDPG-Agenten und (3.) der Vergleich eines DDPG-Agenten mit einem DQN-Agenten hinsichtlich ihrer Performanz in TORCS. Hierfür wird ein Experiment zur Datenerhebung durchgeführt. Das dabei praktisch angewandte theoretische Hintergrundwissen über die Agenten-Algorithmen, die verwendeten Performanzmetriken, die für das Experiment gewählten Untersuchungsgegenstände und die Interpretation der Experimentergebnisse lassen die methodische Vorgehensweise für eine Evaluation von *Reinforcement Learning*-Algorithmen für autonome Fahrzeuge in Simulationsumgebungen erkennen.

Der Aufbau der vorliegenden Arbeit spiegelt diese methodische Vorgehensweise wider. In Kapitel 2 wird zunächst ein Überblick über relevante Forschungsergebnisse zum autonomen Fahren, mittels Methoden des Maschinellen Lernens, gegeben. Dies gibt einen ersten Einblick in die Problemstellungen und Lösungsverfahren des autonomen Fahrens. Kapitel 3 gibt eine Einführung in die theoretischen Hintergründe des *Reinforcement Learning*. In Kapitel 4 wird der Aufbau des Experiments beschrieben und es werden die Performanzmetriken, mit denen die Güte der Strategie eines *Reinforcement Learning*-Agenten gemessen wird, hergeleitet und genannt. Die resultierenden Daten des Experiments werden in Kapitel 5 ausgewertet und interpretiert. Abschließend wird ein Ausblick auf weiterführende Forschungsfragen gegeben und es wird ein Fazit über die Erkenntnisse dieser Arbeit gezogen.

2 Literaturüberblick

In den vergangenen Jahren konnte das Forschungsfeld des autonomen Fahrens, u.a. angetrieben durch neue Erkenntnisse im Bereich *Deep Learning*, erhebliche Forschungsfortschritte aufweisen. [STK18] Ziel dieses Kapitels ist die Zusammenfassung und Einordnung der für diese Arbeit relevanten wissenschaftlichen Literatur zum autonomen Fahren, um den derzeitigen Forschungsstand aufzuzeigen.

Chen et al. geben in [CSKX15] eine Taxonomie bestehender Verfahren zur softwareseitigen Umsetzung autonomer Fahrzeuge an, bestehend aus dem *Mediated Perception*-Ansatz und dem *Behavior Reflex*-Ansatz. Jaritz et al. ergänzen in [JdT⁺18] überdies noch *Reinforcement Learning* als weiteren Ansatz.

Beim *Mediated Perception*-Ansatz wird die komplexe Aufgabe des autonomen Fahrens in mehrere Unteraufgaben aufgeteilt, typischerweise in Wahrnehmungs-, Pfadplanungs- und Regelungsaufgaben. Dieser Ansatz wird in den Publikationen [EAPY17] und [JdT⁺18] als gängigstes Umsetzungsverfahren für autonomes Fahren bezeichnet und wird in Thrun et al. [TMD⁺06] und Montemerlo et al. [MBB⁺08] für autonome Versuchsfahrzeuge, im Rahmen der *DARPA Urban Challenge*, praktisch eingesetzt. Software-Frameworks zum autonomen Fahren, wie Appollo von Baidu oder DriveWorks von NVIDIA, nutzen diesen Ansatz ebenfalls. [FZL⁺18] [ZHBH17]

Dabei hat insbesondere die Aufgabe zur Wahrnehmung (engl. *perception*) der Fahrzeugumgebung von jüngeren Fortschritten im Bereich des Maschinellen Sehens mit *Deep Learning* stark profitiert. Hierzu gehört etwa die Erkennung, Lokalisierung und Bewegungsvorhersage von Verkehrsteilnehmern [RDGF15], [AGR⁺16], Fahrbahnen [GKB⁺16] oder Verkehrszeichen [SSSI11] mittels Lidar, Radar oder Bildkamera. In [HWT⁺15] zeigen die Autoren etwa, dass ein *Convolutional Neural Network* (CNN), trainiert mit mehreren tausend konnotierten Bildern von US-Autobahnen, Fahrzeuge und Fahrbahnmarkierungen mit einer (Berechnungs-)Frequenz von 44 Hz erkennen und im Bild lokalisieren kann.

Die Pfadplanung des *Mediated Perception*-Ansatzes hat zum Ziel, aus der wahrgenommenen Umgebung des Fahrzeuges eine Abfolge von Fahrdynamiken hinsichtlich der Lenkung, der Beschleunigung und der Bremsung zu modellieren, sodass das autonome Fahrzeug erfolgreich manövrieren kann. Die Regelungsaufgabe ist dann dafür zuständig, die auftretenden Abweichungen zwischen der Führungsgröße des geplanten Pfades bzw. der Trajektorie und der Regelgröße des tatsächlich ausgeführten Pfades bzw. der Trajektorie zu minimieren. Eine Übersicht über Verfahren und Algorithmen für Pfadplanungs- und Regelungsaufgaben geben Paden et al. in [PCY⁺16].

El Sallab et al. [EAPY17] und Chen et al. [CSKX15] führen auch Kritik am *Mediated Perception*-Ansatz an. Zum einen führt die Einteilung in die Unteraufgaben Wahrnehmung, Planung und Regelung zu einer Verkomplizierung der eigentlich einfacheren Gesamtaufgabe des autonomen Fahrens, da nun in jeder Unteraufgabe offene Forschungs- und Entwicklungsfragen zu klären sind. Zum anderen weist eine Modularisierung in isolierte Unteraufgaben wenig Koheranz hinsichtlich des gemeinsamen Ziels, dem autonomen Fahren, auf.

Hervorgehend aus dieser Kritik ist der Ende-Zu-Ende-Ansatz für das autonome Fahren, unter dem der *Behavioral Reflex*-Ansatz und auch der *Reinforcement Learning*-Ansatz fallen. In beiden Fällen werden Sensordaten direkt auf Fahrdynamiken wie Fahrzeuglenkung, Beschleunigung und Bremsung abgebildet, sodass es keiner Modularisierung mehr bedarf.

Pomerleau [Pom89] zeigte bereits 1988 eine Umsetzung des *Behavioral Reflex*-Ansatzes, bei dem Bilddaten auf Lenkwinkel in einem *Supervised Learning*-Szenario abgebildet werden. Nach 40 Epochen Training auf 1.200 simulierten Straßenbildern erreicht das dreischichtige *Fully-connected Feedforward*-Netz von Pomerleau eine Genauigkeit von 90% auf Testdaten.

Lecun et al. [LMB⁺05] bauten auf dieser Arbeit auf und nutzten 2005 erstmals ein CNN mit drei *Convolutional*-, zwei *Pooling*- und einer *Fully-connected Feedforward*-Schicht, um einem 50 cm großen Roboterfahrzeug autonomes Fahren, auf unwegsamen Gelände, beizubringen. Eine autonome Fahrt hielt im Durchschnitt 20 m an, ehe der Roboter mit einem Objekt kollidierte. Wie in [Pom89], werden auch in [LMB⁺05] Bilddaten auf Lenkradwinkel, im Sinne des *Behavioral Reflex*-Ansatzes, abgebildet.

Bojarski et al. [BTD⁺16] entwickeln in der Folge ein Hardware-Software-System, NVIDIA DRIVE PX genannt, das drei Kameras, die an einem 2016 Lincoln MKZ und einem 2013 Ford Focus angebracht sind, mit dem CAN-Bus eines Fahrzeugs, zur Ansteuerung der Lenkung, verbindet. Anders als in [LMB⁺05], besteht das CNN aus fünf *Convolutional*- und drei *Fully-connected Feedforward*-Schichten, wurde mit echten Bilddaten antrainiert und im Straßenverkehr eingesetzt. PilotNet von NVIDIA [BYC⁺17] basiert auf dieser Arbeit. Chi und Mu [CM17] gehen noch einen algorithmischen Schritt weiter als [BTD⁺16] und versuchen auch Bildsequenzinformationen, mit *Long Short-Term Memory Recurrent Neural Networks* (LSTM RNN), in den Lenkentscheidungsprozess einfließen zu lassen.

Die hier betrachteten *Behavioral Reflex*-Ansätze haben gemein, dass sie ein aufgezeichnetes Fahrverhalten imitieren, somit eine große, heterogene und konnotierte Anzahl an Trainingsdaten benötigen sowie schlecht mit neuen, unbekannten Fahrsituationen umgehen können. Ein *Reinforcement Learning*-Ansatz zum autonomen Fahren verspricht einige dieser Problematiken zu umgehen, da *Reinforcement Learning*-Agenten, durch zielgerichtete *Trial-and-Error*-Interaktion mit ihrer Umgebung, eine Strategie erlernen können, die keiner *Ground Truth* bedarf.

Wie in Kapitel 1 angesprochen, ist eine Möglichkeit Algorithmen des *Reinforcement Learning* zu klassifizieren, die Einteilung dieser in *value-based* und *policy-based*- bzw. *actor-critic*-Algorithmen. Erstere leiten die Strategie eines *Reinforcement Learning*-Agenten in einer Umgebung indirekt aus einer *Action-Value*-Funktion ab. Letztere können die Strategie eines Agenten direkt ermitteln und werden zumeist in Szenarien eingesetzt, in denen die Aktionen des Agenten in der Umgebung, wie zum Beispiel die Beschleunigung oder der Lenkradwinkel, durch reelle Zahlen ausgedrückt werden. Eine solche Klassifikation von *Reinforcement Learning*-Algorithmen scheint im Kontext des autonomen Fahrens sinnvoll, da der Aktionenraum beim autonomen Fahren zumeist kontinuierlicher Natur ist. In Kapitel 3 werden überdies noch weitere geläufige Taxonomien für das *Reinforcement Learning* genannt. Insbesondere wird in Kapitel 3 dann auf die Literatur, auf derer die in dieser Arbeit implementierten Algorithmen beruhen, spezifisch eingegangen.

Mnih et al. zeigen in [MKS⁺13] erstmals, dass ein *Deep Q-Network*-Agent mit *Convolutional*-Schichten in der Lage ist, aus Bilddaten von sieben Atari 2600 Arcade-Spielen eine erfolgreiche Spielstrategie zu entwickeln. Ein DQN fällt in die Kategorie der *value-based*-Algorithmen. Besonders interessant an dieser Publikation ist, dass neuronale Netze als Funktionsapproximatoren genutzt werden sowie ein *Experience Replay Buffer* eingesetzt wird, um die Trainingsdatenkorrelation zu minimieren. So wurde es möglich, einen hochdimensionalen Zustandsraum, wie dem von Bilddaten, effizienter abzubilden und nutzbar zu machen. In [MKS⁺15] wird für 49 Atari-2600 Arcade-Spiele eine Spielstrategie durch jeweils einen DQN-Agent erlernt. Dabei erreichen die Agenten in 29 Atari-2600-Spielen mindestens 75% der Punkte eines menschlichen Vergleichprobanden.

El Sallab et al. vergleichen in [EAPY16] einen DQN-Agenten mit einem *Deep Deterministic Actor-Critic*-Agenten (DDAC) in der Fahrzeugsimulation TORCS. *Actor-critic*-Algorithmen werden zu den *policy-based*-Algorithmen gezählt, da sie zusätzlich zu einer *Policy*-Funktion die *Action-Value*-Funktion lernen. In [EAPY16] soll aus den Sensordaten der Fahrzeugposition, der Geschwindigkeit und des Spurabstandes eine Strategie erlernt

2 Literaturüberblick

werden, die das Fahrzeug durch Lenkung, Beschleunigung und Bremsung auf der Fahrbahn hält. Interessant an dieser Publikation ist die Gegenüberstellung von *Reinforcement Learning*-Agenten mit diskreten Aktionsmöglichkeiten, wie dem DQN-Agenten, mit Agenten mit kontinuierlichen Aktionsmöglichkeiten, wie dem DDAC-Agenten. So konnte in [EAPY16] empirisch belegt werden, dass ein DQN-Agent ein abrupteres Fahrverhalten als ein DDAC-Agent aufweist, was mit der Diskretisierung des Aktionsraumes beim DQN-Agenten erklärt wird. Basierend auf dieser Arbeit stellen El Sallab et al. dann in [EAPY17] ein ganzheitliches *Deep Reinforcement Learning*-Framework für das autonome Fahren vor.

Jaritz et al. nutzen in [JdT⁺18] einen auf der Publikation von Mnih et al. [MBM⁺16] basierenden *Asynchronous Advantage Actor Critic*-Agenten (A3C) mit diskretem Aktionenraum. Als Simulationsumgebung wird dabei das Rennspiel WRC6 genutzt. Anders als El Sallab et al. in [EAPY16] oder Wolf et al. in [WHW⁺17], wurden dem A3C-Agenten auch Bilddaten der Fahrzeugumgebung bereitgestellt. Besonders interessant an dieser Publikation ist, dass Jaritz et al. dediziert auf die Generalisierbarkeit der erlernten Agentenstrategie eingehen. So wird gezeigt, dass der A3C-Agent Erlerntes auch auf neuen, unbekannten Strecken erfolgreich umsetzen kann. In einem weiteren Szenario konnte dieser Agent sogar korrekte Fahrentscheidungen, basierend auf echten Verkehrsbildern als Eingabe, treffen.

Wang et al. beschreiben in [WJW18] die Implementierung eines *Deep Deterministic Policy-Gradient*-Agenten in TORCS. Der DDPG-Algorithmus basiert auf den Publikationen von Silver et al. [SLH⁺14] und Lillicrap et al. [LHP⁺16] und verbindet die Eigenschaften von DQN-Agenten und *actor-critic*-Agenten mit dem in [SLH⁺14] vorgestellten Satz zu *Deterministic Policy Gradients*. Wie in [EAPY17] werden in [WJW18] nur Positions- und Fahrdynamikdaten, aber keine Kameradaten, als Eingabe verwendet. Allerdings werden in [WJW18] die resultierenden Performanzdaten des DDPG-Agenten nicht mit denen anderer Agenten-Algorithmen verglichen, sondern nur hinsichtlich verschiedener Verkehrsszenarien. Kendall et al. nutzen in [KHJ⁺18] dann einen DDPG-Agenten, um die Querführung eines realen Fahrzeuges zu erlernen. Die Hyperparameter des Agenten wurden dabei in einer Softwaresimulation erprobt und der Agent anschließend auf einer realen Straße in einem Renault Twizy angelernt und getestet. Nach eigenen Angaben der Autoren war dies die, zum damaligen Zeitpunkt, erste Anwendung eines *Deep Reinforcement Learning*-Algorithmus für das Erlernen einer Fahrstrategie in einem realen Fahrzeug.

Aus dem Literaturüberblick wird deutlich, dass die obigen Publikationen zumeist bekannte *Reinforcement Learning*-Algorithmen auf die Problemstellung des autonomen Fahrens anwenden, die Experimentergebnisse dieser, hinsichtlich bestimmter Performanzmetriken, miteinander vergleichen oder die Resultate verschiedener Hyperparametereinstellungen betrachten. Diese Arbeit wird die methodischen Erkenntnisse bisheriger Literatur, insbesondere zu den Performanzmetriken, nutzen, um *Reinforcement Learning*-Algorithmen, im Sinne der Zielstellungen von Kapitel 1, zu evaluieren. Überdies werden in Kapitel 5 die gewonnenen Erkenntnisse dieser Arbeit kurz mit denen der hier angeführten Literatur verglichen, um die vorliegende Arbeit in einen größeren Kontext zu überführen.

3 Theoretische Grundlagen

Ziel dieses Kapitels ist eine fundierte Einführung in die theoretischen Grundlagen des *Reinforcement Learning* zu geben. Dafür wird in Abschnitt 3.1 zunächst eine intuitive Einführung in die Thematik und Begrifflichkeiten gegeben, ehe in Abschnitt 3.2 die Problemstellung des *Reinforcement Learning* mit der Markow-Theorie ausführlich beschrieben und formalisiert wird. Die Definitionen und Herleitungen aus Abschnitt 3.2 bilden die Grundlage für die Beschreibung der Algorithmen des *Reinforcement Learning* in Abschnitt 3.3 und Abschnitt 3.4. Alle für den praktischen Teil dieser Arbeit relevanten Algorithmen werden dort dargestellt und erklärt. Abschnitt 3.3 befasst sich mit *value-based* Algorithmen. Diese schaffen die Grundlage für das Verständnis von den in Abschnitt 3.4 vorgestellten *policy-based* und *actor-critic* Algorithmen. *Policy-based* Algorithmen haben die schöne Eigenschaft in reellen Aktionenräumen, d.h. für Aktionen mit realem Wertebereich, anwendbar zu sein. Sie sind damit nützlich für das autonome Fahren.

3.1 Einführung und Überblick

Denkt man über die Natur des Lernens nach, mag ein erster intuitiver Einfall eines Lernansatzes das Lernen durch Interaktion mit unserer Umgebung sein. Ein Kleinkind lernt Gegenstände zu greifen oder sich bei lauten Geräuschen umzusehen, ohne dass es diesem durch einen Erwachsenen explizit beigebracht wurde. Stattdessen verbindet das Kleinkind die getätigten Aktionen mit den Reaktionen der Umgebung, lernt welche Konsequenzen aus welchen Aktionen folgen und welche Aktionen durchzuführen sind, um ein bestimmtes Ziel zu erreichen. [SB17, S. 1]

Reinforcement Learning ist die algorithmische Modellierung dieses interaktiven Lernens. Ein Agent befindet sich dabei innerhalb einer Umgebung und wählt Aktionen in dieser aus. Infolgedessen erfährt der Agent eine positive, neutrale oder negative Belohnung von der Umgebung sowie eine Rückmeldung über den neuen Zustand der Umgebung. Ziel des Agenten ist die Auswahl von Aktionen, die langfristig die Summe der Belohnungen maximieren. [Sil15b, S. 15] [SB17, S. 1, 43]

Formell betrachtet lässt sich die Interaktion zwischen Agent und Umgebung (engl. *environment*) im *Reinforcement Learning* folgendermaßen beschreiben: Zu einem diskreten Zeitschritt $t \in \mathbb{N}_0$ erhält ein Agent eine Repräsentation des Zustandes der Umgebung S_t , die den derzeitigen Zustand der Umgebung beschreibt. Basierend auf Informationen aus S_t , wählt der Agent eine Aktion A_t aus, um den Zustand der Umgebung, in der er sich befindet, zu verändern. Die Umgebung erzeugt in der Folge für den nächsten Zeitschritt $t + 1$ eine Belohnung $R_{t+1} \in \mathbb{R}$ für die Aktion des Agenten, beschreibt ihren veränderten Zustand S_{t+1} und gibt diese beiden Informationen an den Agenten zurück. Die Dynamiken, mit denen die Umgebung Belohnung und Zustand erzeugt, werden durch ein Übergangsmodell (engl. *transition model* oder einfach *model*) der Umgebung bestimmt. Ziel des Agenten ist das Erlernen einer Strategie (engl. *policy*) zur Auswahl von Aktionen, die über viele Zeitschritte hinweg die Summe der einzelnen Belohnungen maximiert. [SB17, S. 37 f.]

Der Informationskreislauf dieser Agent-Umgebung-Interaktion kann als eine Folge von

3 Theoretische Grundlagen

Zuständen, Aktionen und Belohnungen aufgefasst werden, die mit

$$(S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, S_3, \dots) \quad (3.1)$$

beginnt und in Abbildung 3.1 generativ dargestellt ist. Diese Folge stellt genau die Information bereit, die ein *Reinforcement Learning*-Agent zum Lernen einer Strategie nutzen kann. Wie der Agent dies algorithmisch umsetzt, wird in Abschnitt 3.3 und Abschnitt 3.4 erklärt. Eine solche Folge kann endlich sein, falls die Umgebung einen terminierenden Zustand hat, also einen Zustand der als Folgezustand nur noch sich selber hat, bei einer Belohnung von Null. [SB17, S. 45 f.] Beispiel einer endlichen Folge von Zuständen, Aktionen und Belohnungen ist etwa die Spieler-Spielautomat-Interaktion. Eine Zustand-Aktion-Belohnung-Folge kann auch unendlich sein, wie z.B. bei einer Weltraumsonde-Weltraum-Interaktion. [SB17, S. 43] Im Kontext der vorliegenden Arbeit wird eine endliche Folge durch Interaktion eines Fahrzeuges mit seiner Umgebung, wie der Fahrstrecke und anderen Fahrzeugen, erzeugt.

Anders als beim überwachten Lernen (engl. *supervised learning*) braucht es im *Reinforcement Learning* also keine konnotierten Beispieldaten, sondern die Daten ergeben sich aus der Interaktion von Agent und Umgebung. Auch von unüberwachtem Lernen (engl. *unsupervised learning*) unterscheidet sich *Reinforcement Learning*, da vormaliges Ziel die Maximierung der Summe von Belohnungen und nicht die Aufdeckung von Strukturen in den Daten ist. [SB17, S. 2] [Lap18, S. 2 ff.]

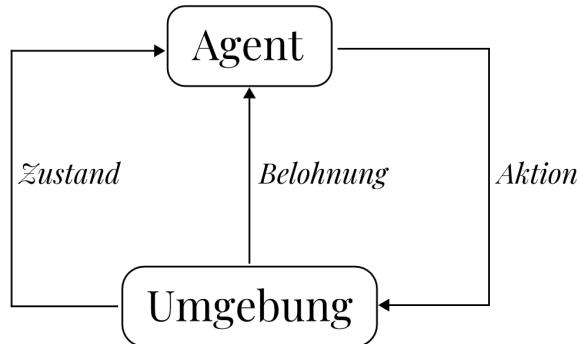


Abbildung 3.1: Dargestellt ist das Interaktionsschema von Agent und Umgebung im *Reinforcement Learning*. Ein Zustand S_t gegeben, führt der Agent eine Aktion A_t aus, um für den nächsten Zeitschritt eine Belohnung R_{t+1} zu erhalten und in einen neuen Zustand S_{t+1} zu übergehen. In Anlehnung an [SB17, Abb. 3.1, S. 38].

Neben einer Darstellung des Interaktionsschemas von Agent und Umgebung, ist auch eine Betrachtung der internen Vorgänge in Agent und Umgebung notwendig, um das *Reinforcement Learning* algorithmisch ganzheitlich beschreiben zu können. Dabei wird vor allem die Funktionsweise eines Agenten betrachtet, da dieser jene Entität ist, die innerhalb einer Umgebung eine Strategie zur Maximierung der Summe der Belohnungen entwickeln muss. Die Dynamiken der Umgebung werden zumeist als gegeben und nicht veränderbar angesehen.

Ein *Reinforcement Learning*-Agenten-Algorithmus kann für die Findung einer *Policy* (dt. Strategie) zur Maximierung der Summe der Belohnungen ein oder mehrere der folgenden, berechenbaren Funktionen nutzen: eine *State-Value*-Funktion, eine *Policy*-Funktion oder ein agenteninternes Modell des Übergangsmodells der Umgebung. Diese Funktionen eines Agenten werden aus den Daten einer oder mehrerer Interaktionsfolgen, wie 3.1, berechnet.

Eine *State-Value*-Funktion beschreibt, wie gut es für einen Agenten ist, in einem bestimmten Zustand innerhalb der Umgebung zu sein. [SB17, S. 46] Während eine unmittelbar hohe Belohnung kurzfristig als für den Agenten erstrebenswert scheint, kann es manchmal besser sein, auf diese zu verzichten und stattdessen eine hohe langfristige Belohnung an-

zustreben. Die langfristig zu erwartende Belohnung eines Zustandes wird durch die *State-Value*-Funktion ausgedrückt. Ein Agent kann so implizit jene Aktionen auswählen, die in Zustände führt, für die die *State-Value*-Funktion einen hohen Wert annimmt. Agenten, die ausschließlich eine *State-Value*-Funktion nutzen, nennt man *value-based*. Neben *State-Value*-Funktionen werden sich ab Unterabschnitt 3.2.3 noch *Action-Value*-Funktionen als wichtig erweisen, die ähnlich zu *State-Value*-Funktionen definiert sind.

Eine *Policy*-Funktion beschreibt, welche Aktion ein Agent in einem bestimmten Zustand ausführen soll. [SB17, S. 5] Ein Agent kann in verschiedenen Interaktionsfolgen verschiedene *Policy*-Funktion haben, d.h. er kann verschiedenen Strategien folgen. Die optimale *Policy*-Funktion ist diejenige *Policy*-Funktion, die langfristig ertragreicher als alle anderen ist. Ziel aller Agenten ist es, diese optimale *Policy*-Funktion für alle Zustände zu finden. Im Gegensatz zu einer *State-Value*-Funktion beschreibt die *Policy*-Funktion nicht die Güte eines Zustandes, sondern welche Aktion in einem Zustand ausgeführt werden soll. *Value-based*-Agenten können implizit auf eine *Policy* schließen, berechnen diese aber nicht explizit. Agenten, die explizit und ausschließlich eine *Policy*-Funktion berechnen, nennt man *policy-based*. Agenten, die eine *Policy*-Funktion und eine *State-Value*-Funktion gleichzeitig berechnen, nennt man *actor-critic*. [Sil15b, S. 34 f.]

Ein internes Modell des Übergangsmodells der Umgebung kann ein Agent berechnen, um antizipatorisch festzustellen, wie sich die Umgebung verhalten wird, wenn der Agent in einem bestimmten Zustand eine bestimmte Aktion ausführt. Dieses interne Modell der Umgebung kann der Agent zur Vorhersage zukünftiger Zustände und zukünftiger Belohnungen nutzen. *Reinforcement Learning*-Agenten, die ein solch internes Modell nutzen, nennt man *model-based*. [SB17, S. 5] Bekannter Vertreter der *model-based Reinforcement Learning*-Agenten ist *AlphaGo*. [SHM⁺16] Einen Einblick in *model-based Reinforcement Learning*-Agenten geben Sutton und Barto in [SB17, S. 131 ff.] und Silver in [Sil15a].

Durch diese verschiedenen berechenbaren Funktionen eines *Reinforcement Learning*-Agenten lässt sich eine Einteilung der Agenten in Kategorien vornehmen. Eine solche ist in Abbildung 3.2 dargestellt. Entsprechend Abbildung 3.2 können *Reinforcement Learning*-

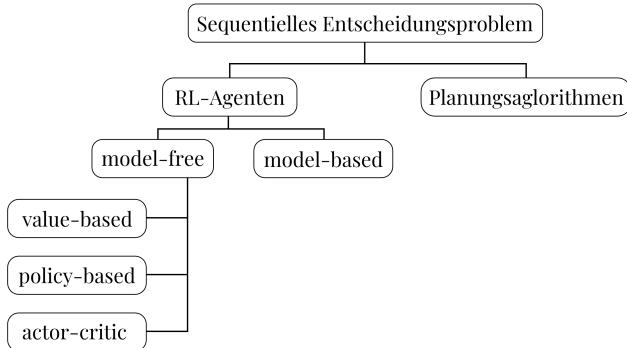


Abbildung 3.2: Dargestellt ist eine Taxonomie von *Reinforcement Learning*-Agenten. Der praktische Teil dieser Arbeit befasst sich, unter anderem, mit dem Vergleich von *value-based* und *actor-critic* Agenten im Kontext des autonomen Fahrens. Bildquelle: Eigens erstellt.

Algorithmen als Lösungen eines sequentiellen Entscheidungsproblems aufgefasst werden, bei denen ein Agent ein Ziel über viele Zeitschritte hinweg verfolgen muss. Planungsalgorithmen können anstatt von *Reinforcement Learning*, genutzt werden, wenn das Übergangsmodell der Umgebung von Anfang an bekannt ist. Die in Unterabschnitt 3.3.1 vorgestellten Algorithmen sind Planungsalgorithmen. Ist das Übergangsmodell nicht bekannt, können *Reinforcement Learning*-Agenten zum Erlernen der ertragsmaximierenden Strategie genutzt werden. In beiden Fällen lässt sich die Umgebung mathematisch mittels Markow-Entscheidungsprozessen (engl. *markov decision processes*, MDPs) modellieren. Die obig dargestellten berechenbaren

Komponenten eines *Reinforcement Learning*-Agenten lassen sich auf einem MDP definieren und durch Algorithmen berechnen. In Unterabschnitt 3.3.2 und Unterabschnitt 3.3.3 wird auf Algorithmen für *value-based Reinforcement Learning*-Agenten eingegangen, in Abschnitt 3.4 auf Algorithmen für *policy-based* und *actor-critic*-Agenten.

3.2 Markow-Theorie der Entscheidungsprozesse

Die Umgebung eines *Reinforcement Learning*-Agenten lässt sich formell mit der Markow-Theorie beschreiben. Auf dieser Basis können Algorithmen, für die in Abschnitt 3.1 dargestellten berechenbaren Funktionen eines Agenten, hergeleitet und entwickelt werden, stets mit der Zielsetzung die Summe der Belohnungen aus den einzelnen Zeitschritten zu maximieren. Für die Modellierung der Umgebung als Makrow-Entscheidungsprozess ist Voraussetzung, dass die Umgebung für einen Agenten ganzheitlich einsehbar (engl. *fully observable*) sowie die Anzahl an Zuständen und Aktionen endlich ist. [Sil15c, S. 3] [SB17, S. 38] Aus dem in Abschnitt 3.1 vorgestellten Interaktionsschema von Agent und Umgebung kann entnommen werden, dass ein formelles Modell Zustände, Belohnungen und Aktionen abbilden können muss. Hierfür wird in Unterabschnitt 3.2.1 zunächst dargestellt, wie Markow-Prozesse die Zustände einer Umgebung abbilden können. In Unterabschnitt 3.2.2 wird mit Markow-Belohnungsprozessen erklärt, wie auch Belohnungen modelliert werden können, ehe in Unterabschnitt 3.2.3 Markow-Entscheidungsprozesse vorgestellt werden, die Zustände, Belohnungen und Aktionen abbilden können.

3.2.1 Markow-Prozesse

Mit Markow-Prozessen (MP) können die Zustände einer Umgebung formell beschrieben werden. Sie beschreiben dabei auch die Dynamiken, mit denen die Umgebung die Zustandsübergänge bestimmt. Ein Agent kann bei einem solchen Markow-Prozess nur die sich verändernden Zustände der Umgebung beobachten und hat somit keinen Einfluss auf den Zustand der Umgebung. [Lap18, S. 12]

Markow-Prozesse werden durch zwei Eigenschaften charakterisiert. Erstens sind Zustandsübergänge nicht deterministisch, sie unterliegen also dem Zufall. Deshalb werden Zustände als Realisation von Zufallsvariablen nach Definition 3.2.1 [MP10, S. 119] modelliert. Zweitens genügt der derzeitige Zustand der Umgebung für die Auswahl des zukünftigen Zustandes, sodass alle vorherigen Zustände nicht beachtet werden müssen. [Lap18, S. 12 ff.]

Definition 3.2.1 (Zufallsvariable)

Eine Funktion $X : \Omega \rightarrow V \subseteq \mathbb{R}$, die jedem Elementarergebnis ω aus der Menge aller möglichen Versuchsausgänge eines Zufallsexperimentes Ω genau ein Realisationswert x einer Wertemenge V zuweist, heißt Zufallsvariable. Ist V endlich oder abzählbar unendlich, heißt X diskrete Zufallsvariable.

Die erstgenannte Eigenschaft von Markow-Prozessen legt dar, dass jeder konkrete Zustand einer Umgebung die Realisation einer diskreten Zufallsvariable ist. Die Menge V enthält also alle Zustände einer Umgebung. Eine diskrete Zufallsvariable X kann, basierend auf einer Wahrscheinlichkeitsfunktion nach Definition 3.2.2 [FKPG03, S. 227], mit einer bestimmten Wahrscheinlichkeit $\mathbb{P}(X = x)$ einen Zustand $x \in V$ annehmen. Ein Zustand kann demnach als Ausgang bzw. Realisation eines Zufallsexperimentes aufgefasst werden, den die Umgebung mit einer bestimmten Wahrscheinlichkeit annimmt.

Definition 3.2.2 (Wahrscheinlichkeitsfunktion)

Sei $X : \Omega \rightarrow V$ eine diskrete Zufallsvariable. Die Wahrscheinlichkeitsfunktion $p : V \rightarrow [0; 1]$ definiert durch

$$\begin{aligned} p(x) &= \mathbb{P}(X = X(\omega)) = \mathbb{P}(X = x) \\ &= \mathbb{P}(\{\omega \in \Omega : X(\omega) = x\}), \end{aligned} \tag{3.2}$$

gibt dann an, mit welcher Wahrscheinlichkeit $p(x)$ die diskrete Zufallsvariable X einen Wert $x \in V$ annimmt.

Die mehrmalige Hintereinanderausführung eines Zufallsexperimentes lässt sich als eine Folge von Zufallsvariablen darstellen. Eine solche Folge wird nach Definition 3.2.3 stochastischer Prozess genannt, da jedes Folgenglied eine Zufallsvariable $X_{t \in \mathbb{N}}(\omega)$ ist. Beispiel eines solchen stochastischen Prozesses ist die n -gliedrige Folge

$$(X_t(\omega), X_{t+1}(\omega), X_{t+2}(\omega), \dots, X_n(\omega)) \quad (3.3)$$

für die man auch kürzer

$$(X_t, X_{t+1}, X_{t+2}, \dots, X_n) \quad (3.4)$$

schreiben kann, da die Ergebnismenge Ω bei allen Zufallsvariablen die selbe ist. Werden die Realisationen der Zufallsvariablen eines stochastischen Prozesses als die Zustände der Umgebung aufgefasst, kann mithilfe von stochastischen Prozessen die zeitliche Abfolge an Zuständen der Umgebung beschrieben werden. Jede Zufallsvariable X_t eines stochastischen Prozesses hat dann eine Wahrscheinlichkeitsfunktion nach Definition 3.2.2, die angibt, mit welcher Wahrscheinlichkeit die Zufallsvariable X_t einen Zustand $x \in V$ annimmt.

Definition 3.2.3 (Stochastischer Prozess)

Sei t Element einer Indexmenge $I \subseteq \mathbb{N}$, ω Elementarergebnis einer Ergebnismenge Ω und $X : \Omega \rightarrow V$ eine diskrete Zufallsvariable. Ein zeitdiskreter stochastischer Prozess wird dann durch eine Folge $f : I \times \Omega \rightarrow V$; $(\omega, t) \mapsto X_t(\omega)$ von Zufallsvariablen beschrieben. Anstatt von f wird $(X_t)_{t \in I}$ geschrieben und $(X_t)_{t \in I}$ in V gesagt, um zu verdeutlichen, dass die Folgenglieder Elemente der Menge V sind.

Häufig hängt bei stochastischen Prozessen die Wahrscheinlichkeit, dass die Umgebung einen bestimmten Zustand annimmt, von den realisierten Zuständen vorheriger Zufallsvariablen ab. Nimmt man beispielsweise den Wettertrend als stochastischen Prozess an, dann mag auch das vorgestrige Wetter noch Einfluss auf das morgige Wetter haben. Diese verketzte Kausalität abzubilden, erschwert die algorithmische Modellierung von stochastischen Prozessen, sodass mit Definition 3.2.4 [Sil15c, S. 4] die Abhängigkeit zukünftiger Zustände nur vom derzeitigen Zustand angenommen wird. Dies ist die zweite wichtige Eigenschaft von Markow-Prozessen.

Definition 3.2.4 (Markow-Eigenschaft)

Ein stochastischer Prozess $(X_t)_{t \in I}$ hat die Markow-Eigenschaft genau dann, wenn für alle $t \in I \subseteq \mathbb{N}$ gilt:

$$\mathbb{P}(X_{t+1} \mid X_t) = \mathbb{P}(X_{t+1} \mid X_1, \dots, X_t). \quad (3.5)$$

Definition 3.2.4 bedient sich der Notation der bedingten Wahrscheinlichkeitsrechnung und lässt sich umgangssprachlich formulieren als: „The future is independent of the past given the present.“ [Sil15c, S. 4] Die Markow-Eigenschaft hat im praktischen Reinforcement Learning konkrete Vorteile wie die Einzigartigkeit und Unterscheidbarkeit von Zuständen [Lap18, S. 12] und kann zur konkreten Formulierung der Wahrscheinlichkeit eines Zustandsübergangs in Definition 3.2.5 [Sil15c, S. 5] genutzt werden.

Definition 3.2.5 (Übergangswahrscheinlichkeit)

Sei $(X_t)_{t \in I \subseteq \mathbb{N}}$ ein stochastischer Prozess mit Markow-Eigenschaft. Die Wahrscheinlichkeit, dass auf das Folgenglied $x \in V$ unmittelbar das Folgenglied $x' \in V$ folgt, wird beschrieben durch

$$\mathcal{P}_{xx'} = \mathbb{P}(X_{t+1} = x' \mid X_t = x). \quad (3.6)$$

Ein Zustand $x \in V$ hat bei $|V| = n$ möglichen Folgezuständen auch n Übergangswahrscheinlichkeiten. Hat eine Umgebung im Reinforcement Learning also n Zustände, kann die Umgebung die Wahrscheinlichkeit des Übergangs von x nach x' durch ein Eintrag in einer quadratischen $n \times n$ Matrix abbilden. Zusammen mit der Modellierung von Zuständen

3 Theoretische Grundlagen

als Realisation von Zufallsvariablen, führt dies zur Beschreibung von Markow-Prozessen in Definition 3.2.6.

Definition 3.2.6 (Markow-Prozess (MP))

Sei $S : \mathcal{S} \rightarrow V \subseteq \mathbb{R}$ eine diskrete Zufallsvariable. Ein Markow-Prozess (auch Markow-Kette) ist ein stochastischer Prozess $(S_t)_{t \in I \subseteq \mathbb{N}}$ mit Markow-Eigenschaft, der durch das Tupel $(\mathcal{S}, \mathcal{P})$ beschrieben werden kann. Dabei ist

- (i) $\mathcal{S} = \{s_1, s_2, \dots, s_n\} \subseteq \mathbb{R}$ eine endliche Menge an Zuständen und
- (ii) \mathcal{P} eine $n \times n$ -Übergangsmatrix über dem reellen Intervall $[0; 1]$, definiert durch

$$\begin{aligned} \mathcal{P} &= \begin{bmatrix} \mathcal{P}_{11} & \cdots & \mathcal{P}_{1n} \\ \vdots & \ddots & \vdots \\ \mathcal{P}_{n1} & \cdots & \mathcal{P}_{nn} \end{bmatrix} \\ &= \begin{bmatrix} \mathbb{P}(S_{t+1} = s_1 \mid S_t = s_1) & \cdots & \mathbb{P}(S_{t+1} = s_n \mid S_t = s_1) \\ \vdots & & \vdots \\ \mathbb{P}(S_{t+1} = s_1 \mid S_t = s_n) & \cdots & \mathbb{P}(S_{t+1} = s_n \mid S_t = s_n) \end{bmatrix}. \end{aligned}$$

Ein Eintrag der Matrix \mathcal{P} gibt an, mit welcher Wahrscheinlichkeit $\mathbb{P}(S_{t+1} = s' \mid S_t = s)$ die Zufallsvariable S_{t+1} den Zustand $s' \in \mathcal{S}$ annimmt, wenn $S_t = s$ gegeben ist. Eine jede Reihe der Matrix \mathcal{P} aus Definition 3.2.6 gibt folglich also eine Wahrscheinlichkeitsfunktion über alle Folgezustände für einen gegebenen Zustand an. In Abbildung 3.3 ist exemplarischer ein terminierender Markow-Prozess dargestellt. Zur Notation in obiger Definition sei angemerkt, dass $S(s_i) = s_i$ mit $1 \leq i \leq n$ gilt, da \mathcal{S} echte Teilmenge der reellen Zahlen ist und somit die Funktion S eine identische Abbildung von \mathbb{R} nach \mathbb{R} ist.

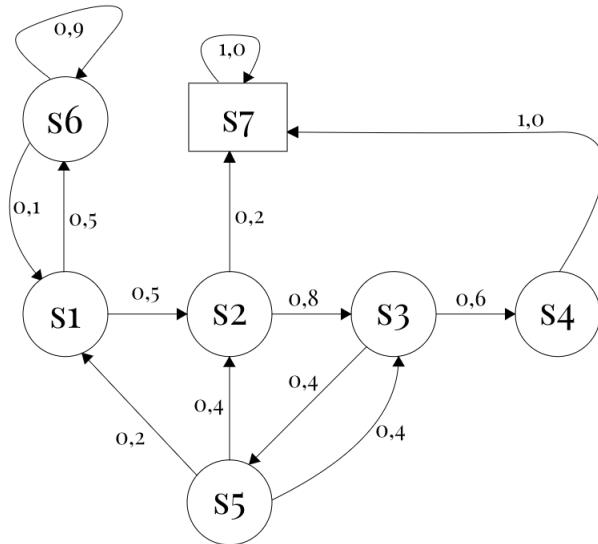


Abbildung 3.3: Beispielhaft dargestellt ist ein Markow-Prozess mit sieben Zuständen. Der Zustand s_7 ist dabei ein terminierender Zustand. Die Wahrscheinlichkeiten für Zustandsübergänge sind auf den Pfeilen angebracht. In Anlehnung an [Sil15c, S. 7].

Ein Agent, der einen Markow-Prozess, wie den in Abbildung 3.3, beobachtet, kann die Folge an Zuständen observieren. Eine Folge an Zuständen wird Episode im *Reinforcement Learning* genannt. Aufgrund der probabilistischen Natur von Markow-Prozessen sind diese

Folgen nicht immer gleich. So könnte ein Agent für den Markow-Prozess in Abbildung 3.3 folgende endliche Folgen an Zuständen beobachten, angenommen s_1 ist immer Startzustand:

$$\begin{aligned} \text{Episode 1: } & (s_1, s_2, s_3, s_4, s_7), \\ \text{Episode 2: } & (s_1, s_2, s_3, s_5, s_2, s_7), \\ \text{Episode 3: } & (s_1, s_6, s_1, s_6, s_1, s_2, s_7) \\ & \vdots \end{aligned}$$

Die hier beispielhaft aufgezeigten Episoden enthalten lediglich die Zustände. Im Kontext des autonomen Fahrens könnte eine Episode als eine Umrundung einer Rennstrecke definiert werden, bei denen die Zustände den Informationen aus der Fahrzeugsensorik entsprechen. Auch andere Definitionen und Terminierungsbedingungen sind möglich. In Unterabschnitt 4.2.4 wird dargestellt, wie eine Episode im Kontext dieser Arbeit definiert ist. Um eine Umgebung im *Reinforcement Learning* ganzheitlich zu beschreiben, ist weiterhin die Modellierung von Belohnungen und Aktionen notwendig.

3.2.2 Markow-Belohnungsprozesse

Belohnungen werden, als Erweiterung von Markow-Prozessen, mittels Markow-Belohnungsprozessen (engl. *markov reward process*, MRP) in Definition 3.2.7 [Sil15c, S. 10] formalisiert. Nach Abschnitt 3.1 ist das Ziel eines *Reinforcement Learning*-Agenten die Maximierung der Summe der Belohnungen aus den einzelnen Zeitschritten. Im beispielhaften Markow-Prozess aus Abbildung 3.3 kann der Agent zwar verschiedene Episoden beobachten, hat aber kein Entscheidungsmaß um quantitativ festzustellen, wie gut oder schlecht eine Episode war. [SB17, S. 42 f.] Gute Episoden sind dabei jene, bei denen die Summe der Belohnungen hoch ist. Diese Summe wird Ertrag genannt. Eine quantitative Beurteilung wird somit durch die Einführung des Ertrages in Definition 3.2.8 [Sil15c, S. 12] möglich. Durch eine *State-Value*-Funktion wird es sogar möglich zu beurteilen, wie gut oder schlecht ein einzelner Zustand ist. Diese Funktion wird im nächsten Unterabschnitt 3.2.3 dann nur noch um Aktionen erweitert, sodass ein Agent aktiv in gute Zustände übergehen kann, um den Ertrag zu maximieren.

Definition 3.2.7 (Markow-Belohnungsprozess (MRP))

Ein Markow-Belohnungsprozess ist ein Markow-Prozess mit Belohnungen, der durch das Tupel $(\mathcal{S}, \mathcal{P}, \mathcal{R}, \gamma)$ beschrieben werden kann. Dabei ist

- (i) $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ eine endliche Menge an Zuständen,
- (ii) \mathcal{P} eine $n \times n$ -Übergangsmatrix über dem reellen Intervall $[0; 1]$,
- (iii) $\mathcal{R} : \mathcal{S} \rightarrow \mathbb{R}$ eine Belohnungsfunktion, die für einen Zustand $s \in \mathcal{S}$ die zu erwartende Belohnung $\mathcal{R}_s := \mathcal{R}(s) = \mathbb{E}(R_{t+1} \mid S_t = s)$ bestimmt, und
- (iv) $\gamma \in [0; 1]$ ein Diskontierungsfaktor.

Die Belohnungsfunktion \mathcal{R} eines MRPs gibt an, wie viel Belohnung ein Agent für einen bestimmten Zustand von der Umgebung erwartet. Die Belohnungsfunktion beruht auf einer Erwartung, da die konkreten Belohnungswerte für einen probabilistischen Zustandsübergang gelten. Entsprechend würden die Pfeile in Abbildung 3.3 bei MRPs um skalare Belohnungswerte ergänzt werden. Die mit den Wahrscheinlichkeiten gewichtete Summe der Belohnungswerte entsprechen dann \mathcal{R}_s . Somit gilt, dass wenn sich ein Agent zum Zeitpunkt t in einem Zustand s befindet, der Agent zum Zeitpunkt $t + 1$, also beim Übergang in einen Folgezustand s' , die Belohnung R_{t+1} erhält. Belohnungen einer Episode können dann als eine Folge $(R_t)_{t \in \mathbb{N}} = (R_1, R_2, \dots)$ in \mathbb{R} beschrieben werden. Die Summe dieser Folge wird mit Definition 3.2.8 Ertrag genannt.

Definition 3.2.8 (Ertrag)

Der Ertrag G_t (engl. return) ist die Summe diskontierter Belohnungen einer Episode

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

ausgehend vom Zeitschritt $t \in \mathbb{N}_0$.

Die Berechnung des Ertrages G_t enthält einen Diskontierungsfaktor $0 \leq \gamma \leq 1$, da es sich bei dem Ertrag um eine unendliche Reihe handelt. Ist $\gamma = 1$, dann ist der Wert der Reihe $\lim_{n \rightarrow \infty} \sum_{k=0}^n \gamma^k R_{t+k+1}$ stets Unendlich. Nur im Falle immer terminierender Episoden könnte ein Agent den Ertrag dann errechnen. Ist $\gamma < 1$, dann hat der Ertrag auch im Falle unendlich langer Episoden einen endlichen Wert, den ein Agent berechnen kann, um festzustellen, wie gut oder wie schlecht eine Episode war. Nicht nur mathematisch ist der Diskontierungsfaktor γ nützlich, sondern auch um einen Agenten auf die Art der Belohnungen einzustellen. Sind frühe Belohnungen in einer Episode deutlich wichtiger als spätere, dann sollte γ einem Wert nahe Null entsprechen. Dies ist beispielsweise der Fall, wenn die Belohnungen finanziellen Gewinn ausdrücken, denn frühe Belohnungen bringen zusätzliche Zinsen ein. Je näher γ an Eins ist, desto wichtiger sind hingegen auch spätere Belohnungen. [Sil15c, S. 12 f.] [SB17, S. 44] [Lap18, S. 19] Für die, nach Kapitel 1, zu evaluierenden DDPG- und DQN-Agenten nimmt γ beispielsweise den Wert 0,99 an.

Mit dem Ertrag G_t kann festgestellt werden wie hoch die Summe der diskontierten Belohnungen einer Episode ist. Mit der *State-Value*-Funktion nach Definition 3.2.9 [Sil15c, S. 14] kann festgestellt werden, wie hoch der langfristig zu erwartende Ertrag ist, ausgehend von einem bestimmten Zustand. [Sil15c, S. 14] [Lap18, S. 17] Aufgrund der probabilistischen Natur der Zustandsübergänge müssen die Episoden, die stets mit einem bestimmten Zustand beginnen, nicht immer gleich sein. Deshalb wird der zu erwartende Ertrag eines bestimmten Zustandes als Erwartungswert der bedingten (Wahrscheinlichkeits-)Dichtefunktion über die Wahrscheinlichkeiten der Erträge für einen Zustand aufgefasst. Der Ertrag G_t kann somit mathematisch als stetige Zufallsvariable behandelt werden.

Definition 3.2.9 (*State-Value*-Funktion für MRPs)

Die *State-Value*-Funktion $v(S_t = s)$ für MRPs gibt für alle Zustände $s \in \mathcal{S}$ den von einem Zustand ausgehenden, langfristig zu erwartenden Ertrag an. Sei der Ertrag $G_t : \mathbb{R} \rightarrow \mathbb{R}$; $G_t(x) = x$ eine stetige Zufallsvariable mit einer bedingten Dichtefunktion $f_{G_t} : \mathbb{R} \times \mathbb{R} \rightarrow [0; 1]$; $(x, s) \mapsto \frac{f(x, s)}{f_S(s)}$. Die *State-Value*-Funktion ist dann definiert durch

$$\begin{aligned} v(S_t = s) &= \mathbb{E}(G_t \mid S_t = s) \\ &= \int_{-\infty}^{+\infty} xf_{G_t}(x, s)dx. \end{aligned}$$

Statt $v(S_t = s)$ wird verkürzend häufig $v(s)$ geschrieben.

Wenn ein Agent nun durch die *State-Value*-Funktion weiß, was der langfristig zu erwartende Ertrag jedes Zustandes einer Umgebung ist, dann kann ein Agent daraus ableiten, in welchen Zustand er übergehen sollte, um den Ertrag einer Episode zu maximieren. Dabei müsste er immer nur in jenen Folgezustand übergehen, der den höchsten langfristig zu erwartenden Ertrag hat. In einem MRP nach Definition 3.2.7 gibt es allerdings keine Aktionen mit denen ein Agent aktiv in Folgezustände wechseln könnte. Aktionen werden in Unterabschnitt 3.2.3 eingeführt.

Offen ist bisher die Frage, wie ein Agent die *State-Value*-Funktion für die Zustände berechnen kann. Eine Antwort ist, dass der Agent die *State-Value*-Funktion aufgrund von Erfahrung schätzen kann. Genauer gesagt kann der Agent die bedingte Dichtefunktion aus Definition 3.2.9 durch Erfahrung schätzen. Die bedingte Dichtefunktion gibt dabei an, mit

welcher Wahrscheinlichkeit welcher Ertrag ausgehend von einem bestimmten Zustand realisiert wird. Wenn der Agent eine Folge an Zuständen und Belohnungen beobachtet, kann er sich für jeden Zustand die nachfolgenden Belohnungen merken, daraus den Ertrag einer Episode berechnen und so über mehrere Episoden hinweg die Wahrscheinlichkeiten der Dichtefunktion für das Auftreten eines Ertrages iterativ anpassen. Geht die Anzahl der Episoden gegen unendlich, stimmen die geschätzten Wahrscheinlichkeiten der Dichtefunktion mit den wahren Wahrscheinlichkeiten überein und der langfristig zu erwartende Ertrag eines Zustandes $v(s)$ ist gefunden. Auch intuitiv lässt sich herleiten, dass je mehr Episoden und daraus folgende Erträge ein Agent beobachtet, desto besser kann er abschätzen, wie gut ein Zustand ist. [SB17, S. 47] Auf diese schätzende Art die *State-Value*-Funktion bzw. die im nächsten Unterabschnitt einzuführende *Action-Value*-Funktion zu bestimmen, wird in Unterabschnitt 3.3.2 und Unterabschnitt 3.3.3 näher eingegangen. In Unterabschnitt 3.3.1 wird dargestellt, wie die *State-Value*-Funktion auch alleinig durch eine rekursive Iterationsgleichung bestimmt werden kann. Diese Iterationsgleichung entspringt der Gleichung von Bellman und der langfristige Ertrag $v(s)$ muss diese Gleichung von Bellman erfüllen. [Lap18, S. 102 ff.] Aus dieser Gleichung lässt sich eine iterative Rechenvorschrift für das Finden jenes langfristigen Ertrages eines Zustandes ableiten, der die wahre Dichtefunktion impliziert.

Um die Gleichung von Bellman herzuleiten genügen wenige Definitionen und Sätze. Nach Definition 3.2.9 gilt zunächst

$$v(s) = \mathbb{E}(G_t \mid S_t = s). \quad (3.7)$$

Wird der Ertrag im Erwartungswert auf der rechten Seite der Gleichung ausgeschrieben, gilt nach Definition 3.2.8

$$\begin{aligned} v(s) &= \mathbb{E}(G_t \mid S_t = s) \\ &= \mathbb{E}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s). \end{aligned} \quad (3.8)$$

Durch Ausklammern von γ in Gleichung 3.8 erhält man

$$\begin{aligned} v(s) &= \mathbb{E}(R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots \mid S_t = s) \\ &= \mathbb{E}(R_{t+1} + \gamma(R_{t+2} + \gamma R_{t+3} + \dots) \mid S_t = s) \quad (3.9) \\ &= \mathbb{E}(R_{t+1} + \gamma G_{t+1} \mid S_t = s) \quad (3.10) \end{aligned}$$

Da der Erwartungswert linear bezüglich der Addition ist [FKPG03, S. 282], d.h. es gilt für Zufallsvariablen X und Y , dass $\mathbb{E}(X + Y) = \mathbb{E}(X) + \mathbb{E}(Y)$ ist, lässt sich Gleichung 3.10 umschreiben zu

$$\begin{aligned} v(s) &= \mathbb{E}(R_{t+1} + \gamma G_{t+1} \mid S_t = s) \\ &= \mathbb{E}(R_{t+1} \mid S_t = s) + \gamma \mathbb{E}(G_{t+1} \mid S_t = s). \end{aligned} \quad (3.11)$$

Ziel der nächsten Umformungen ist es, die *State-Value*-Funktion des derzeitigen Zustandes $v(S_t = s)$ mit der *State-Value*-Funktion des zukünftigen Zustandes $v(S_{t+1} = s')$ in Verbindung zu bringen. [SB17, S. 47] Gleichung 3.11 lässt sich dabei mit dem Satz zu iterierten Erwartungswerten umschreiben zu Gleichung 3.12

$$\begin{aligned} v(s) &= \mathbb{E}(R_{t+1} \mid S_t = s) + \gamma \mathbb{E}(G_{t+1} \mid S_t = s) \\ &= \mathbb{E}(R_{t+1} \mid S_t = s) + \gamma \mathbb{E}(\mathbb{E}(G_{t+1} \mid S_{t+1} = s') \mid S_t = s) \end{aligned} \quad (3.12)$$

$$= \mathbb{E}(R_{t+1} \mid S_t = s) + \gamma \mathbb{E}(v(S_{t+1} = s') \mid S_t = s) \quad (3.13)$$

$$= \mathbb{E}(R_{t+1} + \gamma v(S_{t+1} = s') \mid S_t = s). \quad (3.14)$$

Für die Umformung zu Gleichung 3.13 wurde Definition 3.2.9 genutzt. Gleichung 3.14 nutzt wiederum die Linearität der Addition des Erwartungswertes und wird als Gleichung von Bellman für MRPs bezeichnet. Sie drückt aus, dass der langfristig zu erwartende Ertrag eines Zustandes nur von der unmittelbaren Belohnung und dem langfristig zu erwartenden Ertrag des Folgezustandes abhängt. Deutlich wird dieser Zusammenhang durch weitere Umformung zu

$$\begin{aligned} v(s) &= \mathbb{E}(R_{t+1} + \gamma v(S_{t+1} = s') \mid S_t = s) \\ &= \mathbb{E}(R_{t+1} \mid S_t = s) + \gamma \mathbb{E}(v(S_{t+1} = s') \mid S_t = s) \\ &= \mathcal{R}_s + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'} v(s'). \end{aligned} \quad (3.15)$$

Um auf Gleichung 3.15 zu kommen, kann die Definitionen 3.2.7 und die Erklärung, dass der Erwartungswert von $v(S_{t+1} = s')$ der nach den Übergangswahrscheinlichkeiten gewichtete Mittelwert aller $v(s')$ ist, genutzt werden. Wird Gleichung 3.15 noch um Aktionen erweitert, ergeben sich die zentralsten und wichtigsten Gleichungen im gesamten *Reinforcement Learning*: die Bellman-Gleichungen. Sie drücken aus, dass wenn ein Agent den langfristig zu erwartenden Ertrag von einem Zustand wissen will, er nur die Belohnung des Zustandes und den langfristig zu erwartenden Ertrag des nächsten Zustandes zusammenzählen muss.

3.2.3 Markow-Entscheidungsprozesse

Anders als bei Markow-Prozessen oder Markow-Belohnungsprozessen hat ein Agent bei Markow-Entscheidungsprozessen (MDP) nach Definition 3.2.10 [Sil15c, S. 24] durch Aktionen die Möglichkeit, Kontrolle über die Zustandsübergänge der Umgebung auszuüben. Ein Agent kann bei einem MDP mitentscheiden, in welchen Zustand die Umgebung als nächstes übergehen soll. Ziel eines Agenten ist es, mittels Aktionen in Zustände zu übergehen, sodass der Ertrag nach Definition 3.2.8 maximiert wird. Abbildung 3.4 zeigt einen MDP.

Definition 3.2.10 (Markow-Entscheidungsprozess (MDP))

Ein Markow-Entscheidungsprozess ist ein Markow-Belohnungsprozess mit Aktionen, der durch das Tupel $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma)$ beschrieben werden kann. Dabei ist

- (i) $\mathcal{S} = \{s_1, s_2, \dots, s_n\}$ eine endliche Menge an Zuständen,
- (ii) $\mathcal{A} = \{a_1, a_2, \dots, a_m\}$ eine endliche Menge an Aktionen,
- (iii) \mathcal{P} ein $n \times n \times m$ -Übergangstensor über $[0; 1] \subseteq \mathbb{R}$, bei der ein Eintrag

$$\mathcal{P}_{ss'}^a = \mathbb{P}(S_{t+1} = s' \mid S_t = s, A_t = a)$$

die Wahrscheinlichkeit für den Zustandsübergang von s nach s' angibt, wenn der Agent Aktion a ausgewählt hat,

- (iv) $\mathcal{R} : \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$ eine Belohnungsfunktion, die für einen Zustand $s \in \mathcal{S}$ und Aktion $a \in \mathcal{A}$ die zu erwartende Belohnung $\mathcal{R}_s^a := \mathcal{R}(s, a) = \mathbb{E}(R_{t+1} \mid S_t = s, A_t = a)$ bestimmt, und
- (v) $\gamma \in [0; 1]$ ein Diskontierungsfaktor.

Wie ein Agent bestimmt, welche Aktion er zu einem Zeitschritt in einem Zustand durchführt, wird durch die *Policy*-Funktion beschrieben. Diese *Policy*-Funktion gibt an, welche Aktion ein Agent durchführt und kann sowohl deterministisch als auch stochastisch sein. [Lap18, S. 22] [SB17, S. 46] Zunächst wird in Definition 3.2.11 [Sil15c, S. 26] der stochastische Fall betrachtet.

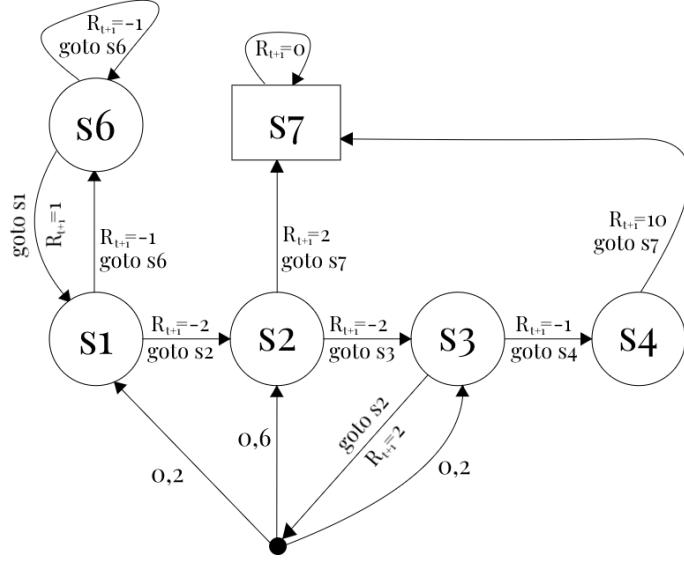


Abbildung 3.4: Dargestellt ist ein MDP mit sieben Zuständen und sechs Aktionen. Aktionen und Belohnungen werden dabei auf die Pfeile geschrieben. Die Aktion *GoTo s₂* hat, exemplarisch, einen nichtdeterministischen Ausgang. Das heißt, obwohl der Agent sich für diese Aktion entschieden hat, führt sie nur mit einer Wahrscheinlichkeit von weniger als Eins in den gewünschten Zustand. Eine *Policy*-Funktion gibt hier an, welche Aktion ein Agent in einem Zustand auswählen soll. Ziel des Agenten ist die Auswahl jener Aktionen, die in ihrer Konsequenz den Ertrag maximieren. Im Anwendungskontext dieser Arbeit betrachtet, entsprechen die Zustände bestimmten Sensorwerten des Fahrzeuges und die Aktionen bestimmten Aktorwerten. In Anlehnung an [Sil15c, S. 25].

Definition 3.2.11 (Stochastische Policy-Funktion)

Die stochastische Policy-Funktion $\pi(a | s)$ gibt die Wahrscheinlichkeit an, dass ein Agent die Aktion $a \in \mathcal{A}$ in Zustand $s \in \mathcal{S}$ zum Zeitschritt $t \in \mathbb{N}_0$ ausführt,

$$\pi(a | s) = \mathbb{P}(A_t = a \mid S_t = s).$$

Demnach kann das gesamte Verhalten eines Agenten, das beschreibt, mit welchen Wahrscheinlichkeiten welche Aktionen in welchen Zuständen ausgeführt werden, durch eine $|\mathcal{S}| \times |\mathcal{A}|$ -Matrix charakterisiert werden. Jeder Eintrag der Matrix ist eine Policy-Funktion $\pi(a_i | s_j)$ mit $1 \leq i \leq m$, $1 \leq j \leq n$. Die $|\mathcal{S}| \times |\mathcal{A}|$ -Matrix wird dann *Policy* π genannt. [SB17, S. 46] Dabei gilt $\sum_{i=1}^{n=|\mathcal{A}|} \pi(a_i | s) = 1, \forall s \in \mathcal{S}$. Die Einträge der Matrix π sind stationär über alle Zeitschritte einer Episode hinweg. Das bedeutet, dass die zugrundeliegenden Policy-Matrizen verschiedener Episoden zwar unterschiedlich sein können, aber sich während einer Episode nicht verändern. [Sil15c, S. 26] Aus diesem Grund muss die *State-Value*-Funktion aus Definition 3.2.9 für MDPs angepasst werden, da immer nur der langfristig zu erwartende Ertrag, ausgehend von einem Zustand, bezogen auf eine, durch *Policy* π beschriebene, Verhaltensweise des Agenten bestimmt werden soll.

Definition 3.2.12 (State-Value-Funktion für Policy π)

Die State-Value-Funktion für eine Policy gibt für alle Zustände $s \in \mathcal{S}$ den von diesem Zustand ausgehenden, langfristig zu erwartenden Ertrag für eine bestimmte Policy π an,

$$v_\pi(s) = \mathbb{E}_\pi(G_t \mid S_t = s).$$

Im Unterschied zu Definition 3.2.9 sagt Definition 3.2.12 also aus, dass nur noch die Erträge von Episoden, denen die gleiche *Policy* des Agenten zugrunde lag, in den Erwartungswert einfließen dürfen. Auch $v_\pi(s)$ muss die Gleichung von Bellman erfüllen. Die Gleichung von Bellman für $v_\pi(s)$ wird in Unterabschnitt 3.3.1 dargestellt. Vorwegnehmend erwähnt werden soll an dieser Stelle, dass für zwei *Policies* π und π' eine Ordnung dieser mittels eines Vergleiches ihrer *State-Value*-Funktionen für alle Zustände induziert werden kann, sodass auch *Policies* miteinander verglichen werden können.

Um neben Zuständen auch Aktionen quantitativ bewerten zu können, wird die *Action-Value*-Funktion nach Definition 3.2.13 [Sil15c, S. 28] vom Agenten genutzt.

Definition 3.2.13 (*Action-Value*-Funktion für *Policy* π)

Die *Action-Value*-Funktion für eine *Policy* gibt für alle Zustände $s \in \mathcal{S}$ und Aktionen $a \in \mathcal{A}$ den von einem Zustand und einer Aktion ausgehenden, langfristig zu erwartenden, Ertrag für eine bestimmte *Policy* π an,

$$q_\pi(s, a) = \mathbb{E}_\pi(G_t \mid S_t = s, A_t = a).$$

Hat ein Agent eine *Policy* π , befindet sich zum Zeitpunkt t in einem Zustand s und führt Aktion a aus, beantwortet die *Action-Value*-Funktion $q_\pi(s, a)$ die Frage nach dem langfristig zu erwartenden Ertrag des Agenten für diesen Zustand und diese Aktion unter dieser *Policy*. Die *Action-Value*-Funktionen lassen sich demnach als eine $|\mathcal{S}| \times |\mathcal{A}|$ -Matrix beschreiben, bei der jeder Eintrag beschreibt, wie langfristig ertragreich es für den Agenten ist, in einem bestimmten Zustand eine bestimmte Aktion durchzuführen. Im Gegensatz dazu lässt sich die *State-Value*-Funktion als $|\mathcal{S}| \times 1$ -Vektor visualisieren, bei dem jeder Eintrag den langfristigen Ertrag eines Zustandes beschreibt. Der Zusammenhang von *State-Value*-Funktion und *Action-Value*-Funktion lässt sich folglich mit

$$v_\pi(s) = \max_a q_\pi(s, a) \quad (3.16)$$

beschreiben. Die Matrix der *Action-Value*-Funktionen wird auch *Q-Table* genannt und ist zentral für Unterabschnitt 3.3.2 und Unterabschnitt 3.3.3. [Rav18, S. 48] Eine *Policy* und die *Action-Value*-Funktionen eines Agenten lassen sich also beide durch $|\mathcal{S}| \times |\mathcal{A}|$ -Matrizen darstellen. Implementierungen von *Reinforcement Learning*-Algorithmen nutzen dann zumeist nur die Matrix der *Action-Value*-Funktionen und schließen über diese auf die *Policy*.

Ziel eines Agenten ist die Maximierung des Ertrages. Im obigen Kontext heißt dies, eine *Policy* zu finden, die möglichst viel Ertrag einbringt. [SB17, S. 50] Hierfür wird zunächst festgehalten, was es bedeutet, dass ein *Policy* besser als eine andere ist. Seien π und π' zwei unterschiedliche stochastische *Policies*. Dann heißt π' besser oder gleich π genau dann, wenn $v_{\pi'}(s) \geq v_\pi(s)$ für alle $s \in \mathcal{S}$ gilt und es wird $\pi' \geq \pi$ geschrieben. Es lässt sich zeigen, dass es mindestens immer eine *Policy* gibt, die besser oder gleich allen anderen *Policies* ist. [Sil15c, S. 40] Diese *Policy* wird die optimale *Policy* π_* genannt. Nutzt ein Agent π_* als *Policy*, wird der Agent für alle $s \in \mathcal{S}$ die optimale *State-Value*-Funktion

$$v_*(s) = \max_\pi v_\pi(s) \quad (3.17)$$

und für alle $s \in \mathcal{S}$ und alle $a \in \mathcal{A}$ die optimale *Action-Value*-Funktion

$$q_*(s, a) = \max_\pi q_\pi(s, a) \quad (3.18)$$

bestimmen können. [SB17, S. 50] Im Umkehrschluss gilt auch, dass wenn die optimale *Action-Value*-Funktion $q_*(s, a)$ gefunden ist, die optimale *Policy* abgeleitet werden kann. [Sil15c, S. 41] Die optimale *Policy*-Funktion $\pi_*(a \mid s)$ hat dabei stets eine Auswahlwahrscheinlichkeit von Eins für jene Aktion a , für die $q_*(s, a)$ für einen Zustand maximal ist. Formell lässt sich dies mit

$$\pi_*(a | s) = \begin{cases} 1 & \text{falls } a = \arg \max_a q_*(s, a) \\ 0 & \text{andernfalls} \end{cases} \quad (3.19)$$

beschreiben. Aus der Fallunterscheidung 3.19 ergibt sich auch, dass die optimale *Policy* deterministisch ist $a = \tau_*(s)$, d.h. für denselben Zustand wählt die optimale *Policy*-Funktion immer genau die selbe Aktion aus. [Sil15f, S. 16] Im Gegensatz dazu lässt Definition 3.2.11 eine Stochastizität der *Policy* zu. Der Übergang von stochastischer zu deterministischer *Policy* erklärt sich durch die gierige (engl. *greedy*) Auswahl von Aktionen in Gleichung 3.19. Wie sich in den Folgeabschnitten herausstellen wird, nutzen alle hier vorgestellten Algorithmen des *Reinforcement Learning* auf die ein oder andere Weise eine stochastische *Policy*, da diese eine Erkundung des Zustandsraumes ermöglicht. Vorwegnehmend sei ausgesagt: *Policies* von *value-based*-Algorithmen werden zum Beinahe-Determinismus $\pi_* \approx \tau_*$ tendieren. *Policies* von *policy-based*-Algorithmen können auch stochastisch bleiben, falls sich die optimale *Policy* als stochastisch erweisen sollte, wobei die Algorithmen aus Unterabschnitt 3.4.4 und Unterabschnitt 3.4.5 eine Ausnahme dessen darstellen. Beide Algorithmenklassen haben gemein, dass sie versuchen π_* zu bestimmen, denn ein MDP gilt als gelöst, wenn π_* gefunden ist. [SB17, S. 53] Der Weg zur Bestimmung der optimalen Strategie π_* eines Agenten in einer Umgebung wird in jedem Fall mit einem iterativen *Try-And-Error*-Verfahren geebnet, bei dem der Agent wiederholend eine *Policy* ausprobiert, aus dem Scheitern lernt und eine neue, verbesserte *Policy* ableitet.

So wie die *State-Value*-Funktionen nach Definitionen 3.2.9 und 3.2.12, muss auch die optimale *State-Value*-Funktion nach Gleichung 3.17 die Gleichung von Bellman erfüllen. [Sil15c, S. 30, 43] In Unterabschnitt 3.2.2 wurde gesagt, dass aus der Gleichung von Bellman eine iterative Rechenvorschrift für das Finden des langfristig zu erwartenden Ertrages jedes Zustandes abgeleitet werden kann. Da nun auch gefordert wird, dass ein Agent Aktionen in sein Handeln einbeziehen kann, wird an dieser Stelle Gleichung 3.14 noch um Aktionen erweitert. Hierfür wird nach Gleichung 3.16 zunächst festgestellt, dass die optimale *State-Value*-Funktion eines Zustandes unter einer *Policy* gleich dem Wert der ertragsmaximierenden Aktion der *Action-Value*-Funktion für einen festen Zustand ist. [SB17, S. 51] Formell bedeutet dies:

$$v_*(s) = \max_a q_*(s, a) \quad (3.20)$$

$$= \max_a \mathbb{E}_{\pi_*}(G_t \mid S_t = s, A_t = a) \quad (3.21)$$

$$= \max_a \mathbb{E}_{\pi_*}(R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a) \quad (3.22)$$

$$= \max_a \mathbb{E}_a(R_{t+1} + \gamma v_*(S_{t+1} = s') \mid S_t = s, A_t = a) \quad (3.23)$$

$$= \max_a \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \right) \quad (3.24)$$

In Gleichung 3.23 wird der rekursive Zusammenhang zum Folgezustand hergeleitet, sodass der Bezug zur optimalen Policy π_* wegfällt. Gleichung 3.24 wird Optimalitätsgleichung von Bellman für $v_*(s)$ genannt. [SB17, S. 51] Analog dazu lässt sich auch eine Optimalitätsgleichung von Bellman für $q_*(s, a)$ herleiten. Anders als Gleichung 3.20 wird für $q_*(s, a)$ in Gleichung 3.25 hier sofort der Bezug zu Belohnungen und dem zu erwartenden Ertrag hergestellt:

$$q_*(s, a) = \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_*(s') \quad (3.25)$$

$$= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a \max_{a'} q_*(s', a'). \quad (3.26)$$

Diese Optimalitätsgleichungen $v_*(s)$ und $q_*(s, a)$ lassen sich sehr intuitiv über *Backup*-Diagramme herleiten, die in [SB17, S. 51] und [Sil15c, S. 43 ff.] ausführlich dargestellt werden. In den nachfolgenden Abschnitten von Kapitel 3 werden Algorithmen dargestellt und hergeleitet, die, basierend auf den Definitionen dieses Unterabschnittes, die optimale *Policy* π_* berechnen können. Mit dieser kann ein Agent den maximalen Ertrag während der Agent-Umgebung-Interaktion einfahren.

3.3 Value-based Reinforcement Learning

In diesem Abschnitt werden Agenten-Algorithmen vorgestellt, die zunächst die *State-Value*-Funktionen bzw. die *Action-Value*-Funktionen der Zustände und Aktionen iterativ berechnen, um aus diesen sich stetig verbessernde *Policies* abzuleiten. Deshalb nennt man diese Algorithmen *value-based*. Unterabschnitt 3.3.1 behandelt Planungsalgorithmen bei denen \mathcal{P} und \mathcal{R} bekannt sein müssen. Unterabschnitt 3.3.2 stellt Algorithmen vor, die in MDPs mit endlichen Episoden eingesetzt werden. Unterabschnitt 3.3.3 erklärt Algorithmen für MDPs mit unendlichen Episoden. Unterabschnitt 3.3.4 zeigt, wie Funktionsapproximatoren, wie neuronale Netze, anstatt von $|\mathcal{S}| \times |\mathcal{A}|$ -Look Up-Matrizen genutzt werden können. Abschließend werden noch Vor-und Nachteile von *value-based*-Algorithmen dargestellt.

3.3.1 Dynamic Programming

Sind die Übergangswahrscheinlichkeiten \mathcal{P} und die Belohnungsfunktion \mathcal{R} der als MDP modellierten Umgebung bekannt, dann kann ein Agent mittels *Dynamic Programming* eine optimale *Policy* π_* berechnen. [SB17, S. 59] *Dynamic Programming* ist dabei eine Klasse von Planungsalgorithmen, die sich zusammensetzen aus der Evaluation einer gegebenen *Policy* (engl. *policy evaluation*) und der anschließenden Verbesserung dieser *Policy* (engl. *policy improvement*). [Sil15f, S. 5, 12] Ziel dieser beiden Schritte ist es in einem iterativen Verfahren, das *Generalized Policy Iteration* genannt wird, auf $v_*(s)$ bzw. $q_*(s, a)$ aller Zustände und Aktionen und damit, mit Gleichung 3.19, auf π_* zu schließen. [SB17, S. 59, 70] In [SB17, S. 80 f.] wird ausführlich auf unterschiedliche Varianten des *Generalized Policy Iteration*-Verfahrens eingegangen. Für die in Unterabschnitt 3.3.1 und Unterabschnitt 3.3.2 dargestellten Algorithmen soll der einfachste Fall des Verfahrens angenommen werden, den charakterisiert, dass im *Policy Evaluation*-Schritt stets $v_\pi(s)$ bzw. $q_\pi(s, a)$ bestimmt wird, bevor im *Policy Improvement*-Schritt eine neue, verbesserte *Policy* π' berechnet wird. Andere Varianten des *Generalized Policy Iteration*-Verfahrens approximieren $v_\pi(s)$ bzw. $q_\pi(s, a)$ nur grob und können damit π_* schneller berechnen. [SB17, S. 80 f.] [Sil15d, S. 21]

Im *Policy Evaluation*-Schritt wird beim *Dynamic Programming* zunächst die *State-Value*-Funktion $v_\pi(s)$ jedes Zustandes unter einer *Policy* π berechnet. Diese wird bis zur Konvergenz hinzu $\lim_{k \rightarrow \infty} v_k(s) = v_\pi(s)$, $k \in \mathbb{N}_0$ iterativ mit

$$v_{k+1}(s) = \sum_{a \in \mathcal{A}} \pi(a | s) \left(\mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_k(s') \right) \quad (3.27)$$

für alle Zustände $s \in \mathcal{S}$ berechnet. Gleichung 3.27 ist die zur Optimalitätsgleichung von Bellman für $v_*(s)$ analoge Gleichung von Bellman für $v_\pi(s)$ in Iterationsform. [Sil15c, S. 33] [SB17, S. 47] Ist $v_\pi(s)$ für eine gegebene *Policy* π gefunden, kann mit

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}_\pi(G_t \mid S_t = s, A_t = a) \\ &= \mathcal{R}_s^a + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}_{ss'}^a v_\pi(s') \end{aligned} \quad (3.28)$$

die *Action-Value*-Funktion bestimmt werden. [Sil15c, S. 32] Gleichung 3.28 ist, analog zu Gleichung 3.26 für $q_*(s, a)$, die Gleichung von Bellmann für $q_\pi(s, a)$. Diese kann dann im *Policy Improvement*-Schritt genutzt werden, um eine neue *Policy* mit

$$\pi'(s) = \arg \max_a q_\pi(s, a) \quad (3.29)$$

für alle Zustände zu finden. Der Satz zum *Policy Improvement* stellt sicher, dass die neue deterministische *Policy* π' besser oder gleich der alten deterministischen *Policy* π ist, nämlich genau dann, wenn für alle $s \in \mathcal{S}$

$$q_\pi(s, \pi'(s)) \geq v_\pi(s) \quad (3.30)$$

gilt. [SB17, S. 63] Das bedeutet, dass die Auswahl von Aktion $a' = \pi'(s)$ in Zustand s unter *Policy* π ein besseres Ergebnis hervorbringt, als Aktion $a = \pi(s)$. Die aus Unterabschnitt 3.2.3 bekannte Ungleichung $v_{\pi'}(s) \geq v_\pi(s), \forall s \in \mathcal{S}$ folgt aus der Ungleichung 3.30. [SB17, S. 62 f.] Der Beweis dieses Satzes ist in [SB17, S. 63] zu finden. In der Folge wird für die neue *Policy* π' im *Policy Evaluation*-Schritt $q_{\pi'}(s, a)$ berechnet und mit $\pi''(s) = \arg \max_a q_{\pi'}(s, a)$ verbessert. Werden *Policy Evaluation*- und *Policy Improvement*-Schritt oft genug nacheinander ausgeführt und wird keine deutliche Verbesserung der alten gegenüber der neuen *Policy* mehr festgestellt, dann sind q_* und π_* gefunden. [Sil15f, S. 12, 17]

Bekannte Algorithmen im *Dynamic Programming*, die diese *Generalized Policy Iteration* umsetzen, sind der *Policy Iteration*-Algorithmus und sein Spezialfall der *Value Iteration*-Algorithmus. Letzterer nutzt die Gleichungen 3.24 und 3.26 zu $v_*(s)$ und $q_*(s, a)$ und eine Variante des *Generalized Policy Iteration*-Verfahrens, bei der nicht $v_\pi(s)$ bestimmt werden muss, sondern auf eine Iteration im *Policy Evaluation*-Schritt sofort der *Policy Improvement*-Schritt folgt. [SB17, S. 65 ff., 81] [Sil15f, S. 21] Hindernis für den praktischen Einsatz dieser Algorithmen ist die häufige Unkenntnis über die Umgebungs dynamiken \mathcal{P} und \mathcal{R} . [Lap18, S. 108] Auch die Annahme, alle Zustände einer Umgebung seien von vornherein bekannt, ist realitätsfern. [Lap18, S. 120] Beispielsweise wird ein Agent zur autonomen Führung eines Fahrzeuges die Zustandsdynamiken der Zustände X -Koordinatenmenge \times Y -Koordinatenmenge \times Fahrspur der Raumumgebung graduell erkunden müssen und nicht von Anfang an wissen. Diesen Problemen wird in den beiden folgenden Unterabschnitten entgegengetreten.

3.3.2 Monte-Carlo-Methoden

Anders als *Dynamic Programming* können Monte-Carlo-Methoden (MC-Methoden) verwendet werden, falls die Übergangswahrscheinlichkeiten \mathcal{P} und die Belohnungsfunktion \mathcal{R} der Umgebung unbekannt sind. [Sil15e, S. 4] [Rav18, S. 71] Mit MC-Methoden kann ein Agent die optimale *Policy* π_* aus der durch Erfahrung geschätzten optimalen *Action-Value*-Funktion $q_*(s, a)$ schließen. Genau wie Algorithmen im *Dynamic Programming*, nutzen MC-Methoden das iterative Verfahren *Generalized Policy Iteration*, um inkrementell auf π_* zu schließen. [SB17, S. 80] Hierfür wird zunächst eine endliche Episode $(S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \dots, R_n, S_n)$ generiert. Die Aktionen werden dabei durch eine stochastische *Policy* $\pi(a | s)$, deren Entwicklung weiter unten erklärt wird, ausgewählt und die Zustände und Belohnungen entspringen den unbekannten Umgebungs dynamiken. Aus dieser Episode wird dann G_t für alle erreichten Zustand-Aktion-Paare berechnet. Hierauf folgt der *Policy Evaluation*-Schritt der *Policy* π mit

$$q_{k+1}(s, a) = q_k(s, a) + \alpha(G_t - q_k(s, a)). \quad (3.31)$$

Dabei ist $\alpha \in [0; 1]$ ein Faktor zur Lernratenanpassung und der Ertrag G_t gibt die Summe diskontierter Belohnungen ausgehend von der Aktion a im Zustand s an. [SB17, S. 97] [Sil15d, S. 16] Folglich korrigiert der Term $G_t - q_k(s, a)$ den Wert von $q_{k+1}(s, a)$ in Richtung des *Target* G_t . Im Unterschied zu Gleichung 3.27 nutzt Gleichung 3.31 weder $\mathcal{P}_{ss'}^a$ noch \mathcal{R}_s^a und ist

damit Unabhängig von der Kenntnis der Umgebungs dynamiken. Eine zu Gleichung 3.31 ähnliche Gleichung existiert auch für die *State-Value*-Funktion, aber diese hat nur theoretischen Wert, da, aufgrund der fehlenden Umgebungs dynamiken bei MC-Methoden, Gleichung 3.28 hier nicht anwendbar ist. [Sil15d, S. 8] Der Zählindex $k \in \mathbb{N}_0$ in Gleichung 3.31 gibt die derzeitige Episode an und im Grenzfall $k \rightarrow \infty$ gilt wieder $\lim_{k \rightarrow \infty} q_k(s, a) = q_\pi(s, a)$.

Da Monte-Carlo-Methoden keine Kenntnis der Umgebungs dynamiken benötigen, sollte eine *Policy* für die Episodengenerierung bei Monte-Carlo-Methoden nicht deterministisch sein, da ansonsten nicht sichergestellt ist, dass alle Zustände und Aktionen erkundet werden. Mit einer stochastischen *Policy* können zufällig Aktionen ausgewählt werden und eine solche wird deshalb von MC-Methoden zu Erkundungszwecken genutzt. Dieses Vorgehen wird ε -*Greedy*-Ansatz für die Erkundung aller Zustände und Aktionen in einem MDP genannt. [Sil15d, S. 11] Eine neue *Policy* π' wird bei MC-Methoden demnach mit

$$\pi'(a | s) = \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon & \text{falls } a = \arg \max_a q_\pi(s, a) \\ \frac{\varepsilon}{m} & \text{andernfalls} \end{cases} \quad (3.32)$$

bei $|\mathcal{A}| = m$ Aktionen und dem abschwellenden Verfallfaktor ε bestimmt. [Sil15d, S. 11] Die Problematik der Erkundung aller Zustände und Aktionen in einem unbekannten MDP ist eine zentrale Problemstellung im *Reinforcement Learning* und wird als *Exploration vs. Exploitation*-Dilemma bezeichnet. [Sil15b, S. 41 f.] [SB17, S. 2] Auch im Fall der Anwendung des ε -*Greedy*-Ansatzes für die Erkundung, stellt ein Satz zum *Greedy Policy Improvement* sicher, dass die neue *Policy* besser als die alte *Policy* ist. [Sil15e, S. 12] Ein ausführlicher Konvergenzbeweis hinzu π_* findet sich in [Sil15f, S. 35 ff.]. Der Prozess der Anpassung der *Action-Value*-Werte im *Policy Evaluation*-Schritt und der Bestimmung einer neuen *Policy* im *Policy Improvement*-Schritt hinzu π_* werden als Training des *Reinforcement Learning*-Agenten bezeichnet.

Aus Gleichung 3.31 ergibt sich, dass MC-Methoden den Ertrag G_t einer Episode nutzen. Dieser kann nur berechnet werden, wenn die Episode bereits terminiert ist. Dies bedeutet folglich, dass MC-Methoden nur in Umgebungen mit endlichen, d.h. immer terminierenden, Episoden eingesetzt werden können. [SB17, S. 75, 97]

3.3.3 Temporal-Difference-Methoden

Temporal-Difference-Methoden (TD-Methoden) können, im Gegensatz zu MC-Methoden, auch in Umgebungen mit unendlichen Episoden eingesetzt werden. [SB17, S. 97] Der Mechanismus, der dies ermöglicht, wird *Bootstrapping* genannt, bestimmt $v_{k+1}(s)$ bzw. $q_{k+1}(s, a)$ aus der unmittelbaren Belohnung zuzüglich $v_k(s')$ bzw. $q_k(s', a')$ und findet schon im *Dynamic Programming* durch die Gleichungen von Bellman in ähnlicher Weise Anwendung. [Sil15d, S. 26 ff.] Anders als Algorithmen des *Dynamic Programming* sind TD-Methoden aber nicht auf die Umgebungs dynamiken \mathcal{P} und \mathcal{R} angewiesen.

Wie *Dynamic Programming* und MC-Methoden, nutzen TD-Methoden ebenfalls das iterative *Generalized Policy Iteration*-Verfahren, um die optimale *Policy* π_* zu bestimmen. Der *Policy-Evaluationsschritt* unterscheidet sich bei TD-Methoden von MC-Methoden, der *Policy Improvement*-Schritt ist mit dem ε -*greedy*-Ansatz bei beiden der selbe. [SB17, S. 105] Der Unterschied von TD-Methoden und MC-Methoden im Evaluationsschritt kann dabei am einfachsten anhand der iterativen Berechnung der *State-Value*-Funktion einer *Policy* π mit folgender Gleichung verdeutlicht werden:

$$v_{t+1}(s) = v_t(s) + \alpha(R_{t+1} + \gamma v_t(s') - v_t(s)), \quad (3.33)$$

wobei der Term $\delta_t = R_{t+1} + \gamma v_t(s') - v_t(s)$ als TD-Error bezeichnet wird und der Term $G_t^{(1)} = R_{t+1} + \gamma v_t(s')$ als TD-Target. [SB17, S. 98] Gleichung 3.33, obgleich es sich hier um die *State-Value*-Funktion handelt, verdeutlicht den Unterschied zu MC-Methoden. TD-Methoden müssen bis zur Anpassung von $v_{t+1}(s)$ nur bis zum nächsten Zeitschritt warten, während

Monte-Carlo-Methoden zur Anpassung meist bis zum Ende der Episode, für die Berechnung von G_t , warten müssen. [SB17, S. 97] Deshalb entspricht der Zählindex $t \in \mathbb{N}_0$ nun dem Zeitschritt. Allgemeiner kann das TD-*Target* als n -stufiger Ertrag $G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n v(S_{t+n} = s^{(n)})$ formuliert werden, wobei für $\lim_{n \rightarrow \infty} G_t^{(n)}$ TD-Methoden zu MC-Methoden werden. TD-Methoden, die $G_t^{(1)}$ nutzen, werden TD(0) genannt. [SB17, S. 98] Dabei stellt sich natürlich die Frage nach dem optimalen n für $G_t^{(n)}$. TD(λ)-Methoden beantworten diese Frage und werden in [Sil15e, S. 34 ff.] und insbesondere in [SB17, S. 115 ff.] ausführlich erklärt.

Wie schon für MC-Methoden geltend, kann die *State-Value*-Funktion in Gleichung 3.33 für den *Policy Evaluation*-Schritt praktisch nicht genutzt werden, da \mathcal{P} und \mathcal{R} unbekannt sind. Aus diesem Grund wird bei TD-Methoden wieder die *Action-Value*-Funktion für den Evaluationsschritt genutzt und iterativ $q_\pi(s, a)$ bestimmt. Eine TD-Methode die dies praktisch umsetzt, wird SARSA genannt und passt im Evaluationsschritt für einen Zustand und eine Aktion die *Action-Value*-Funktion in jedem Zeitschritt der Episode mit

$$q_{t+1}(s, a) = q_t(s, a) + \alpha(R_{t+1} + \gamma q_t(s', a') - q_t(s, a)) \quad (3.34)$$

an. [SB17, S. 105 f.] Das Tupel (s, a, r, s', a') ist namensgebend für den SARSA-Algorithmus und wird vor Berechnung von Gleichung 3.34 durch die Umgebungs dynamiken und die derzeitige ε -greedy *Policy* bestimmt. Ist $q_{t+1}(s, a)$ berechnet, wird im Zeitschritt $t+1$ die neue *Policy* π' mit dem ε -greedy-Ansatz bestimmt. [Sil15d, S. 21] Diese Iterationen werden wieder so lange durchgeführt, bis schlussendlich q_* und damit π_* gefunden sind. Gleichung 3.34 findet Anwendung im SARSA(0)-Algorithmus. Analog zu TD(λ)-Algorithmen gibt es auch SARSA(λ)-Algorithmen, die ausführlich in [SB17, S. 119 ff.] behandelt werden.

SARSA wird als *on-policy*-Algorithmus bezeichnet, da SARSA die selbe *Policy* π evaluier und verbessert, die es auch für die Auswahl der nächsten Aktionen a' nutzt. Im Gegensatz dazu gibt es *off-policy*-Algorithmen, die eine *Policy* π evaluieren und verbessern, aber für die Auswahl der nächsten Aktion a' eine andere *Policy* μ nutzen. [SB17, S. 82] Viele praktische Gründe für die sinnvolle Nutzung von *off-policy*-Algorithmen sind in [Sil15d, S. 31] genannt. Ein Grund kann etwa die Einführung einer neuen Version eines Agenten zur autonomen Führung eines Fahrzeuges mit der *Policy* π sein, der das Verhalten von einem alten, sich bewährten Agenten mit *Policy* μ erlernen oder gar verbessern will. Angemerkt sei aber, dass μ und π durchaus die selbe *Policy* sein dürfen. Bekanntestes Beispiel für einen *off-policy*-TD-Algorithmus ist das Q-Learning. [WD92] Sei $\mu(a | s)$ die *Policy*, die die tatsächliche Aktionen a und a' bestimmt. Sei $\pi(a^\circ | s)$ die *Policy*, mithilfe der das TD-*Target* berechnet werden soll. Dann wird im Evaluationsschritt in jedem Zeitschritt der Episode für einen Zustand s und eine Aktion a die *Action-Value*-Funktion mit

$$q_{t+1}(s, a) = q_t(s, a) + \alpha(R_{t+1} + \gamma \max_{a^\circ} q_t(s', a^\circ) - q_t(s, a)) \quad (3.35)$$

berechnet. [Sil15d, S. 35 ff.] Im Unterschied zu SARSA, wird im Q-Learning bei der Anpassung der *Action-Value*-Funktion für ein (s, a) immer die ertragsmaximierende Aktion a° aller *Action-Value*-Funktionen bei einem festen s' gewählt, unabhängig davon, welche Aktion a° von *Policy* π gewählt wurde. So wie beim SARSA-Algorithmus, folgt im Q-Learning auf die Berechnung von $q_{t+1}(s, a)$ der *Policy Improvement*-Schritt mit dem ε -greedy-Ansatz bezüglich π und wahlweise auch μ . [Sil15d, S. 36]

3.3.4 Value-based Deep Reinforcement Learning

Die in Unterabschnitt 3.3.2 und Unterabschnitt 3.3.3 beschriebenen Algorithmen werden als *Tabular Methods* (dt. Tabellenwert-Methoden) bezeichnet, da zentraler Baustein die $|\mathcal{S}| \times |\mathcal{A}|$ -Matrix der *Action-Value*-Erträge ist. Wird die Anzahl der Zustände $|\mathcal{S}|$ oder der Aktionen $|\mathcal{A}|$ jedoch sehr groß, wird auch diese Matrix sehr groß. Beispiel dafür ist ein Agent zur autonomen Führung eines Fahrzeuges, der den Zustand der Raumumgebung

durch eine Kamera erfährt. Dann stellt jedes Pixel des resultierenden Bildes drei Zustände, im RGB-Farbraum, dar. [Lap18, S. 120] Neben dem hohen Speicherbedarf einer solchen $|\mathcal{S}| \times |\mathcal{A}|$ -Matrix wird es einem Agenten mit MC-Methoden oder TD-Methoden vor allem fast unmöglich sein, alle Zustände zu bewerten, um daraus eine *Policy* abzuleiten. [Lap18, S. 120] Als Lösungsansatz für die Probleme von *Tabular Methods* gehen *Approximate Solution Methods* (dt. approximative Lösungsmethoden) hervor, die sowohl speichereffizienter sind und auch generalisieren können. [SB17, S. 120] Solche *Approximate Solution Methods* nutzen parametrierbare Funktionsapproximatoren wie *Decision Trees*, Regressionsmethoden oder neuronale Netze, anstatt einer $|\mathcal{S}| \times |\mathcal{A}|$ -Matrix, um die *Action-Value*-Erträge zu bestimmen. [SB17, S. 162] Werden mehrschichtige neuronale Netze als Funktionsapproximatoren verwendet, werden die entstehenden Algorithmen dem *Deep Reinforcement Learning* zugeordnet. [MKS⁺13] Die Sekundär- und Tertiärliteratur zu *Deep Learning* weist ausführliche Herleitungen und anschauliche Beschreibungen von mehrschichtigen neuronalen Netzen auf, daher soll in dieser Arbeit auf die Beschreibung der grundlegenden Funktionsweise dieser verzichtet werden und es wird stattdessen auf die einschlägige Literatur wie [GBC16, S. 164 ff.] oder [BL17] verwiesen. Die zugrundeliegenden mathematischen Methoden aus der Linearen Algebra, Analysis und Numerik können etwa [GBC16, S. 27 ff.] entnommen werden.

Anders als *Tabular Methods*, nutzen *Approximate Solution Methods* parametisierte Funktionsapproximatoren $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ bzw. $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$ mit $\mathbf{w} \in \mathbb{R}^d$, $d \in \mathbb{N}$ für alle Zustände und Aktionen im *Policy Evaluation*-Schritt. Der Vektor \mathbf{w} kann dabei der Gewichtsvektor eines Neurons sein. Wird eine Fehlerfunktion $E(\mathbf{w})$ über diesen Gewichtsvektor definiert, dann kann im iterativ-numerischen Gradientenabstiegsverfahren (engl. *gradient descent*) der Gewichtsvektor so lange angepasst werden $E(\mathbf{w}^i) > E(\mathbf{w}^{i+1}) > \dots > E(\mathbf{w}^n)$, bis $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$ für alle Zustände und Aktionen gilt. Falls $d \ll |\mathcal{S}|$ gilt und kein *Overfitting* oder *Underfitting* auftritt, ist der Funktionsapproximator speichereffizienter als *Tabular Methods* und kann generalisieren. [SB17, S. 161] Als Fehlerfunktion soll hier die mittlere quadratische Abweichung (engl. *mean square error*, MSE) $E(\mathbf{w}) := E(\mathbf{w}, T) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ genutzt werden, die typischerweise im Regressionsfall angewandt wird. [BL17, S. 17 f.] Dabei ist T die Menge der Trainingsdaten, aus derer die Zielvariable y entnommen wird.

Da, anders als im klassischen *Supervised Learning*, allerdings keine Informationen über die Zielvariable, hier $y = q_\pi(s, a)$, zur Verfügung stehen, wird für die Fehlerfunktion das *Target* als Zielvariable angenommen. [SB17, S. 161 f.] Werden Funktionsapproximatoren auf MC-Methoden angewandt und *Stochastic Gradient Descent* als Optimierungsverfahren angenommen, ergibt sich so die Fehlerfunktion $E(\mathbf{w}) = \frac{1}{2} (G_t - \hat{q}(s, a, \mathbf{w}))^2$. Für SARSA(0) ergibt sich $E(\mathbf{w}) = \frac{1}{2} (R_{t+1} + \gamma \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}))^2$ als Fehlerfunktion. [Sil15h, S. 24] Intuitiv gesprochen, soll eine Gewichtsvektorkomponente w_j , $j \in \mathbb{N}$ stark angepasst werden, falls sie einen hohen Anteil am Fehlermaß $E(\mathbf{w})$ hat. Ein hoher Anteil eines Gewichtes w_j am Fehlermaß liegt genau dann vor, falls $\frac{\partial E}{\partial w_j}$, also die Steigung von E bzgl. w_j , groß ist. Die Gewichte werden so lange angepasst, bis $\varphi \geq |\nabla E(\mathbf{w}^t)| - |\nabla E(\mathbf{w}^{t-1})|$ eintritt, also keine φ -signifikante Gewichtsverbesserung mehr wahrnehmbar ist. Dann ist \hat{q}_* gefunden, aus der π_* abgeleitet werden kann. Häufig wird im praktischen *Deep Learning* statt einer φ -Signifikanz auch *Early Stopping* verwendet. [GBC16, S. 241 ff.] Mit folgender Regel werden die Gewichte angepasst:

$$\mathbf{w}^{t+1} = \mathbf{w}^t + \Delta \mathbf{w} \quad (3.36)$$

$$= \mathbf{w}^t + (-\alpha \nabla E(\mathbf{w}^t)) \quad (3.37)$$

$$= \mathbf{w}^t - \alpha \begin{pmatrix} \frac{\partial E}{\partial w_1^t} \\ \vdots \\ \frac{\partial E}{\partial w_n^t} \end{pmatrix}. \quad (3.38)$$

Im *Policy Improvement*-Schritt kann wieder der ε -greedy-Ansatz angewandt werden. Da $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$ aber nun ein neuronales Netz und kein *Lookup Table* mehr ist, wird in

der Praxis zumeist ein Vektor

$$\hat{\mathbf{y}} = \begin{pmatrix} \hat{q}(s, a_1, \mathbf{w}) \\ \vdots \\ \hat{q}(s, a_m, \mathbf{w}) \end{pmatrix} \quad (3.39)$$

durch das neuronale Netz approximiert. [Sil15h, S. 7] [Lap18, S. 138, 144] In [MKS⁺13] besteht die Elemente dieses Vektors etwa aus den Ertragswerten der Aktionstasten des Atari-2600-Controllers. Mit einem auf Gleichung 3.39 angepassten ε -greedy-Ansatz können dann neue, verbesserte *Policies* bestimmt werden. Dabei wird Aktion $a = \arg \max_a \hat{q}(s, \mathbf{a}, \mathbf{w})$ mit der Wahrscheinlichkeit $\frac{\varepsilon}{m} + 1 - \varepsilon$ ausgewählt.

Deep Q-Networks aus [MKS⁺13] sind bekannte *value-based, off-policy Deep Reinforcement Learning*-Agenten-Algorithmen, die einen *Mini-Batch Gradient Descent* als Optimierungsverfahren nutzen, ein *Experience Replay Memory* zur Trainingsdatendekorrelation haben und, namensgebend, als *Target* der Fehlerfunktion $\mathbf{y} = R_{t+1} + \gamma \max_{a^\circ} q(s', \mathbf{a}_t^\circ, \mathbf{v})$ nutzen. [MKS⁺13] [Sil15h, S. 38 f.] Hinter diesen Konzepten verbirgt sich eine simple Idee: Sammele (s, a, r, s') -Tupel im *Replay Memory*, ziehe einige dieser Tupel zufällig aus dem *Replay Memory*, nutze diese zufälligen Tupel, um den Fehler zu bestimmen und passe die Gewichte \mathbf{w} und \mathbf{v} mithilfe des *Backpropagation*-Algorithmus an. Eigentlich ist hier die Verwendung zweier Parametersätze, \mathbf{w} und \mathbf{v} , für die Approximation der *Action-Value*-Funktionen $\hat{q}(s, \mathbf{a}, \mathbf{w})$ und $\hat{q}(s', \mathbf{a}^\circ, \mathbf{v})$. Grund dafür ist die Instabilität der Optimierung bei Verwendung der selben Parametervektoren und die *off-policy*-Eigenschaft des DQN-Algorithmus. In Algorithmus 1 wird der DQN-Algorithmus als Pseudocode dargestellt. In Kapitel 4 wird auf die spezifischen Implementierungsdetails, wie die Hyperparametereinstellungen, eingegangen und es wird der Aufbau der neuronalen Netze dargestellt.

Algorithmus 1 : Deep Q-Learning mit Experience Replay Memory

```

Result : a nearly optimal policy  $\pi$ 
Initialize replay memory  $\mathcal{D}$  that has a certain length ;
Initialize parameter  $\mathbf{w}$  ;
Initialize parameter  $\mathbf{v}$  that calculate TD-Target by  $\mathbf{v} \leftarrow \mathbf{w}$ ;
for episode=1 to  $M$  do
    Observe initial state  $s_0$  from environment ;
    for  $t=1$  to  $T$  do
        Select a random action with probability  $\epsilon/m$  otherwise
        select action  $a_t = \arg \max_a \hat{q}(s, \mathbf{a}, \mathbf{w}^t)$  ;
        Observe reward  $r_t$  and next state  $s_{t+1}$  from environment ;
        Store  $(s_t, a_t, r_t, s_{t+1})$  tupel in  $\mathcal{D}$  ;
        Sample random batch from  $\mathcal{D}$  ;
         $\mathbf{y} \leftarrow r_t + \gamma \max_{a^\circ} \hat{q}(s_{t+1}, \mathbf{a}^\circ, \mathbf{v})$  ;
         $\hat{\mathbf{y}} \leftarrow \hat{q}(s_t, \mathbf{a}, \mathbf{w}^t)$  ;
         $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t - \alpha \nabla \frac{1}{2} (\mathbf{y} - \hat{\mathbf{y}})^2$  ;
        Every  $c$  time steps, set  $\mathbf{v} \leftarrow \mathbf{w}^t$  ;
    end
end

```

Viele Verbesserungen des DQN-Algorithmus wurden seither vorgeschlagen und publiziert. [Lap18, S. 155 ff.] [vGS15] [WSH⁺15] [SQAS15] Ein Ziel dieser Arbeit ist der Vergleich eines DQN-Agenten mit einem DDPG-Agenten im Kontext des autonomen Fahrens. Der DDPG-Algorithmus wird in Unterabschnitt 3.4.5 erklärt und zählt zu den *actor-critic*-Agenten-Algorithmen. Wie genau sich der DQN-Algorithmus vom DDPG-Algorithmus unterscheidet, wird aus Abschnitt 3.4 deutlich. Dafür hilft es zunächst die Nachteile von *value-based*-Algorithmen zu kennen, die im nachfolgenden Unterabschnitt erklärt werden.

3.3.5 Nachteile von *value-based*-Algorithmen

Alle bisher dargestellten Algorithmen bauen auf dem *Generalized Policy Iteration*-Ansatz auf und sind damit gut verständlich und einheitlich umsetzbar. Nachteil dieser Einfachheit ist allerdings, dass die bisherigen Algorithmen zunächst eine *Action-Value*-Funktion berechnen müssen, um auf eine verbesserte *Policy* zu schließen. Ohne die *Action-Value*-Funktion würde eine verbesserte *Policy* gar nicht existieren. [SB17, S. 265] Hauptziel von *Reinforcement Learning*-Algorithmen ist es nach Unterabschnitt 3.2.3 jedoch, die optimale *Policy* π_* zu bestimmen, sodass ein Agent den Ertrag maximieren kann. Deshalb interessieren einen Agenten in erster Linie *Policies* und *State-Value*-Funktionen und *Action-Value*-Funktionen sind nur Mittel zum Zweck. [Lap18, S. 242] [SB17, S. 265 f.]

Wird ein ε -*greedy*-Ansatz auf Gleichung 3.39 zum *Policy Improvement* angewandt, können die Berechnungskosten von π' bei einer sehr großen Anzahl von Aktionen sehr stark ansteigen. Im Extremfall ist der Aktionenraum kontinuierlich. Die bisher dargestellten Algorithmen zeigen noch keine effiziente Lösung für diese Problemstellung auf. [Lap18, S. 242] [Sil15g, S. 5] [LHP⁺16] Dieser Nachteil ist durch Betrachtung des Vektors in Gleichung 3.39 leicht ersichtlich. Um die Aktion a mit dem größten langfristig zu erwartenden Ertrag $a = \arg \max_a \hat{q}(s, \mathbf{a}, \mathbf{w})$ zu finden, müssten alle Elemente des Vektors durchlaufen werden. Dies ist bei Aktionsräumen mit n möglichen Aktionen mit einer Laufzeitkomplexität von $\mathcal{O}(n)$ verbunden.

Aufgrund des schwindenden Verfallfaktors ε im ε -*greedy*-Ansatz tendieren die anfänglich stochastischen *Policies* von *value-based*-Algorithmen hinzu deterministischen, d.h. $\pi(a | s) \in \{0, 1\}$ für alle s und a . Zwar kann ε immer größer Null sein und damit die *Policy* nie vollends deterministisch, aber in manchen Umgebungen ist auch ein Beinahe-Determinismus nicht erstrebenswert. Ist die Umgebung etwa nicht *fully observable*, dann können *value-based*-Algorithmen nicht mehr eingesetzt werden. Beispiel einer Umgebung, die nicht *fully observable* ist, ist das Spiel „Schere, Stein, Papier“, da ein Agent den Zustand der gegnerischen Hand nicht vorab kennen wird. In solchen Fällen ist eine durchwegs stochastische *Policy* besser geeignet, da sie immer das Element der Zufälligkeit beinhaltet. [Sil15g, S. 6] [SB17, S. 266]

Wie sich im folgenden Abschnitt 3.4 zeigen wird, können *policy-based*-Algorithmen einige dieser Nachteile entgegenwirken.

3.4 Policy-based Reinforcement Learning

Policy-based-Algorithmen parametrieren die *Policy* direkt, ähnlich der Parametrierung von *State-Value*- und *Action-Value*-Funktion in Unterabschnitt 3.3.4. [Sil15g, S. 3] Um Algorithmen und Methoden, die eine parametrierte *Policy* lernen, geht es in diesem Abschnitt. In Unterabschnitt 3.4.1 werden stochastische *Policy-Gradient*-Methoden eingeführt und fundiert erklärt. Aufbauend darauf werden in den nachfolgenden Unterabschnitten verschiedene stochastische *Policy-Gradient*-Algorithmen vorgestellt. Unterabschnitt 3.4.4 erklärt die Idee hinter deterministischen *Policy-Gradient*-Methoden und in Unterabschnitt 3.4.5 wird derjenige deterministische *Policy-Gradient*-Algorithmus erklärt, der in Kapitel 4 und Kapitel 5 im Experiment, zur Erreichung der in Kapitel 1 genannten Ziele, genutzt wird.

3.4.1 Stochastische Policy-Gradient-Methoden

Sei $\boldsymbol{\theta} \in \mathbb{R}^{d'}$, $d' \in \mathbb{N}$ der Parametervektor einer *Policy* π . Dann ist, erweiternd zu Definition 3.2.11, $\pi(a | s, \boldsymbol{\theta}) = \mathbb{P}(A_t = a | S_t = s, \boldsymbol{\theta}_t = \boldsymbol{\theta})$ die Wahrscheinlichkeit, dass Aktion a in Zustand s mit Parametern $\boldsymbol{\theta}$ zum Zeitschritt t von $\pi_{\boldsymbol{\theta}}$ ausgeführt wird. Wird ein Performanzmaß $J(\boldsymbol{\theta})$ auf dem Parametervektor eingeführt, dann kann formuliert werden, dass *Policy-Gradient*-Methoden mittels des Gradientenaufstiegsverfahrens (engl. *gradient ascent*) den Parametervektor mit

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha \nabla J(\boldsymbol{\theta}^t) \quad (3.40)$$

$$= \boldsymbol{\theta}^t + \alpha \begin{pmatrix} \frac{\partial J}{\partial \theta_1^t} \\ \vdots \\ \frac{\partial J}{\partial \theta_n^t} \end{pmatrix} \quad (3.41)$$

iterativ anpassen. [Sil15g, S. 10 ff.] Alle Optimierungsalgorithmen einer parametrisierten *Policy* werden *Policy-Gradient*-Methoden genannt. [SB17, S. 265]

In Unterabschnitt 3.3.4 wurde für das Performanzmaß noch die Fehlerfunktion der quadratischen Abweichung $E(\mathbf{w}) = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$ angenommen. Die Herleitung des Performanzmaßes $J(\boldsymbol{\theta})$ für *Policy-Gradient*-Methoden ist jedoch weniger offensichtlich. Hierfür muss, ähnlich der Unterscheidung von MC-Methoden und TD-Methoden, zwischen endlichen und unendlichen Episoden differenziert werden. [Sil15g, S. 10] Für beide Fälle erklärt der Satz zu *Policy Gradients*, wie $\nabla J(\boldsymbol{\theta})$ zu berechnen ist. [Sil15g, S. 20] [SB17, S. 269, 276]

Im Fall von endlichen Episoden gilt $J(\boldsymbol{\theta}) = v_{\pi_{\boldsymbol{\theta}}}(s)$. [SB17, S. 268] Umgangssprachlich wird die Güte des Parametervektors $\boldsymbol{\theta}$ also durch den Ertrag des ersten Zustandes der Episode gemessen. Ziel ist es nach Gleichung 3.41 einen Ausdruck zu finden, sodass $\frac{\partial J}{\partial \theta_i} = \frac{\partial}{\partial \theta_i} v_{\pi_{\boldsymbol{\theta}}}(s)$ für alle $i \in \{1, \dots, n\}$ bestimmt werden kann. Zwar ist durch Definition 3.2.12 bekannt, wie $v_{\pi}(s)$ bestimmt wird, jedoch ist dort die *Policy* nicht parametrisiert. Der Gradient des Performanzmaßes einer parametrisierten *Policy* bei endlichen Episoden lässt sich folgendermaßen formulieren, angenommen $\gamma = 1$:

$$\nabla J(\boldsymbol{\theta}) = \nabla v_{\pi_{\boldsymbol{\theta}}}(s) \quad (3.42)$$

$$= \nabla \left(\sum_a \pi(a | s, \boldsymbol{\theta}) \left(\mathcal{R}_s^a + \sum_{s'} \mathcal{P}_{ss'}^a v_{\pi_{\boldsymbol{\theta}}}(s') \right) \right), \forall s \in \mathcal{S} \quad (\text{s. Gl. 3.27})$$

$$= \nabla \left(\sum_a \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \right) \quad (\text{s. Gl. 3.28})$$

$$= \sum_a (\nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi(a | s, \boldsymbol{\theta}) \nabla q_{\pi_{\boldsymbol{\theta}}}(s, a)) \quad (\text{mit } f' = u'v + uv')$$

$$= \sum_a \left(\nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi(a | s, \boldsymbol{\theta}) \nabla \left(\mathcal{R}_s^a + \sum_{s'} \mathcal{P}_{ss'}^a v_{\pi_{\boldsymbol{\theta}}}(s') \right) \right) \quad (\text{Gl. 3.28})$$

$$= \sum_a \left(\nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \pi(a | s, \boldsymbol{\theta}) \left(\mathcal{R}_s^a + \sum_{s'} \mathcal{P}_{ss'}^a \nabla v_{\pi_{\boldsymbol{\theta}}}(s') \right) \right) \quad (\text{Abl. bzgl. } \boldsymbol{\theta})$$

$$= \sum_a \nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) + \sum_a \left(\pi(a | s, \boldsymbol{\theta}) \left(\mathcal{R}_s^a + \sum_{s'} \mathcal{P}_{ss'}^a \nabla v_{\pi_{\boldsymbol{\theta}}}(s') \right) \right). \quad (3.43)$$

Gleichung 3.43 weist eine Rekursion bezüglich $\nabla v_{\pi_{\boldsymbol{\theta}}}(s)$ und $\nabla v_{\pi_{\boldsymbol{\theta}}}(s')$ auf. Statt, wie obig geschehen, kann für $q_{\pi_{\boldsymbol{\theta}}}(s, a) = \mathcal{R}_s^a + \sum_{s'} \mathcal{P}_{ss'}^a v_{\pi_{\boldsymbol{\theta}}}(s')$, auch $q_{\pi_{\boldsymbol{\theta}}}(s, a) = \sum_{s'} \sum_r \mathcal{P}_{ss'}^{a,r} (r + v_{\pi_{\boldsymbol{\theta}}}(s'))$ geschrieben werden, da \mathcal{R}_s^a nach Definition 3.2.10 selber eine Wahrscheinlichkeit ist, die nach Abschnitt 3.1 von der Umgebung festgelegt wird. [SB17, S. 38] Sei weiterhin der erste Summand aus Gleichung 3.43 als $\Lambda(s) = \sum_a \nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a)$ definiert. Dann lässt sich Gleichung 3.43 verkürzend schreiben zu:

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}}(s) \\ &= \Lambda(s) + \sum_a \left(\pi(a | s, \boldsymbol{\theta}) \left(\mathcal{R}_s^a + \sum_{s'} \mathcal{P}_{ss'}^a \nabla v_{\pi_{\boldsymbol{\theta}}}(s') \right) \right)\end{aligned}\quad (3.44)$$

$$= \Lambda(s) + \sum_a \pi(a | s, \boldsymbol{\theta}) \sum_{s'} \mathcal{P}_{ss'}^a \nabla v_{\pi_{\boldsymbol{\theta}}}(s') \quad (3.45)$$

$$= \Lambda(s) + \sum_{s'} \sum_a \pi(a | s, \boldsymbol{\theta}) \mathcal{P}_{ss'}^a \nabla v_{\pi_{\boldsymbol{\theta}}}(s') \quad (3.46)$$

$$= \Lambda(s) + \sum_{s'} \mathbb{P}(s \rightarrow s', 1, \pi_{\boldsymbol{\theta}}) \nabla v_{\pi_{\boldsymbol{\theta}}}(s'), \quad (3.47)$$

wobei $\mathbb{P}(s \rightarrow s', k, \pi_{\boldsymbol{\theta}})$ die Wahrscheinlichkeit eines Zustandsüberganges von s nach s' in $k \in \mathbb{N}$ Schritten unter Policy $\pi(a | s, \boldsymbol{\theta})$ ist. Diese Wahrscheinlichkeit dient also als eine Gewichtung des Gradienten des Ertrages $v_{\pi_{\boldsymbol{\theta}}}(s^{(k)})$. Die Rekursionsgleichung lässt sich weiter ausschreiben zu:

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}}(s) \\ &= \Lambda(s) + \sum_{s'} \mathbb{P}(s \rightarrow s', 1, \pi_{\boldsymbol{\theta}}) \nabla v_{\pi_{\boldsymbol{\theta}}}(s') \\ &= \Lambda(s) + \sum_{s'} \mathbb{P}(s \rightarrow s', 1, \pi_{\boldsymbol{\theta}}) (\Lambda(s') + \sum_{s''} \mathbb{P}(s' \rightarrow s'', 2, \pi_{\boldsymbol{\theta}}) \nabla v_{\pi_{\boldsymbol{\theta}}}(s''))\end{aligned}\quad (3.48)$$

⋮

$$= \sum_s \sum_{k=0}^{\infty} \mathbb{P}(s_0 \rightarrow s, k, \pi_{\boldsymbol{\theta}}) \Lambda(s) \quad (3.49)$$

$$= \sum_s \sum_{k=0}^{\infty} \mathbb{P}(s_0 \rightarrow s, k, \pi_{\boldsymbol{\theta}}) \sum_a \nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a). \quad (3.50)$$

Wird überdies $\eta(s) = \sum_{k=0}^{\infty} \mathbb{P}(s \rightarrow s', k, \pi_{\boldsymbol{\theta}})$ definiert, lässt sich weiterhin schreiben:

$$\begin{aligned}\nabla J(\boldsymbol{\theta}) &= \nabla v_{\pi_{\boldsymbol{\theta}}}(s) \\ &= \sum_s \eta(s) \sum_a \nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a)\end{aligned}\quad (3.51)$$

$$= \left(\sum_s \eta(s) \right) \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \sum_a \nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \quad (\text{Normalisierung})$$

$$\propto \sum_s \psi(s) \sum_a \nabla \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \quad (3.52)$$

$$= \sum_s \psi(s) \sum_a \pi(a | s, \boldsymbol{\theta}) \frac{\nabla \pi(a | s, \boldsymbol{\theta})}{\pi(a | s, \boldsymbol{\theta})} q_{\pi_{\boldsymbol{\theta}}}(s, a) \quad (\nabla \pi(a | s, \boldsymbol{\theta}) = \pi(a | s, \boldsymbol{\theta}) \frac{\nabla \pi(a | s, \boldsymbol{\theta})}{\pi(a | s, \boldsymbol{\theta})})$$

$$= \sum_s \psi(s) \sum_a \pi(a | s, \boldsymbol{\theta}) \nabla \ln \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) \quad (\text{mit } \nabla \ln x = \frac{\nabla x}{x})$$

$$= \mathbb{E}_{s \sim \psi, a \sim \pi_{\boldsymbol{\theta}}} (\nabla \ln \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a)). \quad (3.53)$$

Gleichung 3.52 beschreibt, dass der Gradient von $J(\boldsymbol{\theta})$ proportional zu der Wahrscheinlichkeitensumme $\sum_s \psi(s) = \sum_s \frac{\eta(s)}{\sum_s \eta(s)}$, die die Wahrscheinlichkeit angibt, dass sich der Agent in Zustand s befindet, multipliziert mit der Summe der $q_{\pi_{\boldsymbol{\theta}}}(s, a)$ -gewichteten Gradienten der

Wahrscheinlichkeiten für die Auswahl der Aktionen, ist. Dabei ist $\sum_s \eta(s)$ die Proportionalitätskonstante. Um auf Gleichung 3.53 zu kommen, wird $(\ln x)' = \frac{1}{x}$ genutzt. Praktisch liefert Gleichung 3.53 die Erkenntnis, dass wenn der Agent in Zustand s ist, er sich in die Richtung von Gewichten bewegen soll, die den größten Anstieg in der Wahrscheinlichkeit der Auswahl von Aktion a in s vorweisen, gewichtet mit dem zu erwartenden Ertrag $q_{\pi_\theta}(s, a)$ für diesen Zustand und Aktion. Je größer $q_{\pi_\theta}(s, a)$, desto mehr soll sich der Agent in die Richtung des größten Wahrscheinlichkeitsanstieges der Auswahl von Aktion a in Zustand s bewegen. Abschließend zufriedenstellend ist Gleichung 3.53 jedoch nicht, da sie wiederum einen Gradienten beinhaltet. Wie sich aber gleich noch zeigen wird, kann der Ausdruck $\nabla \ln \pi(a | s, \theta)$ für je diskrete und kontinuierliche Aktionsräume noch weiter ausgeführt werden, sodass sich $\frac{\partial J}{\partial \theta_i}, \forall i \in \{1, \dots, \text{len}(\theta)\}$ analytisch darstellen lässt.

Im Fall von unendlichen Episoden ist das Performanzmaß $J(\theta)$ die durchschnittliche Belohnung pro Zeitschritt. [SB17, S. 275] Formell bedeutet dies:

$$J(\theta) = \lim_{h \rightarrow \infty} \frac{1}{h} \sum_{t=1}^h \mathbb{E}(R_t | A_{0:t-1} \sim \pi_\theta) \quad (3.54)$$

$$= \sum_s \psi(s) \sum_a \pi(a | s, \theta) \mathcal{R}_s^a. \quad (3.55)$$

Der Gradient $\nabla J(\theta)$ kann mit dem Satz zu *Policy Gradients* für unendliche Episoden bestimmt werden. Aus diesem geht hervor, dass

$$\nabla J(\theta) = \sum_s \psi(s) \sum_a \pi(a | s, \theta) \nabla \ln \pi(a | s, \theta) q_{\pi_\theta}(s, a) \quad (3.56)$$

$$= \mathbb{E}_{s \sim \psi, a \sim \pi_\theta} (\nabla \ln \pi(a | s, \theta) q_{\pi_\theta}(s, a)) \quad (3.57)$$

und damit entspricht die Rechenvorschrift dieses Gradienten in schöner Weise die dem Gradienten im endlichen Fall. [SB17, S. 276] [Sil15g, S. 20]

Policy-Gradient-Methoden können in Umgebungen mit diskretem, als auch mit kontinuierlichem Aktionenraum verwendet werden. *Value-based*-Algorithmen aus Abschnitt 3.3 haben den Nachteil nur in Umgebungen mit diskretem Aktionenraum einsetzbar zu sein. Ob eine gegebene *Policy-Gradient*-Methode auf diskretem oder kontinuierlichen Aktionenraum eingesetzt werden kann, wird durch die *Policy*-Parametrisierung bestimmt. [SB17, S. 266]

Im Falle eines diskreten Aktionenraumes, kann für jedes Zustand-Aktion-Paar ein numerischer Wert $h(s, a, \theta) \in \mathbb{R}$ berechnet werden. Die Berechnung dieser Werte kann beispielsweise über ein neuronales Netz geschehen, ähnlich wie in Gleichung 3.39. Eine exponentielle Softmax-Verteilung

$$\pi(a | s, \theta) = \frac{\exp(h(s, a, \theta))}{\sum_b \exp(h(s, b, \theta))} \quad (3.58)$$

kann dann angeben, mit welcher Wahrscheinlichkeit eine Aktion ausgewählt werden sollte. [SB17, S. 266] [BL17, S. 15]

Im Fall eines großen oder sogar kontinuierlichen Aktionenraumes werden nicht die einzelnen Aktionen bewertet, sondern es werden Statistiken von Verteilungsfunktionen wie der Normalverteilung berechnet. Die Statistiken der Normalverteilung sind der Durchschnitt $\mu \in \mathbb{R}$ und die Varianz $\sigma^2 > 0$. [FKPG03, S. 292] Die Normalverteilung ist gegeben durch:

$$p(x) = \frac{1}{\sigma \sqrt{2\pi}} \exp\left(-\frac{(x - \mu)^2}{2\sigma^2}\right). \quad (3.59)$$

Die Normalverteilung ist über die reellen Zahlen definiert, ebenso wie ein kontinuierlicher Aktionenraum. Durchschnitt und Varianz, bzw. Standardabweichung, können parametrisiert

und damit approximiert werden, mit:

$$\pi(a | s, \boldsymbol{\theta}) = \frac{1}{\sigma(s, \boldsymbol{\theta})\sqrt{2\pi}} \exp\left(-\frac{(a - \mu(s, \boldsymbol{\theta}))^2}{2\sigma(s, \boldsymbol{\theta})^2}\right). \quad (3.60)$$

Aus Gleichung 3.59 wird erkenntlich, dass der *Policy*-Parametervektor mit $\boldsymbol{\theta} = (\boldsymbol{\theta}_\mu, \boldsymbol{\theta}_\sigma)^T$ beschrieben werden kann. Die Parameter $\boldsymbol{\theta}_\mu$ und $\boldsymbol{\theta}_\sigma$ können praktisch durch jeweils ein neuronales Netz berechnet werden. Dies ermöglicht die Umsetzung von kontinuierlichen Aktionen für stochastische *Policies*. [SB17, S. 277 f.] [Sil15g, S. 18] [Lap18, S. 404 ff.]

Da sowohl die Softmax-Verteilung aus Gleichung 3.58, als auch die Normalverteilung aus Gleichung 3.60 auf \mathbb{R} differenzierbar sind, können sie in die Gleichung 3.53 des *Policy*-Gradienten eingesetzt werden. So kann über die Kettenregel schlussendlich $\frac{\partial J}{\partial \theta_i}, \forall i \in \{1, \dots, n\}$ analytisch bestimmt werden und damit, nach Gleichung 3.41, der *Policy*-Parametervektor im *Gradient Ascent*-Verfahren angepasst werden.

Die in diesem Unterabschnitt dargestellten und erklärten Formalismen zu stochastischen *Policy-Gradient*-Methoden werden in den nachfolgenden Abschnitten in Algorithmen nutzbar gemacht.

3.4.2 Monte-Carlo Policy-Gradient

Der Monte-Carlo Policy-Gradient-Algorithmus ist ein rein *Policy*-basierter Algorithmus und nutzt, ähnlich der MC-Methoden in Unterabschnitt 3.3.2, den Ertrag vollständiger, endlicher Episoden für die Parameteranpassung. Der Algorithmus wird in seiner initialen Publikation [Wil92] auch REINFORCE genannt. Eine Version dieses *Policy-Gradient*-Algorithmus ist als Pseudocode in Algorithmus 2 zu sehen. [SB17, S. 271] Dabei wird in dieser Version der Ertrag mit jedem Zeitschritt graduell erweitert, anders als in Unterabschnitt 3.3.2. Die Gewichtsanpassung $\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha\gamma^t G_t \nabla \ln \pi(a | s, \boldsymbol{\theta}^t)$ des Algorithmus sorgt dafür, dass der Gewichtsvektor $\boldsymbol{\theta}^t$ mit dem Produkt aus Ertrag G_t und dem Gradientenvektor $\nabla \ln \pi(a | s, \boldsymbol{\theta})$, der den größten Anstieg der Wahrscheinlichkeit der Auswahl von Aktion a in Zustand s angibt, angepasst wird. Der Ertrag G_t funktioniert hier also als Skalierungsfaktor des Gradienten. Je höher der Ertrag, desto größer soll die Wahrscheinlichkeit einer erneuten Auswahl von Aktion a in Zustand s durch die *Policy* $\pi_{\boldsymbol{\theta}}$ sein. [SB17, S. 270]

Algorithmus 2 : Monte-Carlo Policy-Gradient (REINFORCE)

```

Input : a differentiable policy parameterization  $\pi(a | s, \boldsymbol{\theta})$ 
Result : policy parameter  $\boldsymbol{\theta}$ 
Initialize policy parameter  $\boldsymbol{\theta} \in \mathbb{R}^{d'}$  ;
Repeat forever
    Generate an episode  $s_0, a_0, r_1, \dots, s_{T-1}, a_{T-1}, r_T$  using policy  $\pi_{\boldsymbol{\theta}}$  ;
    foreach timestep of the episode  $t = 0, \dots, T-1$  do
         $G_t \leftarrow$  return from step  $t$  ;
         $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t + \alpha\gamma^t G_t \nabla \ln \pi(a | s, \boldsymbol{\theta}^t)$  ;
    end
```

Aus Algorithmus 2 wird erkenntlich, dass der Monte-Carlo Policy-Gradient-Algorithmus eine *Stochastic Gradient Ascent*-Optimierung nutzt, da die Gewichte in jedem Zeitschritt angepasst werden. Für den Gradienten $\nabla \ln \pi(a | s, \boldsymbol{\theta}^t)$ kann, je nachdem ob es sich um eine Umgebung mit diskretem oder kontinuierlichem Aktionenraum handelt, die in Unterabschnitt 3.4.1 erklärten *Policy*-Parametrisierungen eingesetzt werden. Nachteil des Monte-Carlo Policy-Gradient-Algorithmus ist eine hohe Varianz der Erträge zwischen den Zeitschritten und die damit verbundene Langsamkeit des Lernfortschrittes. [SB17, S. 272] Grund der Varianz ist die unadaptive und lose Formulierung des Skalierungsfaktors. [Lap18, S. 264 f.]

3.4.3 Actor-Critic Policy-Gradient

Mit Gleichung 3.53 wurde gezeigt, dass der langfristig zu erwartende Ertrag für eine Aktion in einem Zustand, $q_{\pi_\theta}(s, a)$, ein Skalierungsfaktor für den größten Anstieg der Wahrscheinlichkeit der Auswahl einer Aktion in einem Zustand ist. In Unterabschnitt 3.4.2 ist der Ertrag G_t dieser Skalierungsfaktor. Im Versuch die hohe Varianz und die langsame Lerngeschwindigkeit des Monte-Carlo Policy-Gradient-Algorithmus zu verbessern, wird der Satz zu *Policy Gradients* um eine *Baseline* $b(s)$ erweitert, die adaptiv, d.h. individuell für jeden Zustand, Wirkung auf den Skalierungsfaktor ausübt. [SB17, S. 272] Diese Erweiterung wird formuliert mit:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \psi(s) \sum_a (q_{\pi_\theta}(s, a) - b(s)) \nabla \pi(a | s, \boldsymbol{\theta}). \quad (3.61)$$

Im Fall $b(s) = 0$ für alle $s \in \mathcal{S}$, ergibt sich wieder Gleichung 3.52. Als *Baseline* könnte beispielsweise ein $|\mathcal{S}| \times 1$ -Vektor mit Konstanten für jeden Zustand genutzt werden. Wird als *Baseline* ein Funktionsapproximator mit *Bootstrapping* genutzt, wie etwa TD(0), SARSA(0) oder Q-Learning, kann das Konzept von *Actor-Critic*-Methoden formuliert werden. Der *Critic* berechnet dabei den Skalierungsfaktor und der *Actor* passt die *Policy*-Parameter $\boldsymbol{\theta}$ an. [Sil15g, S. 23] Für den *Actor* ergibt sich, angenommen $\gamma = 1$, dann folgende Formel zur Gewichtsanpassung:

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha \nabla J(\boldsymbol{\theta}) \quad (\text{s. Gl. 3.40})$$

$$= \boldsymbol{\theta}^t + \alpha (q_{\pi_\theta}(s, a) - b(s)) \nabla \ln \pi(a | s, \boldsymbol{\theta}) \quad (\text{s. Gl. 3.61})$$

$$= \boldsymbol{\theta}^t + \alpha (R_{t+1} + \hat{q}(s', a', \mathbf{w}^t) - \hat{q}(s, a, \mathbf{w}^t)) \nabla \ln \pi(a | s, \boldsymbol{\theta}^t) \quad (3.62)$$

$$= \boldsymbol{\theta}^t + \alpha \delta_t \nabla \ln \pi(a | s, \boldsymbol{\theta}^t). \quad (3.63)$$

In Gleichung 3.62 wird als Skalierungsfaktor das *Deep-SARSA(0)-Target* $R_{t+1} + \hat{q}(s', a', \mathbf{w}^t)$ angenommen. Die *Baseline* $\hat{q}(s, a, \mathbf{w}^t)$ versucht stark schwankenden *Target*-Werten entgegenzuwirken. Der *Critic* gibt dem *Actor* somit den Skalierungsfaktor vor. In Algorithmus 3 ist der Pseudocode [SB17, S. 274] für einen solchen *Actor-Critic*-Algorithmus dargestellt.

Algorithmus 3 : One-Step Actor-Critic

```

Input : a differentiable policy parameterization  $\pi(a | s, \boldsymbol{\theta})$ 
Input : a differentiable action-value parameterization  $\hat{q}(s, a, \mathbf{w})$ 
Result : policy parameter  $\boldsymbol{\theta}$  and action-value weights  $\mathbf{w}$ 
Initialize policy parameter  $\boldsymbol{\theta} \in \mathbb{R}^{d'}$  and action-value weights  $\mathbf{w} \in \mathbb{R}^d$  ;
Repeat forever
    Observe state  $s$  from environment ;
    Sample action  $a$  from  $\pi(a | s, \boldsymbol{\theta})$  ;
    foreach timestep of the episode  $t = 0, \dots, T - 1$  do
        Take action  $a$ , observe  $s'$  and  $r$  from environment ;
        Sample next action  $a'$  from  $\pi(a' | s', \boldsymbol{\theta}^t)$  ;
         $\delta_t \leftarrow r_t + \gamma \hat{q}(s', a', \mathbf{w}^t) - \hat{q}(s, a, \mathbf{w}^t)$  ;
         $\boldsymbol{\theta}^{t+1} \leftarrow \boldsymbol{\theta}^t + \alpha \delta_t \nabla \ln \pi(a | s, \boldsymbol{\theta}^t)$  ;
         $\mathbf{w}^{t+1} \leftarrow \mathbf{w}^t + \beta \delta_t \nabla \hat{q}(s, a, \mathbf{w}^t)$  ;
         $a \leftarrow a'$  ;  $s \leftarrow s'$  ;
    end

```

Ebenso wie n -Schritt TD-Methoden, aus Unterabschnitt 3.3.3 bekannt, gibt es auch n -Schritt *Actor-Critic*-Algorithmen, die in [SB17, S. 274 f.] und in [Sil15g, S. 34] grundlegend erklärt und ausführlich dargestellt werden.

3 Theoretische Grundlagen

Um die Varianz weiter zu verringern kann in Gleichung 3.61, anstatt von $q_{\pi_\theta}(s, a)$, eine *Advantage*-Funktion genutzt werden. [Sil15g, S. 29] Diese ist definiert als

$$a_{\pi_\theta}(s, a) = q_{\pi_\theta}(s, a) - v_{\pi_\theta}(s). \quad (3.64)$$

Die *Advantage*-Funktion gibt dann an, ob und um wie viel Aktion a in Zustand s im Vergleich zu allen anderen möglichen Aktionen in Zustand s besser oder schlechter ist. Ist a_{π_θ} größer als Eins, ist Aktion a im Mittel besser als anderen Aktionen. Die *Advantage*-Funktion könnte mit zwei Funktionsapproximatoren mit zwei unterschiedlichen Parametervektoren $\mathbf{w} \in \mathbb{R}^d, \chi \in \mathbb{R}^{d''}$ bestimmt werden, $a_{\pi_\theta}(s, a) \approx \hat{q}(s, a, \chi) - \hat{v}(s, \mathbf{w})$, jedoch ist es einfacher, unter Zuhilfenahme von Gleichung 3.28, nur einen Funktionsapproximator zu nutzen. Dann kann die *Advantage*-Funktion berechnet werden mit:

$$a_{\pi_\theta}(s, a) \approx \hat{q}(s, a, \chi) - \hat{v}(s, \mathbf{w}) \quad (3.65)$$

$$= R_{t+1} + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w}). \quad (3.66)$$

Für den *Policy Gradient* ergibt sich folglich

$$\nabla J(\boldsymbol{\theta}) = \mathbb{E}_{\pi_\theta} (\nabla \ln \pi(a | s, \boldsymbol{\theta}) a_{\pi_\theta}(s, a)). \quad (3.67)$$

Für eine zu Gleichung 3.62 äquivalente Anpassungsvorschrift gilt demnach

$$\boldsymbol{\theta}^{t+1} = \boldsymbol{\theta}^t + \alpha (R_{t+1} + \gamma \hat{v}(s', \mathbf{w}^t) - \hat{v}(s, \mathbf{w}^t)) \nabla \ln \pi(a | s, \boldsymbol{\theta}^t). \quad (3.68)$$

3.4.4 Deterministische Policy-Gradient-Methoden

In obigen Abschnitten wird die *Policy*-Funktion stets als parametrierte Wahrscheinlichkeitsverteilung $\pi(a | s, \boldsymbol{\theta})$ angenommen, die stochastisch eine Aktion a in Zustand s , basierend auf den Parametern $\boldsymbol{\theta}$, bestimmt. In diesem Unterabschnitt werden deterministische *Policy*-Funktionen $a = \tau(s, \boldsymbol{\theta})$ betrachtet, für die Silver et al. in [SLH⁺14] die Existenz des Gradienten des zugehörigen Performanzmaßes $\nabla J(\boldsymbol{\theta})$ bewiesen haben. Unterschied zwischen stochastischer und deterministischer *Policy* ist, dass die deterministische *Policy* keine Wahrscheinlichkeitsverteilung über den Aktionenraum hat, sondern für einen konkreten Zustand immer nur eine konkrete Aktion vorsieht. Deterministische *Policies* können folglich als Spezialfall von stochastischen *Policies* bezeichnet werden, bei denen die stochastische *Policy* eine Wahrscheinlichkeitsverteilung mit genau einem Extremalwert aufweist. [SLH⁺14] [Wen18]

In Unterabschnitt 3.4.3 wurde der *Critic* genutzt, um den Skalierungsfaktor δ_t zu bestimmen. Ist der Aktionenraum einer Umgebung unendlich, ermöglichen deterministische *Policy-Gradient*-Methoden die Berechnung von $\nabla q_{\tau_\theta}(s, \tau(s, \boldsymbol{\theta}))$ bzgl. $\boldsymbol{\theta}$. [Lap18, S. 410 f.] Der Vorteil ist eine einfachere Berechnung des Gradienten, wie weiter unten ausführlich beschrieben wird. Zunächst muss aber, wie gewohnt, das Performanzmaß für deterministische *Policy*-Funktionen festgelegt werden. Dies ist gegeben durch

$$J(\boldsymbol{\theta}) = \int_S \psi(s) q_{\tau_\theta}(s, \tau(s, \boldsymbol{\theta})) ds \quad (3.69)$$

$$= \mathbb{E}_{s \sim \psi}(q_{\tau_\theta}(s, \tau(s, \boldsymbol{\theta}))). \quad (3.70)$$

Das Integral wird hier verwendet, weil per Konvention der Zustands- und Aktionenraum einer Umgebung bei deterministischen *Policies* als immer unendlich angesehen wird. Ähnlich hätte man durchaus auch die Gleichung 3.56 mit einem Integral über die Zustände formulieren können, aber eine einheitliche Formulierung mit Gleichung 3.52 wurde vorgezogen. Überdies lassen stochastische *Policies*, durch die Auswahl der *Policy*-Parametrierung, immer die praktische Möglichkeit eines endlichen Aktionenraumes zu.

Optimierungsaufgabe ist wieder $\max_{\boldsymbol{\theta}} \mathbb{E}_{s \sim \psi}(q_{\tau_\theta}(s, \tau(s, \boldsymbol{\theta})))$ mit dem *Gradient Ascent*-Verfahren. Mit Gleichung 3.69 kann durch den Satz zu *Deterministic Policy Gradients* aus

[SLH⁺14] der Gradient des Performanzmaßes für deterministische *Policies* hergeleitet werden:

$$\nabla J(\boldsymbol{\theta}) = \int_{\mathcal{S}} \psi(s) \nabla q_{\tau_{\boldsymbol{\theta}}}(s, \tau(s, \boldsymbol{\theta})) ds \quad (3.71)$$

$$= \int_{\mathcal{S}} \psi(s) \nabla q_{\tau_{\boldsymbol{\theta}}}(s, \tau(s, \boldsymbol{\theta})) \nabla \tau(s, \boldsymbol{\theta}) ds \quad (\text{mit } f(g(x))' = f'(g(x))g'(x)) \\ = \mathbb{E}_{s \sim \psi} (\nabla q_{\tau_{\boldsymbol{\theta}}}(s, \tau(s, \boldsymbol{\theta})) \nabla \tau(s, \boldsymbol{\theta})). \quad (3.72)$$

Um einen Vorteil von deterministischen *Policy-Gradient*-Methoden gegenüber stochastischen deutlich zu machen, hier nun eine zusammengefasste, integrale Darstellung von Gleichung 3.56 mit den Annahmen, Zustand- und Aktionenraum sind unendlich:

$$\overset{stoch.}{\nabla} J(\boldsymbol{\theta}) \approx \int_{\mathcal{S}} \psi(s) \int_{\mathcal{A}} \nabla \ln \pi(a | s, \boldsymbol{\theta}) q_{\pi_{\boldsymbol{\theta}}}(s, a) da ds. \quad (3.73)$$

Es wird deutlich, dass zur Berechnung des Gradienten einer stochastischen *Policy* mit Gleichung 3.73 über den gesamten Aktionenraum integriert werden muss. Dies ist nicht nur aufwändiger als Gleichung 3.71, sondern stochastische *Policies* benötigt in der Praxis dadurch auch mehr Daten aus der Agent-Umgebung-Interaktion. [SLH⁺14] In der Tat widerspricht die Annahme eines unendlichen Zustand- und Aktionenraum der Definition eines MDP aus Definition 3.2.10, jedoch ist in der Praxis mit unendlich meist nur sehr, sehr groß gemeint.

Ein ausführlicher Beweis zu Gleichung 3.72 findet sich im Anhang von [SLH⁺14]. Aus Gleichung 3.72 ergibt sich wieder die für alle *Policy-Gradient*-Methoden ähnlich aufgebaute Anpassungsvorschrift des Parametervektors $\boldsymbol{\theta}$ im *Gradient Ascent*-Verfahren, dargestellt in Gleichung 3.74.

$$\begin{aligned} \boldsymbol{\theta}^{t+1} &= \boldsymbol{\theta}^t + \alpha \nabla J(\boldsymbol{\theta}^t) && (\text{s. Gl. 3.40}) \\ &= \boldsymbol{\theta}^t + \alpha \nabla \hat{q}(s, \tau(s, \boldsymbol{\theta}^t), \mathbf{w}^t) \nabla \tau(s, \boldsymbol{\theta}^t). \end{aligned} \quad (3.74)$$

Ein Vorteil stochastischer *Policies* ist die probabilistische Auswahl von Aktionen. Die Erkundung des Zustand- und Aktionenraumes wird damit auf implizite Weise ermöglicht. Dies ist bei deterministischen *Policies* nicht gegeben. Lösung dieser Problematik ist die Formulierung eines *off-policy*-Algorithmus nach Unterabschnitt 3.3.3, bei der eine stochastische *Policy* die Aktionen auswählt und eine deterministische *Policy* erlernt wird. [SLH⁺14] Ein solcher *off-policy*-Algorithmus ist der *Deep Deterministic Policy-Gradient*-Algorithmus, der im nachfolgenden Unterabschnitt 3.4.5 dargestellt wird.

3.4.5 Deep Deterministic Policy-Gradient

Der DDPG-Algorithmus ist ein *model-free, off-policy actor-critic* Algorithmus, der, charakteristisch, eine deterministische *Policy* mit Gleichung 3.72 erlernt sowie das Konzept des *Replay Memory* des DQN-Algorithmus nutzt. Zur Erkundung der Umgebungszustände wird eine, durch einen Ornstein-Uhlenbeck-Prozess (OU) induzierte, stochastische *Policy* genutzt.

Ein OU-Prozess ist ein stochastischer Prozess, der mithilfe einer stochastischen Differentialgleichung mathematisch formuliert wird und, typischerweise, einem Gleichgewicht μ entgegen strebt. Wie schnell dem Gleichgewicht entgegen gestrebt wird, bestimmt ein Parameter θ . Wie stark der Einfluss des Zufalls auf den Prozess wirkt, wird durch ein Parameter σ bestimmt. Praktisch wird den reellen Aktionswerten aus der *Policy* τ des DDPG-Algorithmus ein Rauschen hinzugefügt, dessen Mechanik der Gleichung des OU-Prozesses und den Parameterwerten von μ, θ und σ unterliegt. In Definition 3.4.1 [Gru14] wird die Gleichung des OU-Prozesses dargestellt. In Kapitel 4 wird dargestellt, welche Werte die OU-Parameter in der vorliegenden Arbeit einnehmen.

Definition 3.4.1 (Ornstein-Uhlenbeck-Prozess (OU-Prozess))

Seien $a, \mu, \theta, \sigma \in \mathbb{R}$ mit $\theta, \sigma > 0$ konstante Parameter des OU-Prozesses. Sei $(W_t)_{t \in I}$ ein Wiener-Prozess, der Zufallszahlen erzeugt. Seien die Aktionen von Policy τ als Realisationen eines stochastischen Prozesses $(X_t)_{t \in I}$ aufgefasst. Dann wird $(X_t)_{t \in I}$ Ornstein-Uhlenbeck-Prozess mit Anfangswert a , Gleichgewichtsniveau μ , Steifigkeit θ und Diffusion σ genannt, wenn dieser das Anfangswertproblem

$$dX_t = \theta(\mu - X_t)dt + \sigma dW_t, \quad X_0 = a \quad (3.75)$$

löst. Praktisch wird der obigen Gleichung noch der während des Trainings stetig abschwellende Verfallfaktor ε als Skalierungsfaktor vor gestellt.

Für die Formulierung des DDPG-Algorithmus als Pseudocode wird für den OU-Prozess folgendes vereinbart: Sei $\tau(s, \boldsymbol{\theta})$ die deterministische Policy des DDPG-Agenten. Diese soll, entsprechend Gleichung 3.72, verbessert werden. Sei \mathcal{N} das durch den OU-Prozess entstehende Rauschen. Dann ist $\tau(s, \boldsymbol{\theta}) + \varepsilon \mathcal{N} = a$ eine stochastische Policy, die die Aktion a dem Agenten in Zustand s vorgibt. So kann der DDPG-Agent die Umgebung erkunden.

In Unterabschnitt 3.3.4 wurde gesagt, dass sich die *off-policy*-Eigenschaft des DQN-Algorithmus zunutze gemacht wurde, um zwei Parametervektoren für die Stabilisierung der *Gradient Descent*-Optimierung einzuführen. Da das TD-Target als Zielvariable fungiert, gut ersichtlich aus Gleichung 3.35, sollte ein anderer Parametervektor für die Berechnung des derzeitigen Ertrages genutzt werden. Der Parametervektor des TD-Target wird als *Target*-Parametervektor oder, falls die Parameter ein neuronales Netz beschreiben, *Target*-Netz bezeichnet. Der Parametervektor, der den Ertrag des derzeitigen Zustandes errechnet, soll entsprechend des Performanzmaßes optimiert werden. Der *Target*-Parametervektor wurde beim DQN-Algorithmus alle $c \in \mathbb{N}$ Zeitschritte dem zu optimierenden Parametervektor vollends angepasst. Beim DDPG-Algorithmus wird der *Target*-Parametervektor hingegen mit den in Algorithmus 4 dargestellten Gleichungen langsam angepasst. Hierbei gilt $v \ll 1$. Hinzu kommt beim *actor-critic* DDPG-Algorithmus, dass es sowohl Parametervektoren für den *Critic* als auch den *Actor* gibt. Insgesamt werden also vier Parametervektoren genutzt. In Algorithmus 4 ist der DDPG-Agenten-Algorithmus als Pseudocode dargestellt.

Algorithmus 4 : Deep Deterministic Policy-Gradient

```

Result : policy parameter  $\boldsymbol{\theta}$  and action-value weights  $\mathbf{w}$ 
Initialize policy parameter  $\boldsymbol{\theta} \in \mathbb{R}^{d'}$  and action-value weights  $\mathbf{w} \in \mathbb{R}^d$  ;
Initialize target policy parameter  $\boldsymbol{\theta}' \in \mathbb{R}^{d'}$  and target action-value weights  $\mathbf{w}' \in \mathbb{R}^d$  ;
Initialize experience replay memory  $\mathcal{D}$  ;
for episode = 1,  $M$  do
    Observe initial state  $s_0$  from environment ;
    for  $t=1, T$  do
        Select action  $a_t = \tau(s, \boldsymbol{\theta}_t) + \mathcal{N}_t$  ;
        Observe reward  $r_t$  and next state  $s_{t+1}$  from environment ;
        Store  $(s_t, a_t, r_t, s_{t+1})$  tupel in  $\mathcal{D}$  ;
        Sample random batch  $(s_i, a_i, r_i, s_{i+1})$  of size  $B$  from  $\mathcal{D}$  ;
         $\delta_i \leftarrow r_i + \gamma \hat{q}(s_{i+1}, \tau(s_{i+1}, \boldsymbol{\theta}'_t), \mathbf{w}'_t) - \hat{q}(s_i, a_i, \mathbf{w}_t)$  ;
         $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \beta \frac{1}{B} \sum_i^B \delta_i \nabla_{\mathbf{w}} \hat{q}(s_i, a_i, \mathbf{w}_t)$  ;
         $\boldsymbol{\theta}_{t+1} \leftarrow \boldsymbol{\theta}_t + \alpha \frac{1}{B} \sum_i^B \nabla_{\boldsymbol{\theta}} \hat{q}(s_i, \tau(s_i, \boldsymbol{\theta}_t), \mathbf{w}_t) \nabla_{\boldsymbol{\theta}} \tau(s_i, \boldsymbol{\theta}_t)$  ;
        Update target networks by
             $\boldsymbol{\theta}'_{t+1} \leftarrow v \boldsymbol{\theta}_t + (1 - v) \boldsymbol{\theta}'_t$ 
             $\mathbf{w}'_{t+1} \leftarrow v \mathbf{w}_t + (1 - v) \mathbf{w}'_t$ 
    end
end

```

Großer Vorteil des DDPG-Algorithmus ist seine Einsetzbarkeit in Umgebungen mit Aktionen, die durch reellwertige Zahlen ausgedrückt werden. In Kapitel 4 wird der Nutzen reeller Aktionswerte im Kontext autonomer Fahrzeuge deutlich und es wird die Implementierung des DDPG-Algorithmus beschrieben. Der DDPG-Agenten-Algorithmus übernimmt die algorithmische Hauptrolle im praktischen Teil dieser Arbeit aufgrund seiner Einsetzbarkeit in Umgebungen mit reellwertigen Aktionen. Mit dem in Kapitel 4 beschriebenen Experiment werden Performanzdaten zum DDPG-Algorithmus im Kontext des autonomen Fahrens erhoben, die in Kapitel 5 interpretiert werden.

3.4.6 Weitere Policy-Gradient-Algorithmen

Neben den obig dargestellten *Policy-Gradient*-Algorithmen gibt es noch viele weitere. Einer der am häufigst zitierten Algorithmen ist der *Asynchronous Advantage Actor-Critic*-Algorithmus (A3C) von Mnih et al. aus [MBM⁺16]. Idee hinter dem A3C-Algorithmus ist es, mit multiplen, parallelisierten Agenten ein *Experience Replay Memory*, also ein Trainingsdatensatz bestehend aus den Tupeln (s, a, r, s') , aufzubauen, um die Trainingsdatenkorrelation zu mindern. [Lap18, S. 283 ff.] Grund dafür ist das langsame Lernen eines herkömmlichen *Advantage-Actor-Critic*-Agenten, wie aus Unterabschnitt 3.4.3, aufgrund der stark korrelierenden Episodendaten.

Als *Trust-Region*-Algorithmen wird eine Klasse von *Policy-Gradient*-Algorithmen bezeichnet, die eine Antwort auf die Frage nach der größtmöglichen *Policy*-Parameteranpassung suchen. Zu diesen *Trust-Region*-Algorithmen gehört der TRPO- und der PPO-Algorithmus. [SLM⁺15] [SWD⁺17] Eine besonders ausführliche und schöne Übersicht über *Policy-Gradient*-Algorithmen ist auf [Wen18] zu finden.

Im Kontext des autonomen Fahrens sind nach bestem Wissensstand zum Zeitpunkt der Abgabe der vorliegenden Arbeit der DDPG- [WJW18] [KHJ⁺18] [Yi18] [GCSX16], der DQN- [EAPY16] [VN16] [WHW⁺17] und der A3C-Algorithmus [JdT⁺18] [MBM⁺16] die am häufigst vertretenden, modernen *Reinforcement-Learning*-Algorithmen in der zu dieser Arbeit ähnlichen Literatur.

4 Experiment

We were supposed to play
videogames while AI is doing all
the work but now it's the other
way around.

Andrei Karpathy
Director of AI at Tesla

Ziel dieses Kapitels ist die Beschreibung des in der vorliegenden Arbeit durchzuführenden Experiments zur Datenerhebung. Diese Daten werden dann, für die Erreichung der in Kapitel 1 beschriebenen Ziele, in Kapitel 5 ausgewertet. Worüber und welche Daten im Experiment erhoben werden, wird in diesem Kapitel genauer beschrieben. Anfänglich soll zunächst die Tätigkeitsbeschreibung eines autonomen Fahrzeuges anhand dreier Taxonomien dargestellt werden, damit die Anforderungen an einen autonom fahrenden *Reinforcement Learning*-Agenten umrissen sind. Danach wird der Experimentaufbau beschrieben.

4.1 Das autonome Fahrzeug

Thematischer Kontext dieser Arbeit ist das autonome Fahren. Das heißt, es wird ein *Reinforcement Learning*-Agent entwickelt, der autonom ein Fahrzeug, innerhalb einer Simulationsumgebung, führen kann. Dabei ist nicht abschließend definiert, was unter autonomen Fahren allgemeingültig zu verstehen ist. [Mau16] Die Bundesanstalt für Straßenwesen (BASt) hat zwecks einer Rechtsfolgenabschätzung für Fahrzeuge mit hohem Automationsgrad 2010 erstmalig eine fünfstufige Klassifikation der unterschiedlichen Automatisierungsgerade von Fahrzeugen erstellt. [GAA⁺12, S. 31] Diese Klassifikation beschränkt sich nur auf die Automation der Quer- und Längsführung eines Fahrzeuges auf Autobahnen. Nicht auf die Automation von Aspekten des Fahrens wie dem Einparken, der Fußgänger-Fahrzeug-Interaktion oder dem Fahren im Stadtverkehr. Während ein teilautomatisiertes Fahrsystem, Klassifikationsstufe drei, noch einer dauerhaften Überwachung der Quer- und Längsführung durch den Fahrer bedarf, wird dies bei hochautomatisiertem Fahren, Klassifikationsstufe vier, nicht mehr gefordert. Beim vollautomatisierten Fahren, der höchsten und fünften Stufe, kann das Fahrsystem das Fahrzeug zusätzlich jederzeit in einen risikominimalen Systemzustand führen, falls eine Systemgrenze erkannt wird.

Auch die amerikanische National Highway Traffic Safety Administration (NHTSA) hat 2013 ein Klassifikationsschema vorgestellt. Ähnlich dem der BASt besteht es aus fünf Stufen die sich nach dem Grad der Autonomie und Verantwortung gegenüber dem menschlichen Fahrer voneinander abgrenzen. Die Stufen Null bis Drei entsprechen dabei den Stufen Eins bis Vier der Taxonomie des BASt. Stufe Vier der NHTSA geht dabei über die Stufe Fünf des BASt hinaus, da kein risikominimaler Systemzustand mehr vorgesehen ist. Erwartet wird nämlich, dass das autonome Fahrzeug in jeder Situation sicherheitskritische Aktionen ausführen kann. Dies gilt für Fahrzeuge mit und ohne menschlichen Fahrer. [NHTSA13] 2016 wurde das Klassifikationsschema der NHTSA um eine weitere Stufe ergänzt, um der nachfolgend erklärten Taxonomie der Society of Automotive Engineers (SAE) zu entsprechen.

Diese dritte Taxonomie zur Definition und Einteilung von autonomen Fahrzeugen wurde im SAE-Standard J3016 in erster Version 2014 festgelegt. [SI18] Dieser kennt sechs Stufen der Fahrautomation. Diese reichen von „*No Automation*“, Stufe Null bis hin zur „*Full Automation*“, Stufe Fünf. Stufe Fünf des SAE-Standard übersteigt die höchste Klassifikationsstufen der Taxonomie der Bundesanstalt für Straßenwesen, da eine Rückführung in einen risikominimalen Zustand, aufgrund der autonomen Bewältigung jedweder Aufgaben, nicht mehr notwendig ist. Anders als die Stufe Vier der NHTSA-Taxonomie ist aber ein Eingreifen des menschlichen Fahrers immer möglich. Ein Vergleich der hier angeführten Taxonomien ist in Anhang A einsehbar.

Der in dieser Arbeit zu implementierende *Reinforcement Learning*-Agent wird die Quer- und Längsführung eines Fahrzeuges übernehmen. Dazu wird der Agent die Lenkung, Beschleunigung und Bremsung des Fahrzeuges selbstständig erlernen. Eine Übertragung in die obigen Klassifikationsschmata ist dabei nur bedingt möglich, da sich die Stufen der Taxonomien aus dem Autonomie- und Verantwortungsgrad der Fahrautomation, bezogen auf den menschlichen Fahrer, ergründen. Als ideale Zielvorstellung eines solchen *Reinforcement Learning*-Agent für die autonome Führung eines Fahrzeuges ist jedoch die Erreichung von Stufe Fünf der SAE-Taxonomie denkbar. Neben Quer- und Längsführung eines Fahrzeuges sind auch noch weitere automatisierbare Fahrsysteme wie das Ein- und Ausparken oder die Fußgänger-Fahrzeug-Interaktion denkbar. Die vorliegende Arbeit wird sich aus folgenden Gründen ausschließlich auf die Quer- und Längsführung eines autonomen Fahrzeuges fokussieren: Erstens ist die automatisierte Quer- und Längsführung das in der Literatur [KHJ⁺18] [WJW18] [EAPY16] [JdT⁺18] [BTD⁺16] am häufigsten anzutreffende automatisierte Fahrsystem. Dies erhöht die Vergleichbarkeit der vorliegenden Arbeit. Zweitens bilden die geeigneten und frei verfügbaren Simulationsumgebungen zumeist nur die Quer- und Längsführung adäquat ab und nicht etwa die komplexe Fußgänger-Fahrzeug-Interaktion oder das rechtlich richtige Fahrverhalten nach einem Unfall. Dieser letztgenannte Grund weist auf die festzulegenden Anforderungen an die zu verwendende Simulationsumgebung für das Experiment hin, die im nachfolgenden Unterabschnitt genauer diskutiert werden.

4.2 Aufbau des Experimentes

4.2.1 Die verwendete Simulationsumgebung

In der wissenschaftlichen Literatur zum autonomen Fahren mit *Reinforcement Learning*-Agenten werden die Agenten zumeist in einer synthetischen Simulationsumgebung angeleert und getestet. [WJW18] [JdT⁺18] [EAPY17] [EAPY16] [MBM⁺16] [WHW⁺17] Eine Ausnahme, wie in Kapitel 2 schon beschrieben, stellt dabei die Publikation von Kendall et al. [KHJ⁺18] dar, bei der auch ein reales Fahrzeug durch einen DDPG-Agenten gesteuert wird. Grund für die überwiegende Nutzung von Simulationsumgebungen ist die Bewahrung des Fahrzeugs vor physischem Schaden. [WJW18] [JdT⁺18] Die von der Umgebung verwendete Belohnungsfunktion spielt eine entscheidende Rolle für das Fahrverhalten des Agenten und wird zumeist handisch erstellt. Die Modellierung einer Belohnungsfunktion, die eine gute Quer- und Längsführung belohnt, ist weitaus weniger komplex als die Modellierung einer Belohnungsfunktion, die sämtliche Aspekte eines korrekten Fahrverhaltens im öffentlichen Verkehr modelliert. Folglich lassen Kendall et al. nur eine einfache Quer- und Längsführung auf einem privaten Feldweg von ihrem Agenten erlernen. Gleichwohl stellt diese Einschränkung keine Minderung hinsichtlich der Komplexität und Verwertbarkeit von *Reinforcement Learning*-Agenten gegenüber anderen Ende-Zu-Ende-Ansätzen wie dem *Behavioral Reflex*-Ansatz aus [BTD⁺16] dar, da diese oft ebenfalls höchstens die Quer- und Längsführung erlernen. In Unterabschnitt 4.2.4 werden die im Experiment verwendeten Belohnungsfunktionen dargestellt und erklärt.

Infolge der obigen Argumentation und den in Abschnitt 4.1 dargestellten Gründen, wurde sich bei der Suche nach einer geeigneten Simulationsumgebung auf Umgebungen fo-

4 Experiment

kussiert, die einen starken Fokus auf die Quer- und Längsführung des Fahrzeugs legen. Auf Fahraspekte wie die Beachtung von Ampelsignalen, Verkehrsschildern, Fahrzeugabständen oder Fußgängern wird dagegen kein Fokus gelegt. Rennsimulatoren entsprechen dieser Anforderung. Auch in der Literatur zum autonomen Fahren mit *Reinforcement Learning*-Agenten werden zumeist Rennsimulatoren genutzt. Ein beliebte und langjährig erprobte Simulationsumgebungen in der Forschung zum Maschinellen Lernen ist dabei das Computerspiel *The Open Racing Car Simulator* (TORCS). [WEG⁺15] [WJW18] [LZZC18] [EAPY16] [MBM⁺16] Aus Gründen der Vergleichbarkeit dieser Arbeit mit anderen Arbeiten und Publikationen, der beliebten Nutzung von TORCS im Umfeld der Forschung zur künstlichen Intelligenz [LCL13] [LZZC18] und einer weit entwickelten Agenten-Schnittstelle zu TORCS, die in Unterabschnitt 4.2.2 noch beschrieben werden wird, wird TORCS in der vorliegenden Arbeit als Simulationsumgebung für die *Reinforcement Learning*-Agenten genutzt. Quelloffene, frei verfügbare Alternativen zu TORCS sind etwa Duckitown [PTA⁺17] oder CARLA [DRC⁺17].

TORCS ist ein modularer, quelloffener und zeitdiskreter Rennsimulator, der Fahrzeugmechaniken, wie die Federung, Fahrzeugkinematiken, wie die Beschleunigung und Fluidodynamiken, wie das *Slipstreaming*, abbildet. [WEG⁺15] Abbildung 4.1 zeigt einige Impressionen aus TORCS. In der vorliegenden Arbeit wird mit zwei unterschiedlichen Spielmodi von TORCS gearbeitet. Im *Practice*-Modus werden die *Reinforcement Learning*-Agenten die Quer- und Längsführung des Fahrzeugs als alleinig auf der Strecke befindliches Fahrzeug lernen und testen. Im *Quick Race*-Modus werden die Agenten die Quer- und Längsführung neben anderen auf der Strecke befindlichen Fahrzeugen erlernen. Diese anderen Fahrzeuge werden direkt von der TORCS-Spielmechanik gesteuert. Da das Interaktionsmodell mit anderen Fahrzeugen durch simple Belohnungsfunktionen ausgestaltet wird und damit abzusehen ist, dass diese in keinem Fall den Ansprüchen einer echten Fahrzeuginteraktion genügt, liegt der Fokus in den meisten Experimenten auf dem *Practice*-Modus.



Abbildung 4.1: Dargestellt sind Bildausschnitte aus TORCS. Von links nach rechts: Auswahlbildschirm für die Strecken, *Practice*-Modus in Erste-Person-Perspektive und *Quick Race*-Modus mit Konkurrenten in Dritte-Person-Perspektive. Neben den genannten Spielmodi, gibt es noch einen Turniermodus und einen *Challenge*-Modus. Bildquellen: Eigens erstellt.

In TORCS stehen nativ insgesamt 42 Rennstrecken zur Verfügung. Diese lassen sich grob in drei Kategorien unterteilen. In die erste Kategorie fallen alle Rennstrecken, die asphaltiert und nicht achsensymmetrisch sind. Dieser Rennstreckentypus ähnelt dem öffentlichen Straßennetz und vielen echten Rennstrecken am meisten. In die zweite Kategorie fallen alle asphaltierten Rennstrecken, die achsensymmetrisch sind. Diese sind formal besonders eingängig, haben keine abrupten Kurven und können, im einfachsten Fall, beispielsweise ein Oval sein. Vertreter dieses Rennstreckentypus ist in TORCS etwa die Michigan-Speedway-Strecke. In eine dritte Kategorie fallen alle *off-road* Strecken, die in TORCS zumeist recht kurz und kurvig sind.

Aus den 42 zur Verfügung stehenden Rennstrecken in TORCS wurden drei ausgewählt, auf denen die Agenten angelernt und getestet werden. Entscheidungsgrundlage war dabei die Komplexität der Rennstrecke hinsichtlich der Anzahl der Kurven, der Kurvenradien und eine möglichst heterogene Zusammenstellung von Strecken mit verschiedenen Charakteristi-

ka. Alle ausgewählten Stecken gehören zur ersten Kategorie der asphaltierten, nicht achsen-symmetrischen Rennstecken, da die Streckencharakteristika dieser am ehesten jener echter Rennstrecken oder öffentlicher Verkehrswege entsprechen. Die drei ausgewählten Rennstrecken sind in der Folge erklärt und abgebildet:

CG Speedway 1 Dies ist eine 2.057 m Meter lange Rennstrecke. Sie weist ausschließlich länglich gezogene Kurven und Geraden auf und sollte damit für einen Agenten schnell erlernbar sein.

E-Road Diese Rennstrecke ist mit 3.260 m, vielen engen Kurven und einem vielfältigen Höhenprofil deutlich anspruchsvoller als die Strecke CG Speedway. Ein Agent sollte hier länger zum Erlernen der komplexen Streckengegebenheiten brauchen, als auf der Strecke CG Speedway. Das Erlernte sollte aber zu besseren Testergebnissen führen.

Forza Mit 5.784 m ist Forza die längste der ausgewählten Rennstrecken und weist auch die längsten Geraden auf. Es wird deshalb erwartet, dass ein Agent hier die höchsten Geschwindigkeiten erreichen kann.

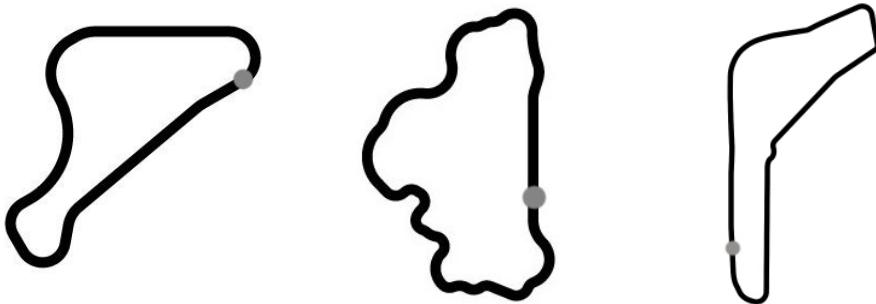


Abbildung 4.2: Dargestellt sind die drei für das Experiment ausgewählten TORCS-Rennstrecken. Von links nach rechts: CG Speedway 1, E-Road und Forza. Der graue Punkt markiert die Startposition des Fahrzeuges zu Beginn der Fahrt. Die Fahrtrichtung ist bei CG Speedway und E-Road gegen den Uhrzeigersinn und bei Forza im Uhrzeigersinn. Alle dargestellten Rennstrecken fallen in die Kategorie der asphaltierten, nicht achsensymmetrischen Rennstrecken. Bildquellen: [WEG⁺19b].

4.2.2 Das verwendete *Reinforcement Learning*-Framework

Nachdem in Unterabschnitt 4.2.1 die Verwendung der Rennsimulation TORCS für das Experiment dieser Arbeit begründet wurde, soll an dieser Stelle die Nutzbarmachung dieser Simulationsumgebung erklärt werden. TORCS ist eine allein lauffähige Applikation bei der die Fahrzeugbots, egal ob programm- oder spielergesteuert, zur Laufzeit in den von TORCS allozierten Hauptspeicherbereich geladen werden. [LCL13] Nachteil dieser Architektur ist die Möglichkeit des direkten Zugriffes der initialisierten Fahrzeugbots auf Datenstrukturen, die Auskunft über die genauen Streckendaten und Konkurrentendaten geben. Damit fehlt eine strikte Trennung zwischen Simulationsumgebung und Fahrzeugbotlogik in TORCS. [LCL13] Loiacono et al. beschreiben in [LCL13] eine Architektur, die diese Nachteile überkommt und TORCS für die Verwendung in der Forschung und in Wettbewerben nutzbar macht. Die Architektur von Loiacono et al. sieht eine Auftrennung von TORCS in einen Server- und einen Client-Teil vor, bei der der Client einen intelligenten Agenten implementiert und über einen UDP-Socket mit dem Server kommuniziert. Der Fahrzeugbot auf dem Server führt dann die Befehle des auf dem Client befindlichen intelligenten *Reinforcement Learning*-Agenten aus. Alle 20 ms schickt der Server ein UDP-Datagram mit Informationen über die neusten

Sensordaten der dem Agenten zugeordneten Botinstanz und erwartet innerhalb von 10 ms eine Antwort über die auszuführenden Aktionen der Botinstanz vom Agenten. Übermittelte Sensordaten des Servers können etwa Geschwindigkeit oder Abstand zum Fahrbahnrand des Fahrzeugbots sein. Zu übermittelnde Aktionen des Agenten können etwa die Quer- und Längsführung des Fahrzeuges sein. Auf die zur Verfügung stehenden Sensoren und Aktoren wird in Unterabschnitt 4.2.3 spezifisch eingegangen. Abbildung 4.3 zeigt graphisch den Aufbau der von Loiacono et al. entwickelten TORCS-Architektur. Diese bietet den Vorteil einer strikten Trennung von Simulationsumgebung und Agenten. Auch ermöglicht diese Architektur die Nutzung verschiedener Programmiersprachen für die Agenten-Entwicklung auf dem Client, da durch die zur Verfügung stehenden Sensoren und Aktoren eine einheitliche Schnittstelle für die UDP-Datagramme definiert ist. [LCL13] Ein solcher Client für die Programmiersprache Python ist das Programm `snakeoil3_gym.py` von Edwards [Edw17], das, mittels UDP-Datagrammen, Informationen mit dem TORCS-Server austauscht. Der TORCS-Server wiederum ist ein allein lauffähiges Programm aus [LCL13], das vom Client aufgerufen wird und in der Folge UDP-Server-Sockets öffnet.

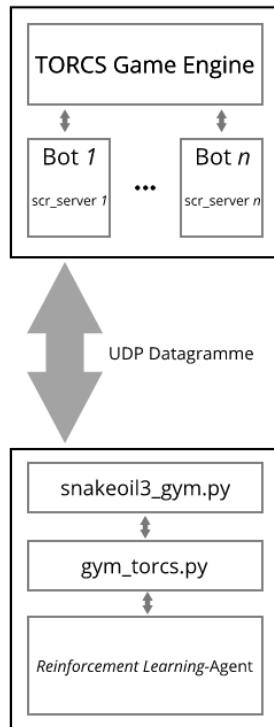


Abbildung 4.3: Dargestellt ist die in der vorliegenden Arbeit verwendete Softwarearchitektur der Schnittstelle zu TORCS. Durch die Darstellung mehrerer serverseitiger Fahrzeugbots soll durch obige Abbildung betont werden, dass diese Architektur auch für Multiagentensystem geeignet ist. In dieser Arbeit wird allerdings nur ein Fahrzeugbot durch den Agenten gesteuert. Die UDP-Kommunikation des Client mit dem Server wird mit dem Programm `snakeoil3_gym.py` durchgeführt. Die OpenAI-gym-konforme Schnittstelle wird dem Agenten über das Programm `gym_torcs.py` bereitgestellt. Bildquelle: Eigens erstellt.

Obgleich die UDP-Socket-Kommunikation zwischen der TORCS-Simulationsumgebung und dem Agenten durch Nutzung des Programmes `snakeoil3_gym.py` bereits abstrahiert wird, lässt sich die Kommunikation zwischen Agent und Simulationsumgebung noch weiter vereinfachen und schablonisieren. Hierfür wird das Programm `gym_torcs.py` von [Yos17] genutzt, das eine OpenAI-gym-konforme Schnittstelle zwischen `snakeoil3_gym.py` und dem

Agenten schafft. So ist eine reine Fokussierung auf die Algorithmenentwicklung in Unterabschnitt 4.2.8 möglich und der resultierende Programmcode ist befreit von den Spezifika der Kommunikation mit TORCS. So könnte der Programmcode später leichter auf andere Simulationsumgebungen portiert werden.

OpenAI ist eine 2015 im kalifornischen San Francisco gegründete Non-Profit-Organisation, die künstliche Intelligenzen erforscht und dabei die sozial-gesellschaftlichen Folgen dieser nicht außer Acht lässt. [Ope15] Von OpenAI wurde im August 2016 die quelloffene Python-Bibliothek `gym` veröffentlicht, die *Reinforcement Learning*-Agenten eine einheitliche Schnittstelle hinzu verschiedenen Umgebungen anbietet. In `gym` enthaltene Umgebungen sind etwa die Cart-Pole-Umgebung von Sutton und Barto [SB17, S. 56], die Mountain-Car-Umgebung von Moore [Moo91, S. 4-4] oder die aus [MKS⁺13] bekannten Atari-2600-Umgebungen. Die `gym`-Umgebungen zeichnen sich durch einen einfachen und standardisierten Aufbau der Agent-Umgebung-Interaktion aus, die dem im Abschnitt 3.1 beschriebenen Interaktionsschema entspricht. Diese Vereinheitlichung ermöglicht eine Fokussierung auf die Entwicklung neuer *Reinforcement Learning*-Algorithmen und erhöht die Vergleichbarkeit von verschiedenen *Reinforcement Learning*-Agenten, da individuelle Unterschiede in der Implementierung einer Schnittstelle vermieden werden. Für eine ausführliche Einführung in die OpenAI-`gym`-Schnittstelle wird auf [Lap18, S. S. 30 ff.] verwiesen. Als Alternativen zur `gym`-Bibliothek sind das *Arcade Learning Environment* [BNVB13] oder das garage-Framework [DCH⁺16] zu nennen, das wiederum kompatibel mit der `gym`-Bibliothek ist. In Fachbüchern zu angewandtem *Reinforcement Learning* wird häufig die `gym`-Bibliothek genutzt. [Lap18] [Rav18]

TORCS ist nicht als Umgebung in der `gym`-Bibliothek enthalten. Das Schnittstellenprogramm `gym_torcs.py` eröffnet einem Agenten für TORCS das gleiche Interaktionsschema, das die `gym`-Bibliothek anderen Agenten für `gym`-Umgebungen eröffnet. Aus diesem Grund wird `gym_torcs.py` in der vorliegenden Arbeit genutzt. Daraus ergibt sich auch, dass die Agenten-Algorithmen in der Programmiersprache Python zu implementieren sind. Aus der Theorie zu DDPG- und DQN-Agenten in Unterabschnitt 3.4.5 und Unterabschnitt 3.3.4 geht hervor, dass Funktionsapproximatoren in Form von neuronalen Netzen genutzt werden. In der vorliegenden Arbeit wird hierfür die Python-Bibliothek `keras` [C⁺15] und, zu einem kleineren Anteil, direkt die Python-Bibliothek von `tensorflow` [AAB¹⁵] genutzt. Grund für die Auswahl von `keras` als Bibliothek für die zu implementierenden neuronalen Netze ist der hohe Abstraktionsgrad dieser Bibliothek und die sich damit erschließenden Vorteile einer *Rapid Prototyping*-Methodik für den Entwicklungsprozess. Alternativen zu `keras` und `tensorflow` sind etwa `pytorch` [PGC¹⁷] oder `caffe` [JSD¹⁴].

Ein typischer Programmaufbau mit der OpenAI-`gym`-Schnittstelle ist in Algorithmus 5 dargestellt. Aus der `step`-Funktion in Algorithmus 5 wird auch deutlich, welche Informationen die TORCS-Simulationsumgebung dem Agenten auf dem Client bereitstellen muss: Um den Umgebungszustand s und s' geht es im nachfolgenden Unterabschnitt 4.2.3, um die Belohnung r geht es im Unterabschnitt 4.2.4. In Unterabschnitt 4.2.5 geht es um die begründete Herleitung der in diesem Experiment genutzten *Reinforcement Learning*-Algorithmen.

Algorithmus 5 : Programmaufbau für die OpenAI-`gym`-Schnittstelle

```

Initialize an environment with gym.make()-Function ;
for episode = 1,  $M$  do
    Observe initial state  $s$  from env.reset-Function ;
    while True do
        Select action  $a$  according to some policy ;
        Get  $s'$ ,  $r$ , done from step(a)-Function ;
        if done then
            | break;
            |  $s \leftarrow s'$  ;
        end
    end

```

4.2.3 Die verwendete Fahrzeugsensorik und -aktorik

Nachdem in Unterabschnitt 4.2.1 die Verwendung von TORCS begründet und in Unterabschnitt 4.2.2 die Nutzbarmachung dieser Simulationsumgebung beschrieben worden ist, soll in diesem Unterabschnitt auf die für das Experiment zu verwendende Fahrzeugsensorik und -aktorik eingegangen werden. Im vorherigen Unterabschnitt 4.2.2 wurde bereits angesprochen, dass der Server dem Client-Agenten Informationen, nachdem diese durch `gym_torcs.py` in ein gym-konformes Format übertragen worden sind, über den derzeitigen Zustand der Umgebung in Form von Sensordaten bereitstellt. Der Agent reagiert mit Aktionen auf diese Informationen, sendet die wertmäßige Ausgestaltung dieser als UDP-Datagramme über die Programme `gym_torcs.py` und `snakeoil3_gym.py` zum Server, der dann die entsprechenden Aktoren des Fahrzeugbots anspricht. Der Server antwortet wiederum mit den nächsten Informationen s' zum sich, aufgrund der Aktionen, veränderten Umgebungszustand. An dieser Stelle der Arbeit wird begründet und festlegt, welche Sensoren, zur Beschreibung der Umgebungszustände, und welche Aktoren, zur Verwirklichung der Quer- und Längsführung nach Unterabschnitt 4.2.1, im durchzuführenden Experiment genutzt werden sollen.

Insbesondere die Auswahl der Fahrzeugsensorik ist im autonomen Fahren ein häufiger Untersuchungsgegenstand der Forschung. [Sch17, S. 1 ff.] Je nach Automatisierungsgrad des Fahrens, wie in Abschnitt 4.1 beschrieben, werden unterschiedliche und unterschiedlich viele Sensoren genutzt. In Abbildung 4.4 ist exemplarisch die Anbringung verschiedener Sensoren auf und in einem Fahrzeug dargestellt. Ein Sensor kann dabei mehrere Aufgaben übernehmen. Beispielsweise kann ein *Light Detection And Ranging*-Sensor (Lidar) durch die Auswertung der reflektierten Intensität von Laserstrahlen für die Objekterkennung und für das 3D-Mapping der Umgebung genutzt werden. Schoettle vergleicht in [Sch17, S. 6 ff.] Lidar-Sensorik, Radar-Sensorik und Bildkamera-Sensorik für verschiedene Aufgabenstellungen des autonomen Fahrens, wie der Objekterkennung und -klassifikation. Er führt an, dass erst im Zusammenspiel mit einer Fahrzeug-Zu-Fahrzeug-Kommunikation alle Aufgaben, die für ein autonomes Fahren der SAE-Stufe Fünf notwendig sind, ausreichend zufriedenstellend erfüllt werden können. Wie in Kapitel 2 dargestellt, ist *Reinforcement Learning* ein Ende-Zu-Ende-Ansatz für das autonome Fahren. Praktisch bedeutet dies, dass keine Aufteilung in Abstraktionsschichten, wie beim *Mediated Perception*-Ansatz, stattfindet. Als Eingabe erhält der *Reinforcement Learning*-Agent Sensordaten, entwickelt in der Folge durch *Try-And-Error* eine ertragreiche *Policy*, lernt dabei implizit was beispielsweise die Fahrbahnmarkierung für Auswirkung auf das Fahren hat und nutzt die erlernte *Policy* dann, um direkt von Sensordaten auf Aktionen zu schließen. An der Notwendigkeit verschiedener Sensoren, für eine umfängliche Beschreibung der Umgebung, ändert sich dabei nichts. Lediglich die Auswertung der Messdaten wird abstrahiert. Ein spannender Untersuchungsgegenstand ist ein Vergleich der Performanzergebnisse eines Agenten hinsichtlich der verwendeten Sensoren. Da TORCS virtuelle Sensoren nutzt, die nur oberflächlich mit echter Sensorik verglichen werden kann, scheint eine solche Untersuchung mit echter Fahrzeugsensorik aber sinnvoller.

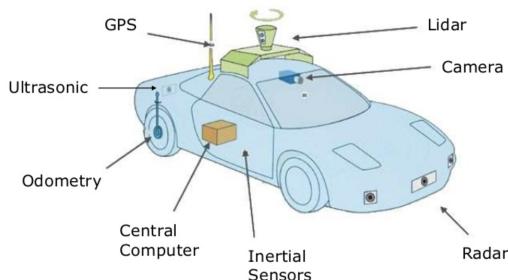


Abbildung 4.4: Dargestellt ist eine mögliche Ausgestaltung der Sensorik eines autonomen Fahrzeuges. Diese kann etwa Lidar, Radar, GPS und Inertialsensoren umfassen. [WJW18] [EAPY16] Bildquelle: [Hon16].

In Tabelle 4.1 sind die im Experiment verwendeten virtuellen Sensoren von TORCS genannt und erklärt. Bei der Auswahl dieser Sensoren wurde sich grob an Wang et al. [WJW18] und Lau [Lau16] orientiert. Auch eigene Experimente flossen in die Entscheidungsfindung mit ein. Die in der Tabelle beschriebenen Sensoren sind die einzigen Informationsquellen, die dem TORCS-Server zur Beschreibung der Umgebung zur Verfügung stehen und genau jene, die vom Agenten zur Auswahl der Aktionen genutzt werden. Manche dieser Sensorwerte werden im Unterabschnitt 4.2.4 auch zur Erstellung der Belohnungsfunktionen verwendet.

Die im Experiment genutzten Sensoren umfassen einige Sensoren, die Auskunft über den Zustand des Fahrzeugs geben, wie der SpeedX-Sensor, der RPM-Sensor oder der WheelSpinVel-Sensor. Andere Sensoren geben Auskunft über den Zustand der Umgebung des Fahrzeugs, wie der Track-Sensor oder der Opponents-Sensor. Zusammen müssen diese Sensoren in der Lage sein, dem Agenten ein adäquates Abbild der Wirklichkeit zu liefern. Die Track- und Focus-Sensoren entsprechen in etwa niedrigauflösenden, auf maximal 180 Grad beschränkten, Lidar-Sensoren. Insbesondere die Position des Fahrzeugs, üblicherweise durch GPS- und Lidar-Sensorik gemessen, wird im Experiment vernachlässigt. Grund dafür ist die von Loiacono et al. entworfene TORCS-Architektur, die diese Informationen nicht direkt zur Verfügung stellt. Anders als in [WHW⁺17] oder [JdT⁺18] wird in dieser Arbeit kein Kamerasensor genutzt, da für diesen zumeist *Convolutional*-Schichten in das neuronale Netz eingebunden werden müssen. In Abbildung 4.5 wird die Wahrnehmung des Agenten von der Umgebung dargestellt.

Sensor	Werte	Einheit	Beschreibung
Angle	$[-\pi; \pi]$	[rad]	Winkel zwischen Fahrzeugrichtung und der Streckenrichtung.
Opponents	$[0; 200]$	[m]	Vektor mit 36 Werten, die jeweils die Distanz zum nächsten Konkurrenten, innerhalb von 200 m, angeben. Der Sensor deckt den gesamten Bereich um das Fahrzeug herum ab.
Track	$[0; 200]$	[m]	Vektor mit 19 Werten, die jeweils die Distanz vom Fahrzeug zur Fahrbahngrenze, innerhalb von 200 m, angeben. Ein Bereich von 180 Grad vor dem Fahrzeug wird dabei mit einer Auflösung von 10 Grad abgetastet.
Focus	$[0; 200]$	[m]	Vektor mit 5 Werten, die jeweils die Distanz vom Fahrzeug zur Fahrbahngrenze, innerhalb von 200 m, angeben. Anders als der Track-Sensor, betrachtet der Focus-Sensor einen Bereich von nur fünf Grad. Der betrachtete Bereich kann aber erlernt werden.
TrackPos	$(-\infty; \infty)$	Dim.-los	Distanz zwischen der Fahrzeugposition und der Fahrbahnmitte. Null wenn Fahrzeug in Fahrbahnmitte. Normalisierte Größe.
SpeedX	$(-\infty; \infty)$	[km/h]	Geschwind. des Fahrzeuges in X-Richtung.
SpeedY	$(-\infty; \infty)$	[km/h]	Geschwind. des Fahrzeuges in Y-Richtung.
SpeedZ	$(-\infty; \infty)$	[km/h]	Geschwind. des Fahrzeuges in Z-Richtung.
WheelSpinVel	$[0; \infty)$	[rad/s]	Vektor mit vier Werten, die jeweils die Drehgeschwindigkeit eines Rades angeben.
RPM	$[0; \infty)$	[rpm]	Drehzahl der Kurbelwelle des Motors.

Tabelle 4.1: Diese Tabelle enthält die Erläuterungen der verwendeten Sensoren, ihrer Wertebereiche und Einheiten. Die Track- und Focus-Sensoren ähneln dabei einem Lidar-Sensor. Diese Sensoren sind auch für die Erstellung der Belohnungsfunktion wichtig. \mathcal{S} entspricht dem Kreuzprodukt dieser Sensoren und ist damit sehr mächtig. In Anlehnung an [LCL13].

4 Experiment

Ausdrücklich zu betonen ist, dass es sich bei den in der Tabelle 4.1 erklärten Sensoren um, von der TORCS-Simulationsumgebung bereitgestellte, ideale Sensoren handelt. Das heißt, sie unterliegen keinem Rauschen und werden nicht von Umwelteinflüssen, wie schlechtem Wetter oder Abnutzung, beeinflusst. [Sch17, S. 6] Dies mindert zwar den Realismusgrad, jedoch zeigen Kendall et al. in [KHJ⁺18] in einem *Bootstrapping*-Ansatz, dass ein Agenten-Algorithmus zunächst in einer Simulationsumgebung angelernt werden kann, um dann auf einem realen Fahrzeug schneller eine ertragreiche *Policy* zu erlernen. Für das Anlernen eignen sich hyperrealistische Simulationsumgebungen, wie etwa CARLA, besonders. Diese zeichnen sich nicht nur durch bessere Grafik-Texturen aus, was vorwiegend bei der Nutzung eines Bildkamera-Sensors zum Tragen käme, sondern auch durch einstellbare Wetterbedingungen und eine realitätsnahe Umsetzung von Sensoren wie dem Lidar-Sensor. [DRC⁺17] Aufgrund des erheblichen Hardwareaufwandes derlei Simulationsumgebungen und der in Unterabschnitt 4.2.1 angeführten Gründe, wurde auf die Nutzung hyperrealistischer Simulationsumgebungen in der vorliegenden Arbeit verzichtet.

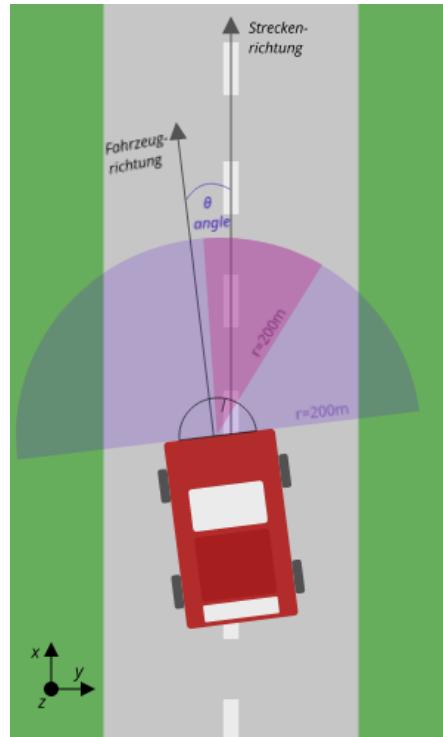


Abbildung 4.5: Dargestellt ist einige Sensorik des im Experiment verwendeten Fahrzeuges. Der lila Halbkreis stellt den Track-Sensor aus Tabelle 4.1 dar. Der rosa Ausschnitt den Focus-Sensor. Der Angle-Sensor gibt den Winkel ϑ zwischen Strecken- und Fahrzeugrichtung an. Durch die Abbildung wird die Wahrnehmung des Agenten in der Umgebung erkenntlich, der diese durch numerische Werte erfährt. Die Elemente der Abbildung sind zueinander nicht maßstabsgerecht. Bildquelle: Eigens erstellt.

In Tabelle 4.2 sind die verwendeten Aktoren des TORCS-Agenten dargestellt und erklärt. Nach Unterabschnitt 4.2.1 soll der Agent die autonome, also unabhängige, Quer- und Längsführung des Fahrzeugs erlernen. Hierfür werden die drei Aktoren *Steering* (dt. Lenkung), *Acceleration* (dt. Beschleunigung) und *Brake* (dt. Bremse) genutzt. Neben den in Tabelle 4.2 aufgeführten Aktoren, stellt die verwendete Schnittstelle zu TORCS, `gym_torcs.py`, noch weitere Aktoren bereit. Zu diesen zählen die Kupplung, der Getriebegang (engl. *clutch* und *gear*) oder der Richtungswinkel des Focus-Sensors. [LCL13] Diese vom Agenten nicht genutzten Aktoren werden dann von TORCS automatisch bedient. Eine Einbindung weite-

rer Sensoren und Aktoren in die in dieser Arbeit implementierten Agenten-Algorithmen ist denkbar, jedoch für die in Kapitel 1 formulierten Ziele nicht zwingend notwendig, da es in dieser Arbeit mehr um die Darstellung der methodischen Vorgehensweise einer Evaluation von *Reinforcement Learning*-Algorithmen im Kontext autonomer Fahrzeuge geht, als um die praktische Entwicklung eines hochoptimierten Agenten-Algorithmus.

Aktor	Werte	Beschreibung
Steering	$[-1; 1]$	Wert für die Querführung des Fahrzeuges. Der Wert -1 entspricht einem vollen Linksschlag des Lenkrades, der Wert 1 entspricht einem vollen Rechtsenschlag.
Acceleration	$[0; 1]$	Wert für die vorwärtsgerichtete Längsführung des Fahrzeuges. Der Wert 0 bedeutet keine positive Beschleunigung, der Wert 1 bedeutet volle positive Beschleunigung.
Brake	$[0; 1]$	Wert für die rückwärtsgerichtete Längsführung des Fahrzeuges. Der Wert 0 bedeutet keine negative Beschleunigung, der Wert 1 bedeutet volle negative Beschleunigung. Dieser Aktor wird als Bremse genutzt, ein Rückwärtsfahren wird unterbunden.

Tabelle 4.2: Diese Tabelle enthält die Erläuterung der verwendeten Aktoren und ihrer Wertebereiche. Die reellen Intervalle der Aktoren können problemlos mit dem, aus Unterabschnitt 3.4.5 bekannten, DDPG-Agenten-Algorithmus umgesetzt werden.

Das in TORCS verwendete Fahrzeugmodell hört auf den Namen `car1-trb1` und wird in dieser Arbeit genutzt, da es standardmäßig in der verwendeten Agenten-Schnittstelle zu TORCS vorgegeben wird und somit am ehesten für Vergleiche mit anderen Arbeiten dienen kann. [LCL13] Es hat eine Leistung von 405kW bei einem Gewicht von 1150kg und ist in Abbildung 4.6 dargestellt. Mit diesen Leistungsdaten zählt es zu den besonders leistungsstarken TORCS-Fahrzeugmodellen.



Abbildung 4.6: Dargestellt ist das in der Simulationsumgebung TORCS verwendete Fahrzeugmodell `car1-trb1`. Es gilt in TORCS als besonders leistungsstark. Bildquelle: [WEG⁺19a].

4.2.4 Die verwendeten Belohnungsfunktionen

Anders als Definition 3.2.10 zu Markow-Entscheidungsprozessen suggeriert, wird die Belohnungsfunktion einer Umgebung im *Reinforcement Learning* häufig nicht direkt vorgegeben und muss daher händisch erstellt werden. So auch im Fall der TORCS-Simulationsumgebung, bei der, nachdem die Sensordaten des neuen Zustandes s' durch den Server erhalten wurden, das Client-Programm `gym_torcs.py` die Belohnung für den Agenten berechnet und, nach Algorithmus 5, diesem mitsamt s' zur Verfügung stellt. Die Ausgestaltung der Belohnungsfunktion obliegt also dem Algorithmiker und wird, in diesem Falle, nicht durch die Umgebung vorgeschrieben. In Unterabschnitt 4.2.3 wurde bereits erwähnt, dass der Zustand der Umgebung s bzw. s' genutzt werden kann, um eine eigene Belohnungsfunktion aufzustellen. Dies kann etwa über der Penalisierung von Sensorwerten, die aus einem vorher bestimmten Normbereich fallen, geschehen. Auch ist denkbar, dass bestimmte Sensorwerte direkt

4 Experiment

als positive Belohnung genutzt werden, also ein linearer Zusammenhang mit der Belohnung besteht. Wie genau die Belohnungsfunktionen in dieser Arbeit berechnet werden, wird in diesem Unterabschnitt erklärt. Dabei ist zu erwähnen, dass der Zustand der Umgebung, ausgedrückt durch die Sensordaten des Fahrzeugbots in der TORCS-Simulationsumgebung, hier als Tupel $b = (s_1, s_2, \dots, s_n)$ aufgefasst wird, bei der jedes Tupelement einen Sensorwert repräsentiert. Beispielsweise kann s_1 der reelle Sensorwert der Geschwindigkeit, SpeedX, s_2 die Drehzahl des Motors, RPM, und s_3 ein boolescher Sensorwert sein, der angibt, ob der Fahrzeugbot die Fahrbahnmarkierung überfahren hat. Für Sensoren wie den Track-Sensor ist das Tupelement ein Vektor $\mathbf{s}_i \in \mathbb{R}^m$, $1 \leq i \leq n$, der nach Tabelle 4.1 neunzehn Vektor-elemente besitzt. Diese Notation weicht leicht von der aus Definition 3.2.10 ab und wird im Pseudocode im Algorithmus 6 genutzt.

Die Belohnungsfunktion ist intensiver Forschungsgegenstand im Bereich des autonomen Fahrens mit *Reinforcement Learning*-Agenten. [WHW⁺17] [JdT⁺18] [TSK⁺19] Grund dafür ist, dass die Belohnungsfunktion das Verhalten des Agenten entscheidend beeinflusst. [SB17, S. 6] Ist die Belohnung hoch, wird die zuvor vom Agenten ausgeführte Aktion verstärkt. Ist die Belohnung niedrig, wird die *Policy* beim nächsten Auftreten des Zustandes eher eine andere Aktion auswählen. Dieser einfache Zusammenhang ist, mitsamt den Gleichungen von Bellman, charakteristisch für das *Reinforcement Learning* und alle in Kapitel 3 erklärten Algorithmen setzen diesen Zusammenhang von Zustand, Aktion und Belohnung auf die ein oder andere Weise algorithmisch um. Deshalb ist es wichtig in Umgebungen, die keine Belohnungsfunktion vorgeben, eine sinnvolle Belohnungsfunktion zu definieren. In der vorliegenden Arbeit werden, vor dem Hintergrund der in Kapitel 1 formulierten Ziele, drei unterschiedliche Belohnungsfunktionen hergeleitet, empirische Daten aus dem Experiment zu diesen erhoben und anschließend, hinsichtlich bestimmter Performanzmetriken des Agenten, diskutiert und bewertet.

Alle drei Belohnungsfunktionen werden nach den in Algorithmus 6 beschriebenen Regeln berechnet. Die Berechnung von $\mathcal{R}(b, a)$ wird dabei, je nach verwendeter Belohnungsfunktion, unterschiedlich ausgestaltet. Die Penalisierung von Ereignissen, wie der Überschreitung der Fahrbahnmarkierung, wird dabei durch eine **if-else**-Verzweigung abgebildet. Die genauen Berechnungsvorschriften der drei im Experiment untersuchten Belohnungsfunktionen werden im Laufe dieses Unterabschnittes genannt und erklärt.

Algorithmus 6 : Pseudocode für die Berechnung der Belohnungsfunktion

```

Input : the environmental state  $b$  of TORCS
Result : the reward  $r$  at time step  $t$ 
Calculate  $r = \mathcal{R}(b, a)$  ;
 $hasOvertaken = s_1 = b[0]$  ;  $hasBeenCalled = s_2 = b[1]$  ;
 $hasCollided = s_3 = b[2]$  ;  $IsOutOfTrack = s_4 = b[3]$  ;
 $hasStoppedMoving = s_5 = b[4]$  ;  $IsDrivenBackwards = s_6 = b[5]$  ;
if  $hasOvertaken$  then
|  $r += 150$  ;
else if  $hasBeenCalled$  then
|  $r -= 150$  ;
else if  $hasCollided$  then
|  $r -= 50$  ;
else if  $IsOutOfTrack$  then
|  $r -= 50$  ;
else if  $hasStoppedMoving$  then
|  $r -= 10$  ;
else if  $IsDrivenBackwards$  then
|  $r -= 50$  ;
else
| ;
end
```

Aus Algorithmus 6 werden die möglichen Terminierungsbedingungen einer Episode deutlich. Beginnt eine Rennfahrt in TORCS, dauert die Episode im durchzuführenden Experiment so lange an, bis der Fahrzeugbot entweder die Fahrbahn verlassen hat, bis der Fahrzeugbot keinen bedeutenden Fortschritt mehr macht, etwa weil er sich an einer Leitplanke festgefahren hat, oder bis der Fahrzeugbot rückwärts fährt. Da es durchaus vorkommen kann, dass der Agent eine sehr gute *Policy* für eine Rennstrecke erlernt hat und keiner der genannten Terminierungsbedingungen eintritt, wird eine Episode nach spätestens 3.000 Zeitschritten, d.h. nach 3.000 Agent-Umgebung-Interaktionen in der Form von 3.000 Aufrufen der `gym.step()`-Funktion, beendet und eine neue Episode beginnt. Die in dieser Arbeit genutzten Terminierungsbedingungen wurden experimentell festgelegt. El Sallab et al. untersuchen in [EAPY16] einen Zusammenhang zwischen Konvergenzzeit hinzu π_* und den verwendeten Terminierungsbedingungen während des Trainings. Demnach führen wenige Terminierungsbedingungen zu einer Konvergenz nach wenigen Episoden. In diesem Experiment werden alle Terminierungsbedingungen genutzt, da El Sallab et al. nur die Anzahl der Episoden betrachten, nicht aber die Anzahl der Zeitschritte.

Die erste im Experiment zu untersuchende Belohnungsfunktion $\mathcal{R}_1(b, a)$ wird, in grober Form, Wang et al. [WJW18] entnommen, die wiederum auf der Belohnungsfunktion von Lau [Lau16] aufbaut. Diese ist folgendermaßen definiert:

$$r = \mathcal{R}_1(b, a) = \text{SpeedX} \cos(\vartheta) - |\text{SpeedX} \sin(\vartheta)| - \text{SpeedX} |\text{TrackPos}| \quad (4.1)$$

Diese Belohnungsfunktion erklärt sich leicht durch Betrachtung von Abbildung 4.5. Die X-Achse entspricht dabei dem SpeedX-Sensor aus Tabelle 4.1 und verläuft in Streckenrichtung. Werden Streckenrichtung und Fahrzeugrichtung als Seiten eines Dreiecks angesehen, ergibt sich ein rechtwinkliges Dreieck, bei dem der SpeedX-Sensor die Länge der Hypotenuse ausdrückt. Die Länge der Ankathete entspricht dem Betrag des Vektors in ϑ -Richtung des Fahrzeugs. Ist dieser Betrag groß, wird der Fahrzeugbot weit fahren, was als erstrebenswert angesehen wird. Durch einfache Trigonometrie können also die Terme von Gleichung 4.1 hergeleitet werden: Es gilt

$$\cos(\vartheta) = \frac{\text{Ankathete}}{\text{Hypotenuse}} = \frac{\text{Längsweg}}{\text{SpeedX}} \quad (4.2)$$

$$\cos(\vartheta) \text{SpeedX} = \text{Längsweg} \quad (4.3)$$

und

$$\sin(\vartheta) = \frac{\text{Gegenkathete}}{\text{Hypotenuse}} = \frac{\text{Querweg}}{\text{SpeedX}} \quad (4.4)$$

$$\sin(\vartheta) \text{SpeedX} = \text{Querweg}. \quad (4.5)$$

Der Querweg, bestimmbar durch Gleichung 4.5, soll möglichst minimal sein, d.h. das Fahrzeug soll möglichst mittig der Fahrbahn fahren. Deshalb wirkt der Querweg in Gleichung 4.1 negativ auf die Belohnung. Der dritte Term in Gleichung 4.1 verstärkt dies mittels des TrackPos-Sensors. Die in Gleichung 4.1 vorgestellte Belohnungsfunktion wird im durchzuführenden Experiment primär genutzt. Die zweite zu untersuchende Belohnungsfunktion $\mathcal{R}_2(b, a)$ wird [JdT⁺18] entnommen und ist gegeben durch

$$r = \mathcal{R}_2(b, a) = \text{SpeedX}(\cos(\vartheta) - |\text{TrackPos}|). \quad (4.6)$$

Anders als die Belohnungsfunktion aus Gleichung 4.1 wird in Gleichung 4.6 der Wert des TrackPos-Sensors sofort vom Seitenverhältnis abgezogen und danach mit dem SpeedX-Sensor gewichtet. Die dritte zu untersuchende Belohnungsfunktion $\mathcal{R}_3(b, a)$ ist eigens erdacht, bezieht den Abstand zu anderen Fahrzeugbots mit ein und lautet

$$r = \mathcal{R}_3(b, a) = \mathcal{R}_1(b, a) + \begin{cases} -10 & \text{falls } \min(\text{Opponents}) \leq 7 \\ 0 & \text{andernfalls} \end{cases}. \quad (4.7)$$

Zwar üben die vorgestellten Belohnungsfunktionen unmittelbar Einfluss auf die Quer- und Längsführung des Fahrzeugs aus, jedoch sind die in Algorithmus 6 dargestellten Ereignisse mitsamt Penalisierung ebenso wichtig. In der vorliegenden Arbeit werden ausschließlich die Belohnungsfunktionen \mathcal{R}_1 , \mathcal{R}_2 und \mathcal{R}_3 untersucht. Für eine Untersuchung der Auswirkung der Terminierungsbedingungen wird auf El Sallab et al. [EAPY16] verwiesen. Wichtig ist, dass die Terminierungsbedingungen aus Algorithmus 6 für alle drei Belohnungsfunktionen im Experiment gleich sind, um die Vergleichbarkeit der Erhebungen zu gewährleisten.

4.2.5 Die verwendeten *Reinforcement Learning*-Algorithmen

In dieser Arbeit wird primär der DDPG-Algorithmus aus Unterabschnitt 3.4.5 untersucht, da dieser, als deterministischer *Policy-Gradient*-Algorithmus, eine reellwertige Ausgabe aus der *Policy*-Funktion $\tau(s, \theta)$ hat. Die in Tabelle 4.2 dargestellten Aktionen sind ebenfalls über reelle Intervalle definiert. Somit ist die Verwendung des DDPG-Agenten-Algorithmus naheliegend. Im Gegensatz zum DDPG-Algorithmus kann der bekannte *value-based* DQN-Algorithmus nur auf diskreten Aktionsräumen angewandt werden. Ein Ziel dieser Arbeit ist der Vergleich zwischen diesen beiden Algorithmen. Diese Untersuchung soll den Unterschied zwischen *value-based*- und *policy-based*- bzw. *actor-critic*-Algorithmen herausarbeiten und damit an die aus Kapitel 3 genutzte Gliederung der Theorie des *Reinforcement Learning* anknüpfen. Überdies kommen nur wenig weitere der in dieser Arbeit erklärten Algorithmen für die Untersuchung in einem Experiment in Frage. Einer der infrage kommenden Algorithmen wäre jedoch der in Unterabschnitt 3.4.3 dargestellte *Advantage Actor-Critic*-Algorithmus (A2C). Da der DDPG-Algorithmus allerdings als algorithmische Weiterführung des A2C-Algorithmus geltend gemacht werden kann, wird sich in dieser Arbeit auf den DDPG-Algorithmus als Vertreter der *actor-critic*-Algorithmen fokussiert. Somit werden in der vorliegenden Arbeit der DDPG- und der DQN-Algorithmus implementiert und im Experiment untersucht.

4.2.6 Die verwendeten Performanzmetriken

In Kapitel 3 wurde erklärt, wie ein Agent eine gute, wohl möglich optimale, *Policy* erlernen kann. Im Kontext dieser Arbeit soll ein *Reinforcement Learning*-Agent, der nach Unterabschnitt 4.2.2 als Algorithmus auf dem Client existiert und mit dem TORCS-Server Informationen über Sensor- und Aktordaten austauscht, über mehrere Episoden, die den Terminierungsbedingungen aus Unterabschnitt 4.2.4 unterliegen, stetig verbesserte *Policies* erlernen, um einen möglichst hohen Ertrag zu erzielen. Während zu erwarten ist, dass der Agent in den ersten Episoden die adäquate Quer- und Längsführung des Fahrzeugs noch nicht gut beherrscht und wenig Ertrag erwirtschaftet, lernt der Agent im Laufe der Zeit, durch die Anpassung der Gewichte, immer bessere *Policies*. Nach vielen Episoden wird der Agent die Gewichte so angepasst haben, dass er in verschiedenen Zuständen die richtigen, d.h. ertragreichen, Entscheidungen hinsichtlich der Aktorik treffen kann.

Aus dieser Ablaufbeschreibung des Trainings eines *Reinforcement Learning*-Agenten lassen sich Rückschlüsse über die zu verwendenden Performanzmetriken ziehen. Die aus den Performanzmetriken entstehenden Performanzdaten sind dann Gegenstand der Ergebnisinterpretation des Experiments in Kapitel 5. Überdies zeigen die Performanzmetriken auch auf, wie *Reinforcement Learning*-Algorithmen hinsichtlich ihrer Güte untersucht werden können. Anders als im klassischen *Supervised Learning*, bei dem auf einem Training folgenden Test Performanzdaten mittels Genauigkeitsmetriken erhoben werden [GBC16, S. 417 ff.], spielt im *Reinforcement Learning* die Zeit eine Rolle. [Lap18, S. 1 f.] Ein *Reinforcement Learning*-Agent lernt graduell mehr über die Umgebung kennen, während er versucht immer bessere *Policies* zu erlernen. Übertragen auf das *Supervised Learning*-Szenario, würde ein *Supervised Learning*-Algorithmus dann nach wenigen Trainingsdatensätzen auf einem Testdatensatz evaluiert werden und es würde erwartet werden, dass der *Supervised Learning*-Algorithmus mit der Zeit bessere Ergebnisse auf dem Testdatensatz erzielt. Tatsächlich wird

im *Supervised Learning* aber nur nach Abschluss des Trainings der Testdatensatz zur Performancevaluation des erlernten Modells genutzt. Beim *Cross Validation*-Verfahren wird, zur frühzeitigen Erkennung von *Over-* oder *Underfitting*, keine harte Trennung von Training- und Testdatensatz durchgeführt. [GBC16, S. 120]

Zwecks der Findung geeigneter Performanzmetriken für *Reinforcement Learning*-Agenten, im Kontext des autonomen Fahrens, wurde eine Erhebung über die in der Literatur gebräuchlichen Performanzmetriken durchgeführt, die in Anhang B als Abbildung B.1 einzusehen ist. Nach Abbildung B.1 werden insbesondere die Metriken

- Erreichter Ertrag des durch den Agenten gesteuerten Fahrzeugbots pro Episode,
- Erreichte Distanz des durch den Agenten gesteuerten Fahrzeugbots pro Episode und
- Durchschn. Geschwindigkeit des durch den Agenten gesteuerten Fahrzeugbots pro Episode

für die Auswertung der Performanz eines *Reinforcement Learning*-Agenten im Kontext des autonomen Fahrens in der Literatur genutzt. In algorithmisch-konzeptionellen Arbeiten wie [MKS⁺13] oder [MBM⁺16] wird zusätzlich häufig die Performanz, z.B. als *Score*-Wert in einem Atari-2600-Spiel, eines menschlichen Probanden in der Simulationsumgebung als Vergleichswert erhoben. Die vorliegende Arbeit orientiert sich an den obig aufgelisteten Performanzmetriken aus der Literatur. So werden für alle Experimente Ertrag pro Episode, erreichte Distanz pro Episode und durchschnittliche Geschwindigkeit pro Episode als Performanzmetriken des Agenten genutzt. Für den *Quick Race*-Modus wird außerdem die Anzahl an Kollisionen pro Episode betrachtet. Im Laufe des Trainings des Agenten wird erwartet, dass sich diese Metriken verbessern. Weiterhin wird festgelegt, dass dem Agenten 350 Episoden zum Training zugesprochen werden. Dies sind mehr Episoden, als für das Eintreten des *Policy*-Konvergenzfalles erwartet wird. Außerdem wird das Erkundungsparameter der Agenten so eingestellt, dass nach 350 Episoden keine weitere Erkundung stattfindet. [Lau16] In Abbildung 4.7 ist ein Beispiel einer solchen Performanzmetrik gegeben.

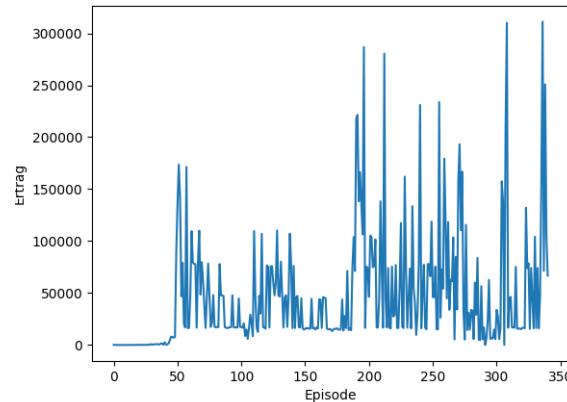


Abbildung 4.7: Dargestellt ist ein Beispiel für die im Experiment generierten Performanzdaten. Auf der X-Achse sind die Episoden abgebildet, auf der Y-Achse der Ertrag. Derlei Darstellungen zeigen in schöner Weise den Lernfortschritt des Agenten, bringen aber auch Nachteile mit. Bildquelle: Eigens erstellt.

Die ausgewählten Performanzmetriken sind nicht frei der Kritik. So ist etwa absehbar, dass der Agent auf der Strecke Forza, die viele Geraden hat, mehr Ertrag einfahren wird als auf der Strecke E-Road, die viele Kurven hat, da die Abweichung von der Fahrbahnmitte in zwei der untersuchten Belohnungsfunktionen penalisiert wird. Auch eignet sich der Ertrag

4 Experiment

pro Episode nicht als Performanzmetrik für den Vergleich der Belohnungsfunktionen, da diese bewusst unterschiedlich viel Ertrag generieren. Zudem ist aus Abbildung 4.7 ersichtlich, dass nicht direkt klar wird, wie der Ertrag auf der Y-Achse zu deuten ist. In Abbildung 4.7 wird beispielsweise der meiste Ertrag kurz nach der dreihundertsten Episode eingefahren. Ob dieser Ertrag nun bedeutet, dass der Agent gelernt hat, wie die Rennstrecke zu umrunden ist, bleibt unklar. Ebenso könnte der hohe Ertrag kurz nach der dreihundertsten Episode nämlich bedeuten, dass der Agent die Streckenmitte erstmalig erreichte. Gleiches gilt für die Distanz-Episode-Metrik. Es fehlt ein Referenzrahmen für die obig genannten Performanzmetriken, um diesen ambivalenten Interpretationsmöglichkeiten entgegenzutreten. Dieser Referenzrahmen wird durch Einführung einer weiteren, denkbar einfachen, Metrik geschaffen. Sie setzt die erreichte Distanz jeder Episode ins Verhältnis mit der Streckenlänge. Hierfür wird festgelegt, dass ein vom Agenten gesteuerter Fahrzeugbot mindestens eine Streckenumrundung schaffen sollte und dies als Referenzrahmen im Experiment dient. Mit diesem wird es dann deutlicher, wie die Erträge und Distanzen pro Episode zu deuten sind. Auf Normalisierung oder z-Transformation der Datenreihen zur besseren Vergleichbarkeit wird hingegen verzichtet, da angenommen werden kann, dass, aufgrund des Lernprozesses des Agenten, die Häufigkeiten der Einzelerträge und -distanzen nicht normalverteilt sind.

Häufig steht in der Literatur, wie [WJW18], [JdT⁺18] oder [EAPY16], zu autonomen Fahren mit *Reinforcement Learning*-Agenten das Training im Mittelpunkt der empirischen Auswertungen. Eine empirische Auswertung zum Test des Erlernten auf unbekannten Strecken findet selten statt. [CKH⁺18] [ZSSH19] Dabei zeichnen sich gute statistische Modelle gerade durch ihre Fähigkeit zur Generalisierung aus. Deshalb soll dieser Aspekt des Lernens in der vorliegenden Arbeit Beachtung finden. Im Kontext des autonomen Fahrens mit *Reinforcement Learning*-Agenten bedeutet dies, dass ein Agent auf einer bestimmten Strecke eine möglichst gute *Policy* erlernt und diese im Anschluss auf neuen Strecken testet. Ist der Ertrag gleich zu Beginn des Testens auf den neuen Strecken hoch, unter Berücksichtigung des obig vorgestellten Referenzrahmens, kann davon ausgegangen werden, dass der Agent generalisieren kann. Während des Testens wird die Umgebung kaum bis gar nicht erkundet. In [CKH⁺18] untersuchen Cobbe et al. die Problematik der Generalisierung im *Reinforcement Learning*. Sie nutzen dabei *Regularization*-Methoden wie *Dropout*, ℓ_2 -*Penalty* oder *Batch Normalization* und zeigen, dass bis ein Agent gute Generalisierungsfähigkeiten hat, dieser länger als früher angenommen trainiert werden muss. Auch die Publikationen von Zhao et al. [ZSSH19] und Zhang et al. [ZVMB18] beschäftigen sich mit der Thematik der Generalisierungsfähigkeit von *Reinforcement Learning*-Agenten. Aus den Erscheinungsdaten dieser Publikationen lässt sich schließen, dass diese Fragestellung erst in den vergangenen ein bis zwei Jahren Aufmerksamkeit in der Forschung fand. Aus diesen Gründen wird im Experiment dieser Arbeit ein Agent auf einer der in Unterabschnitt 4.2.1 dargestellten Strecke angelernt und dann auf den anderen getestet. Die dabei genutzten Performanzmetriken entsprechen denen des Trainings.

4.2.7 Die Untersuchungsgegenstände des Experiments

Gemäß den in Kapitel 1 genannten Zielen dieser Arbeit und unter Zuhilfenahme der in den vorigen Abschnitten dieses Kapitels beschriebenen Rahmenbedingungen, werden zu den folgenden Untersuchungsgegenständen Performanzdaten im Experiment erhoben:

1. Ein DDPG-Agent wird auf den in Unterabschnitt 4.2.1 genannten TORCS-Rennstrecken im *Practice*-Modus trainiert und getestet. Während des Trainings und Testens werden die in Unterabschnitt 4.2.6 beschriebenen Performanzmetriken aufgezeichnet. Auch im *Quick Race*-Modus von TORCS soll der DDPG-Agent auf einer geeigneten Strecke trainiert werden, um das Verhalten des Agenten bei der Interaktion mit anderen Fahrzeugen aufzuzeichnen. Als Belohnungsfunktion wird stets \mathcal{R}_1 genutzt. Es sollen im Experiment so empirische Daten über den Lernfortschritt eines DDPG-Agenten im Kontext des autonomen Fahrens erhoben werden.

2. Die in Unterabschnitt 4.2.6 beschriebenen Performanzmetriken werden unter Verwendung des DDPG-Agenten-Algorithmus auch für die Belohnungsfunktionen \mathcal{R}_2 und \mathcal{R}_3 während des Trainings im *Quick Race*-Modus aufgezeichnet. So sollen empirische Daten über die Auswirkungen verschiedener Belohnungsfunktionen erhoben werden.
3. Der aus Kapitel 3 bekannte DQN-Agenten-Algorithmus soll mit dem DDPG-Algorithmus verglichen werden. Dazu werden die aus Unterabschnitt 4.2.6 bekannten Performanzmetriken zum DQN-Agenten-Algorithmus im *Practice*-Modus durch das Experiment erhoben. Als Belohnungsfunktion wird \mathcal{R}_1 genutzt.

In Kapitel 5 werden die im Experiment gewonnenen Daten, in Form von Graphen zu den Performanzmetriken wie Abbildung 4.7, dann ausgewertet und interpretiert. Dies komplettiert die in Kapitel 1 beschriebene methodische Vorgehensweise einer Evaluation von *Reinforcement Learning*-Algorithmen durch den empirisch induzierten Erkenntnisgewinn.

Auf die Motivation für die Auswahl dieser drei Untersuchungsgegenstände im Experiment wurde in den vorherigen Abschnitten dieses Kapitels indirekt eingegangen. Motivation für die Auswahl des erstgenannten Untersuchungsgegenstandes ist vor allem die Anwendbarkeit des DDPG-Algorithmus für Aktionen mit kontinuierlichem Wertebereich. Dieser wird in der vorliegenden Arbeit einer Evaluation, durch empirische Datenerhebung und Auswertung dieser Daten, unterzogen. Motivation für den zweitgenannten Untersuchungsgegenstand ist die mehrfach genannte, starke Abhängigkeit eines *Reinforcement Learning*-Algorithmus von der Belohnungsfunktion der Umgebung. Da die Belohnungsfunktion in der TORCS-Simulationsumgebung händisch erstellt werden muss, sollen drei verschiedene Belohnungsfunktionen untersucht werden. Motivation des drittgenannten Untersuchungsgegenstandes ist der interessante Vergleich zwischen dem *value-based* DQN-Agenten und dem *actor-critic* DDPG-Agenten im Kontext autonomer Fahrzeuge. Das Fehlen einer solchen Untersuchung in der bisherigen Forschung wirkt verstärkend auf die Motivation.

Für den erstgenannten Untersuchungsgegenstand wird erwartet, dass sowohl Ertrag, erreichte Distanz, als auch die durchschnittliche Geschwindigkeit pro Episode während des Trainings ansteigen. Im *Quick Race*-Modus wird erwartet, dass die Anzahl der Kollisionen mit anderen Fahrzeugen stetig abnimmt, da diese penalisiert werden. Für den Vergleich der Belohnungsfunktionen wird erwartet, dass \mathcal{R}_3 die besten Performanzergebnisse liefert, da diese Belohnungsfunktion die komplexeste der überprüften ist und als Einzige den Opponents-Sensor miteinbezieht. Für den Vergleich des DQN-Agenten mit dem DDPG-Agenten soll besonders auf die Anzahl der Episoden bis zu einer Streckenumrundung nach Unterabschnitt 4.2.6 geachtet werden. Da der DDPG-Agent durch den *Critic*, anders als reine *policy-based*-Algorithmen wie REINFORCE, zusätzlich auch Elemente eines *value-based* Algorithmus mitbringt, wird erwartet, dass der DDPG-Agent bessere Performanzmetriken als der DQN-Agent aufweist. In [SB17, S. 323 f.] stellen Sutton und Barto dar, dass manche Umgebungen eher für *value-based* und manche eher für *policy-based Reinforcement Learning*-Algorithmen geeignet sind, je nachdem ob die *Action-Value*-Funktion oder die *Policy*-Funktion leichter zu approximieren ist. Daher stellt sich die Frage, welcher dieser Agenten-Algorithmen-Typen die Aufgabenstellung der autonomen Quer- und Längsführung eines Fahrzeugs am schnellsten und stabilsten löst. Unter anderem auf diese Frage wird in Kapitel 5 versucht eine Antwort zu geben.

4.2.8 Die Implementation und Experimentdurchführung

Nachdem die Rahmenbedingungen und Untersuchungsgegenstände des Experimentes geklärt wurden, wird an dieser Stelle kurz auf die Implementation der Algorithmen und auf die Experimentdurchführung eingegangen. Dies ist für die Reproduzierbarkeit der Experimentergebnisse unabdingbar. Entgegen der Reihenfolge der Untersuchungsgegenstände in Unterabschnitt 4.2.7, wird hier zunächst der Aufbau des DQN-Agenten erklärt, da dieser nur auf diskreten Aktionsräumen anwendbar ist und somit weiterer Vorarbeit bedarf.

4 Experiment

Nach Unterabschnitt 3.3.4 zeichnet sich der DQN-Algorithmus durch die Nutzung eines Funktionsapproximators, anstatt einer *Q-Table*, die Umsetzung eines *off-policy*-Ansatzes und die Verwendung eines *Experience Replay Memory* zur Trainingsdatendekorrelation aus. Diese Aspekte wurden in Python umgesetzt und der dazugehörige Quellcode befindet sich auf dem dieser Arbeit beiliegenden Datenträger und ist ebenso auf https://github.com/koeller21/ma_code einsehbar. Der Aufbau des neuronalen Netzes des DQN, sowie die verwendeten Hyperparametereinstellungen, sind in Anhang C in Abbildung C.1 und Tabelle C.1 dargestellt. Das dort abgebildete neuronale Netz hat nur drei verdeckte Schichten, da eine hohe Anzahl an verdeckten Schichten die Reaktionszeit des Agenten verringert und dieser, nach Unterabschnitt 4.2.2 nur 10ms für die Auswahl von Aktionen hat. Nachteil des DQN-Algorithmus ist seine ausschließliche Anwendbarkeit auf diskrete Aktionsräume. Die nachfolgende Tabelle 4.3 zeigt auf, wie die reellen Wertebereiche der Aktoren diskretisiert wurden, um eine endliche Anzahl von Aktionswerten zu erhalten. Dies ist notwendig, da, nach Gleichung 3.39, ein DQN als Ausgabe einen endlichen Vektor mit den Erträgen der Aktionen hat. Auf einem reellen Intervall würde somit die Auswahl der Aktion im Ausgabevektor, die den größten Ertrag aufweist, nicht möglich sein. Für den DDPG-Agenten gibt es diesen Mehraufwand nicht und die Wertebereiche aus Tabelle 4.2 gelten.

Bei der Auswahl der Klassengrößen für die Diskretisierung wurde folgendermaßen vorgegangen: Das Lenkintervall $[-1; 1]$ wurde in fünf Klassen aufgeteilt. Jede Klasse entspricht im neuronalen Netz dann einem linear-aktivierten Ausgabeneuron. Das Ausgabeneuron mit dem höchsten Ertragswert wird ausgewählt und repräsentiert die Aktion des Agenten, falls nicht erkundet wird. Diese Aktionen reichen nach Tabelle 4.2 von $-0,5$ bis $0,5$. Es sollen zu starke und erratische Lenkbewegungen des Agenten vermieden werden, sodass sich auf diesen Lenkbereich beschränkt wurde. Eine ähnliche Einteilung nehmen Wolf et al. in [WHW⁺17] vor. Eine feinere Einteilung nehmen Jaritz et al. in [JdT⁺18] vor. Für die Beschleunigung wurde der Wertebereich in acht Klassen diskretisiert, da der Agent das Fahrzeug möglichst gefühlvoll, insbesondere in Kurven, längsführen soll. Bei der Bremsung wurde auf den größtmöglichen Wert im Wertebereich verzichtet, da dies einer Vollbremsung entsprechen würde, die durch gefühlvolles und vorausschauendes Fahren möglichst vermieden werden sollte.

Aktor	Werte	Diskretisierung	Anz. Neuronen
Steering	$[-1; 1]$	$[-1; -0,5)$ Linksdrehung ist $-0,5$ $[-0,5; -0,3)$ Linksdrehung ist $-0,3$ $[-0,3; 0,3)$ Lenkraddrehung ist $0,0$ $[0,3; 0,5)$ Rechtsdrehung ist $0,3$ $[0,5; 1]$ Rechtsdrehung ist $0,5$	5
Acceleration	$[0; 1]$	$\{0\}$ Beschleunigung ist $0,0$ $(0; 0,4)$ Beschleunigung ist $0,2$ $[0,4; 0,5)$ Beschleunigung ist $0,45$ $[0,5; 0,6)$ Beschleunigung ist $0,55$ $[0,6; 0,7)$ Beschleunigung ist $0,65$ $[0,7; 0,8)$ Beschleunigung ist $0,75$ $[0,8; 0,9)$ Beschleunigung ist $0,85$ $[0,9; 1]$ Beschleunigung ist $0,95$	8
Brake	$[0; 1]$	$[0; 0,3)$ Bremsung ist $0,0$ $[0,3; 0,5)$ Bremsung ist $0,3$ $[0,5; 1]$ Bremsung ist $0,5$	3

Tabelle 4.3: Tabellarische Übersicht über die, für den DQN-Agenten genutzte, Diskretisierung der Wertebereiche der Aktionen. Die Spalte ganz rechts gibt an, wie viele Ausgabeneuronen jede Aktion hat. Die Werte der Ausgabeneuronen einer Aktion können als Vektor aufgefasst werden. Dann muss der Agent, nach Unterabschnitt 3.3.4, nur noch die Aktion auswählen, die den größten, langfristig zu erwartenden, Ertrag für einen Zustand erbringt.

Aus Unterabschnitt 3.4.5 werden die Charakteristika des DDPG-Algorithmus, wie das neuronale *Actor*- und *Critic*-Netz oder das *Experience Replay Memory*, deutlich. Diese wurden implementiert. Der Aufbau der neuronalen Netze des *Actor* und des *Critic* sind aus Abbildung C.2 und Abbildung C.3 in Anhang C ersichtlich. Ebenso wie der Quellcode des DQN-Algorithmus, ist auch der Quellcode des DDPG-Algorithmus auf dem dieser Arbeit beiliegenden Datenträger und auf https://github.com/koeller21/ma_code einsehbar. Bei der Implementierung hat sich die Findung einer geeigneten Verfallrate von ε sowie geeigneter OU-Parameter als besonders aufwendig herausgestellt. Die im Experiment verwendeten OU-Parameter sind, zusammen mit den anderen Hyperparametereinstellungen des DDPG, in Tabelle C.2 dargestellt. Dabei wurde sich grob an den Hyperparametereinstellungen von Wang et al. aus [WJW18] und Lau aus [Lau16] orientiert.

Die im Experiment verwendete Hard- und Software ist in der nachfolgenden Auflistung genannt. Aufgrund der relativ wenig verdeckten Schichten in den Architekturen der neuronalen Netze der Agenten-Algorithmen ist das Training der Agenten, anders als etwa im bildverarbeitenden *Supervised Learning*, auch mit wenig leistungsstarker Hardware möglich.

1. Hardware

- (a) Intel Core i5-4570 @ 3,20GHz
- (b) GeForce GTX 960 mit 2048MB Videospeicher
- (c) 8,1GB DDR3 RAM @ $1333\frac{\text{MT}}{\text{s}}$
- (d) ASUSTeK H87-Plus Motherboard

2. Software

- (a) elementary OS 5.0 Juno
- (b) 4.15.0-43-generic Linux Kernel
- (c) Python Version 3.6.7
- (d) Keras 2.2.4
- (e) Tensorflow 1.12.0
- (f) numpy 1.16.2

Mit der oben genannten Hard- und Software dauert das Training und Testen bei 350 Episoden etwa zwölf Stunden pro Strecke. In TORCS kann die Simulationsgeschwindigkeit auf das Vierfache der normalen Simulationsgeschwindigkeit erhöht werden, wovon auch Gebrauch gemacht wurde. Damit reduzierte sich die Zeit zum Trainieren und Testen eines Agenten auf etwa drei bis vier Stunden. Besonders viel Zeit in Anspruch nimmt dabei die Konfiguration der Hyperparameter. Ist etwa die Lernrate des DDPG-Actors oder des DQN-Agenten zu klein, wird der Agent in lokalen Minima gefangen und entsprechend wenig Ertrag erwirtschaften. Während der Experimentdurchführung hat sich gezeigt, dass ein sorgfältiges Einstellen der Hyperparameter, insbesondere beim DDPG-Algorithmus, wichtig für eine schnelle Konvergenz ist.

Unter der Webadresse <https://www.youtube.com/watch?v=W7YKE8cbzg0> ist ein Video zu finden, das den Lernfortschritt der Agenten erkenntlich macht. In Anhang J wird überdies erklärt, wie das auf dem dieser Arbeit beiliegenden Datenträger befindliche Programm `pyrl.py`, mit dem der DQN- und der DDPG-Algorithmus aufgerufen werden kann, zu bedienen ist.

5 Ergebnisse und Interpretation

In diesem Kapitel werden die Ergebnisse des Experimentes dargestellt und interpretiert, um die in Kapitel 1 formulierten Ziele dieser Arbeit zu erreichen. In Abschnitt 5.1 werden die Experimentergebnisse des DDPG-Agenten gezeigt. In Abschnitt 5.2 folgen die Ergebnisse zum Vergleich verschiedener Belohnungsfunktionen. In Abschnitt 5.3 werden die Ergebnisse des DQN-Agenten dargestellt und denen des DDPG-Agenten gegenübergestellt. In Abschnitt 5.4 werden die gefolgerten Erkenntnisse zusammengefasst und mit anderen Arbeiten verglichen.

5.1 Ergebnisauswertung DDPG-Agent

5.1.1 Training und Test des DDPG-Agenten im *Practice*-Modus

Abbildung 5.1 zeigt die aufgezeichneten Performanzdaten des DDPG-Agenten unter der Belohnungsfunktion \mathcal{R}_1 auf der Strecke CG Speedway während des Trainings.

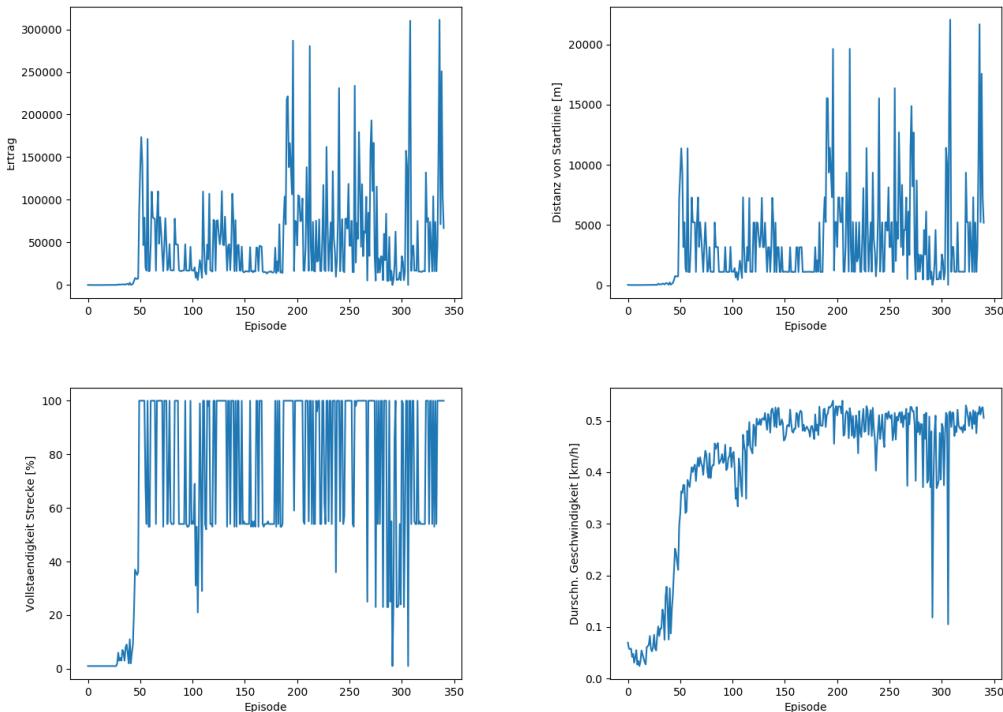


Abbildung 5.1: Dargestellt sind die Performanzdaten des DDPG-Agenten während des Trainings auf der Strecke CG Speedway 1. Von l.o. nach r.u.: (a) Ertrag pro Episode, (b) erreichte Distanz des Agenten von der Startlinie pro Episode, (c) Vollständigkeit der Rennstrecke pro Episode und (d) durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.

Wie erwartet, verbessert sich die Performanz des DDPG-Agenten mit zunehmender Anzahl an Episoden. Dies wird aus allen Performanzgraphen in Abbildung 5.1 deutlich. Sowohl der Ertrag, die Distanz, als auch die Geschwindigkeit weisen den Lernfortschritt des Agenten aus. Insbesondere fällt auf, dass kurz nach der 50. Episode der Agent erstmals einen hohen Ertrag einfährt und eine hohe Distanz zurücklegt. Aus dem Performanzgraphen (c) zum Referenzrahmen nach Unterabschnitt 4.2.6 wird deutlich, dass der Agent kurz nach der 50. Episode die erste volle Streckenumrundung schafft. Der Lernforschritt wird aus den Performanzdaten des Graphen 5.1 (d), zur durchschnittlichen Geschwindigkeit des Agenten, besonders schön deutlich. Zunächst ist die durchschn. Geschwindigkeit relativ gering, steigt jedoch ab der 50. Episode rapide an, was mit dem Zurücklegen größerer Distanzen zu erklären ist. Ungefähr ab der 150. Episode stabilisiert sich die durchschnittliche Geschwindigkeit auf einem Niveau von 0,5. Neben dem Lernfortschritt des Agenten ist vor allem die Betrachtung von Performanzdaten in Episoden interessant, in denen die gemessenen Werte aus dem vermuteten Trend fallen oder in einem anderen Maße erwartet wurden. So sind beispielsweise die Performanzwerte um die 300. Episode interessant, da sowohl Graph (c) als auch Graph (d) in Abbildung 5.1 einen Einbruch in der prozentual erreichten Streckenweite und in der durchschnittlichen Geschwindigkeit aufzeigen. Auch scheint Graph (c) in Abbildung 5.1 ab der 50. Episode ständig zwischen 60 und 100 Prozent zu fluktuieren. Überdies scheint es so, als ob der gemessene Ertrag aus 5.1 (a) direkt aus der Distanz aus 5.1 (b) hervorgeht, da beide Performanzgraphen fast identisch aussehen. Vor einer genaueren Interpretation und Erklärung dieser Beobachtungen seien aber noch die Performanzdaten des DDPG-Agenten auf den Strecken E-Road und Forza dargestellt, um diese Beobachtungen mit mehr Daten zu bestätigen.

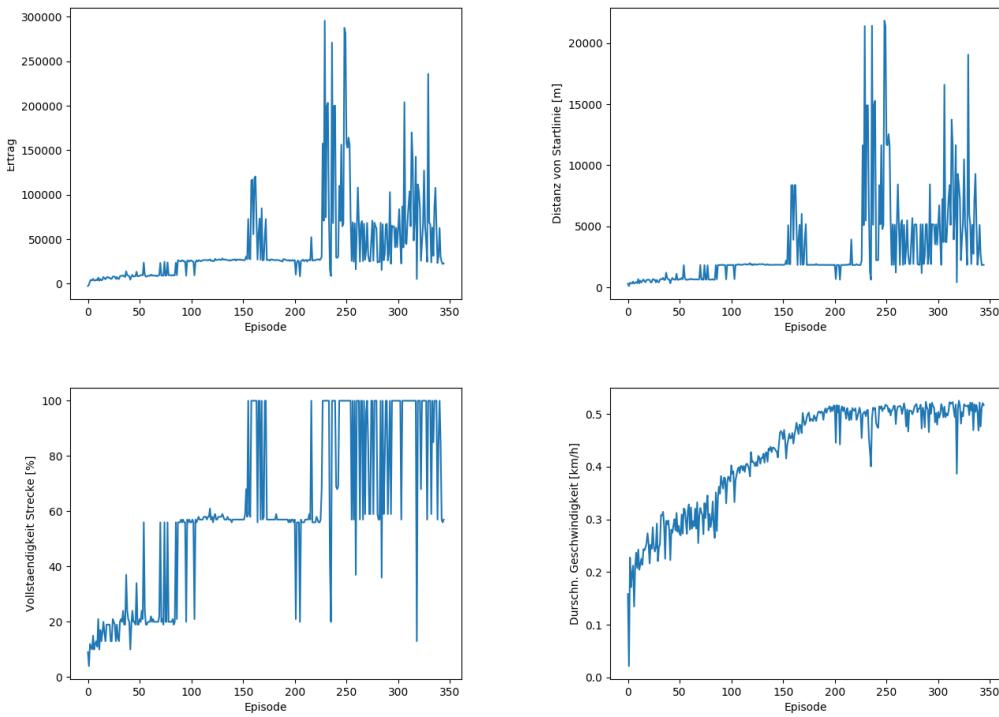


Abbildung 5.2: Dargestellt sind die Performanzdaten des DDPG-Agenten während des Trainings auf der Strecke E-Road. Von l.o. nach r.u.: (a) Ertrag pro Episode, (b) erreichte Distanz des Agenten von der Startlinie pro Episode, (c) Vollständigkeit der Rennstrecke pro Episode und (d) durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.

5 Ergebnisse und Interpretation

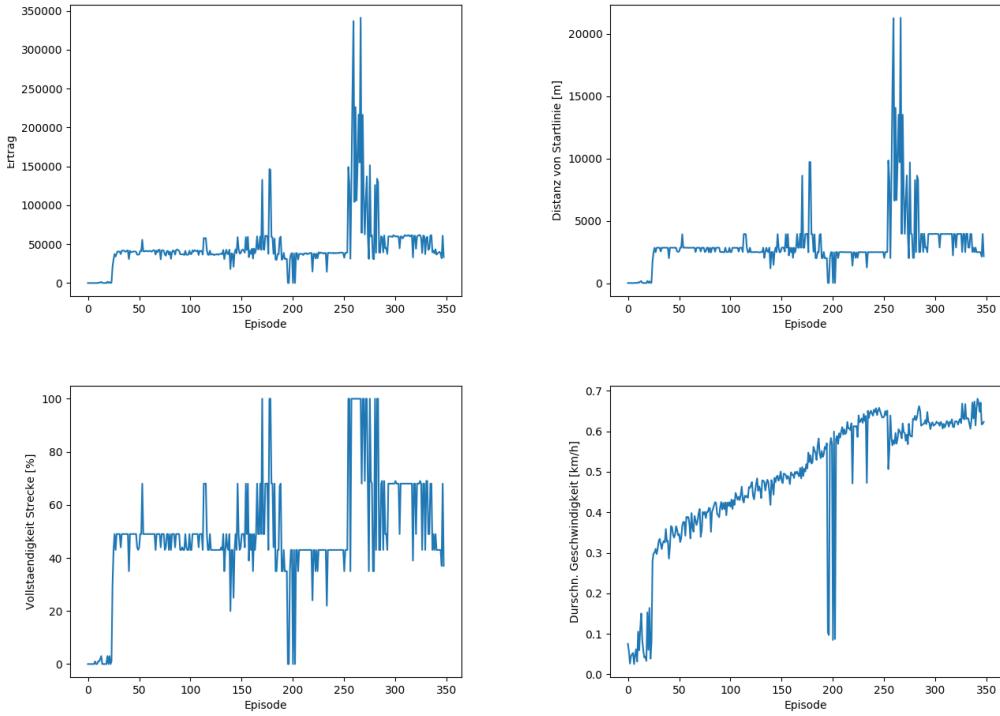


Abbildung 5.3: Dargestellt sind die Performanzmetriken des DDPG-Agenten während des Trainings auf der Strecke Forza. Von l.o. nach r.u.: (a) Ertrag pro Episode, (b) erreichte Distanz des Agenten von der Startlinie pro Episode, (c) Vollständigkeit der Rennstrecke pro Episode und (d) durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.

Mit den Performanzgraphen aus den Abbildungen 5.2 und 5.3 können die obig gemachten Beobachtungen bestätigt werden. Zudem fällt auf, dass der DDPG-Agent auf den Strecken E-Road, Abbildung 5.2, und Forza, Abbildung 5.3, deutlich weniger häufig die Rennstrecken vollständig umrunden kann, als dies noch auf der Strecke CG Speedway der Fall gewesen ist. Wird diese Beobachtung im Kontext der Streckendarstellung von Abbildung 4.2 gesehen, ist sie trivial erklärbar: Da die Strecken E-Road und Forza deutlich länger und kurviger als die Strecke CG Speedway sind, gibt es mehr Möglichkeiten für den Agenten zu scheitern. Im Vergleich zum Performanzgraphen zur durchschnittlichen Geschwindigkeit des DDPG-Agenten auf der Strecke CG Speedway, fällt in 5.2 (d) überdies auf, dass der Agent auf der Strecke E-Road mit ca. 200 Episoden deutlich länger braucht, um ein gleichbleibendes, durchschnittliches Geschwindigkeitsniveau, das ebenfalls bei 0,5 liegt, zu finden. Auf der Strecke Forza braucht der Agent mit ca. 225 Episoden am längsten für diesen Konvergenzvorgang, hat dann mit etwa 0,65 aber auch die höchste durchschnittliche Geschwindigkeit. Dies ist aus Abbildung 5.3 (d) ersichtlich. Weniger aus den Graphen dieses Abschnitts, als mehr aus der visuellen Darstellung bestimmter Fahrsituationen in TORCS, lassen sich weitere Schlüsse über den Lernfortschritt und das Verhalten des DDPG-Agenten ziehen. So ist in Abbildung D.1 der Bremsvorgang eines DDPG-Agenten in einer Kurve auf der Strecke E-Road dargestellt. Dabei wird deutlich, dass der Agent erlernt hat, vor einer Kurve abzubremsen, um nicht durch eine zu hohe Geschwindigkeit die Fahrbaummärkierung zu überfahren. Dies zeigt vor allem, dass der Agent antizipatorische Fähigkeiten erlangt hat und beweist eindrucksvoll die Mächtigkeit der Gleichungen von Bellman für das *Reinforcement Learning*.

Aus den obigen Abbildungen sollen, wie bereits angesprochen, zwei Beobachtungen genauer untersucht, interpretiert und die Folgen dieser abgeschätzt werden:

1. Häufig folgen auf eine oder mehrere Episoden, bei denen der Agent die Strecke vollends umrundet hat, Episoden in denen der Agent die Strecke nicht völlig umrundet. Dieses Verhalten des DDPG-Agenten ist auf allen drei Strecken durch die Performanzgraphen 5.1, 5.2 und 5.3 (c) zu beobachten.
2. Der Ertrag scheint sich alleinig aus der zurückgelegten Distanz des Agenten zu ergeben. Dies ist ebenfalls auf allen Strecken durch einen Abgleich der Performanzdaten in den Graphen (a) und (b) in den Abbildungen 5.1, 5.2 und 5.3 zu beobachten.

Die erste Beobachtung wirft insbesondere die Frage nach der Verlässlichkeit und Stabilität des Agenten auf. So ist aus Abbildung 5.1 (c) ersichtlich, dass der Agent bereits nach ca. 50 Episoden die Strecke vollends umrunden kann, dies jedoch, um Episode 300 herum, mehrheitlich nicht mehr schafft. Ähnliche Leistungseinbrüche lassen sich auf den anderen Strecken beobachten. In der Folge dieser Beobachtung stellt sich die Frage nach der Verlässlichkeit des Agenten. Diese Frage kann leicht beantwortet werden; indem zunächst festgehalten wird, dass der Agent während des Trainings fortwährend die Umgebung erkundet. Der Agent kann nur die optimale *Policy* mit Sicherheit gefunden haben, wenn er während des Trainings alle möglichen Aktionen in allen möglichen Zuständen ausprobiert hat. Für einen fast unendlichen Zustands- und Aktionenraum wäre dies sehr zeitaufwendig. Aus diesem Grund werden neuronale Netze eingesetzt, die die optimale *Policy* immer nur approximativ bestimmen. Für den Test auf neuen, unbekannten Strecken wird dann erwartet, dass der trainierte Agent möglichst gute und stabile Ergebnisse, bezüglich der Performanzmetriken, erzielt. Wie in Unterabschnitt 4.2.6 angekündigt, wird der trainierte Agent im Laufe dieses Unterabschnittes auch auf neuen, unbekannten Strecken getestet.

Die zweite Beobachtung wirft zwei Fragen auf. Der Ertrag scheint sich direkt aus der Distanz zu ergeben. Wird \mathcal{R}_1 aus Gleichung 4.1 betrachtet, scheint diese Beobachtung plausibel, da \mathcal{R}_1 ein schnelles, fahrbahnmittiges Verhalten des Agenten belohnt. Eine hohe Distanz von der Startlinie ist Ausdruck und Folge eines solchen Verhaltens und führt damit zu viel Ertrag. Inwiefern etwa die Ereignisse und Terminierungsbedingungen aus Unterabschnitt 4.2.4 auf den Ertrag wirken, geht aus den bisher vorgestellten Performanzmetriken und -graphen nicht hervor. Somit stellt sich die Frage nach einer guten Darstellungsform für die Beschreibung der Auswirkung der Einflussgrößen der Belohnungsfunktion auf den Ertrag. Eine unspezifische Performanzmetrik, wie die Distanz von der Startlinie, zeigt nur den aggregierten Effekt dieser Einflussgrößen auf. Genau wie die Geschwindigkeit, sollte also auch der Längsweg, nach Gleichung 4.3, als Performanzmetrik genutzt werden, da der Längsweg ein Term von \mathcal{R}_1 ist. So ließe sich ein Zusammenhang zwischen Längsweg und Ertrag leichter erkennen, was für die Modellierung der Belohnungsfunktionen nützlich wäre sowie den Ertrag mitbegründen würde und damit leichter interpretierbar machen würde. Auch in den Publikationen in Anhang B werden zumeist nur aggregierte Metriken genutzt, was für eine transparente Modellierung einer Belohnungsfunktion hinderlich ist. Aus dieser Erfahrung heraus wird für Unterabschnitt 5.1.2 eine weitere Performanzmetrik eingeführt, die die Anzahl der Kollisionen mit anderen Fahrzeugen beschreibt. Die zweite sich stellende Frage aus der zweiten Beobachtung schließt an die erste Frage an und fragt nach den notwendigen Einflussgrößen der Belohnungsfunktion, um eine bestimmte Fahraufgabe zu erfüllen. In der vorliegenden Arbeit ist diese Fahraufgabe, nach Abschnitt 4.1, die Quer- und Längsführung des Fahrzeugs. Wird diese Fahraufgabe noch um das Einparken, das Bestehen im innerstädtischen Verkehr und die Verkehrsregelkonformität erweitert, würde eine Belohnungsfunktion sehr kompliziert werden. Aufgrund der vielen Einflussgrößen wird eine termweise Zusammenhangsanalyse mit dem Ertrag zu aufwändig und undurchsichtig. Diese ist dann nur noch mit aggregierten Metriken möglich. Um diesem Dilemma zu entrinnen, wird Folgendes vorgeschlagen: Zwecks der Modellierung einer der Fahraufgaben entsprechenden Belohnungsfunktion, wird ein *Inverse Reinforcement Learning*-Algorithmus, der aus einer gegebenen *Policy* und den Umgebungszuständen die Belohnungsfunktion ableitet, durch einen menschlichen Fahrer, für den optimales Fahrverhalten angenommen wird, trainiert. Ein *Reinforcement*

5 Ergebnisse und Interpretation

Learning-Agent kann dann diese Belohnungsfunktion nutzen, um eine optimale *Policy* zu lernen. Ähnliches Vorgehen schlagen Vasquez et al. in [VYKL14] vor. Im Ausblick in Kapitel 6 wird genauer auf diesen Vorschlag eingegangen. Dieser Lösungsvorschlag erleichtert die Interpretation des Ertrages eines Agenten deutlich, da dieser, aufgrund einer händisch modellierten Belohnungsfunktion, nun nicht weiter hinterfragt und ergründet werden muss. Stattdessen ergibt sich der Ertrag direkt aus der, auf menschlichem Verhalten basierenden, Belohnungsfunktion, die einem *Inverse Reinforcement Learning*-Algorithmus entsprungen ist. Die Implementation eines *Inverse Reinforcement Learning*-Algorithmus fällt aus dem Rahmen der Ziele von Kapitel 1 und daher wird in dieser Arbeit versucht, das Agentenverhalten über die bekannten Performanzmetriken zu ergründen und zu interpretieren.

Wie in der Interpretation zur ersten Beobachtung angekündigt, soll der trainierte DDPG-Agent nun auf neuen, unbekannten Rennstrecken getestet werden. Die Wichtigkeit eines solchen Testes ist in Unterabschnitt 4.2.6 hervorgehoben. Der Test dient dazu, die Fähigkeit des trainierten Agenten zur Generalisierung zu überprüfen. Hierfür wird der Agent in neuen, bisher unbekannten Zuständen, Aktionen ausführen müssen. Sind diese Aktionen weiterhin ertragreich, dann ist davon auszugehen, dass der Agent generalisieren kann. Im Kontext des autonomen Fahrens kann ein illustratives Beispiel der Fähigkeit zur Generalisierung das plötzliche Fehlen der Fahrbahnmarkierung, aufgrund von Straßenbauarbeiten, sein. Hat der Agent im Training das Halten der Spur lediglich mit der weißen Fahrbahnmarkierung zu assoziieren gelernt, dann würde er auf einer Straße ohne Fahrbahnmarkierung versagen. Hat der Agent im Training einen groben Zusammenhang zwischen Fahrbahnmarkierung, Abstand zum Vordermann und typischer Farbe des Fahrbahnbelages gelernt, dann ist es wahrscheinlicher, dass der Agent auch auf einer Straße ohne Fahrbahnmarkierungen besteht.

Der DDPG-Agent wurde auf drei unterschiedlichen Strecken stets von Neuem trainiert. Nach jedem Training von 350 Episoden auf einer Strecke, wird der Agent auf den jeweils anderen beiden Strecken für 350 Episoden getestet. Die Ergebnisse dieses Testens sind in Anhang E einsehbar. Aus den aufgezeichneten Performanzdaten in Anhang E lassen sich vier Auffälligkeiten beobachten:

1. Beim Testen ist ein unmittelbarer Anstieg der Performanzwerte des Agenten ab der ersten Episode zu beobachten. Dies geht aus allen Performanzgraphen aller Testfälle E.1, E.2 und E.3 in Anhang E hervor.
2. Der Agent scheint allerdings keine umfassenden Fähigkeiten zur Generalisierung erlangt zu haben. Dies ist aus der Betrachtung der prozentual erreichten Streckenweite pro Episode durch den Agenten in Graph (c) in E.1 gut zu erkennen, da der Agent hier zumeist nicht hundert Prozent erreicht.
3. Anders als beim Training des Agenten, ist keine kontinuierliche Verbesserung der Performanzwerte, durch einen Lernprozess, erkennbar. Dies geht aus allen Performanzgraphen aller Testfälle in Anhang E hervor.
4. Aus Abbildung E.2 in Anhang E wird deutlich, dass der auf der Strecke E-Road trainierte DDPG-Agent beim Testen eine hohe Varianz bezüglich der Performanzdaten auf der Strecke CG Speedway aufweist.

Die erste Beobachtung lässt zunächst vermuten, dass der DDPG-Agent das Erlernte auf neuen Strecken anwenden kann. So ist im Vergleich zu den Performanzwerten im Training der Ertrag, die Distanz, die prozentual erreichte Streckenweite und die durchschnittliche Geschwindigkeit des Agenten ab der ersten Episode sprunghaft hoch. Dies geht sehr schön aus den Graphen in z.B. Abbildung E.3 hervor. Der Agent scheint also ein grundsätzliches Verständnis für ein ertragreiches Fahrverhalten erlangt zu haben und kann dies auf den Teststrecken umsetzen. Die zweitgenannte Beobachtung schränkt dieses wünschenswerte Ergebnis aber wieder ein. Zwar sind die Performanzdaten im Test ab der ersten Episode sprunghaft hoch, doch geht aus beinahe allen Graphen, zur prozentual erreichten Streckenweite auf den Teststrecken, hervor, dass der trainierte Agent keine volle Streckenumrundung

schafft. Der Agent scheint dabei zumeist an einer bestimmten Stelle der Rennstrecke zu scheitern, was sich aus dem gleichbleibenden Niveau der Performanzwerte in den Graphen zur prozentual erreichten Streckenweite schließen lässt. Dies ist ein Hinweis darauf, dass der Agent mit einer bestimmten Situation nicht richtig umzugehen weiß, da er dieser auf der Trainingstrecke nicht begegnet ist oder nicht generalisieren kann. Dann scheitert der Agent in jeder Episode des Tests an dieser Situation. Die dritte Beobachtung ist der argumentative Ausgangspunkt für die Begründung dieses konstanten Scheiterns des Agenten. Da sich der Agent im Test befindet, wird die Umgebung nicht weiter erkundet. Der Agent wird also nicht zufälligerweise ertragreiche Aktionen in den, bisher unbekannten, Situationen finden können und somit an diesen stets scheitern. Die, aus der ersten Beobachtung, vermutete Fähigkeit zur Generalisierung scheint dies nicht verhindern zu können. Insbesondere wird der Agent in Abbildung E.3 an der komplizierten Strecke Forza trainiert und, unter anderem, auf der einfacheren Strecke CG Speedway getestet. Selbst in diesem Szenario schafft der Agent auf der Strecke CG Speedway keine volle Streckenumrundung. Gute Trainingsergebnisse, aber schlechte Testergebnisse legen den Verdacht eines *Overfitting* nah. Um die Fähigkeit eines DDPG-Agenten zur Generalisierung im Kontext des autonomen Fahrens in Simulationsumgebungen noch präziser zu untersuchen, müsste an dieser Stelle ein Folgeexperiment ein mögliches *Overfitting* des Agenten während des Trainings durch Metriken belegen. Hierfür empfiehlt sich das, aus dem *Supervised Learning* bekannte, *Cross Validation*-Verfahren für die frühzeitige Erkennung von *Over-* oder *Underfitting* der neuronalen Netze des DDPG-Agenten. Die Nutzung des *Cross Validation*-Verfahrens für *Reinforcement Learning* wird als perspektivische Fragestellung weiterer Forschung im Ausblick in Kapitel 6 genannt.

Die vierte Beobachtung geht auf die, bereits vorgebrachte, Besorgnis über die Stabilität des Fahrverhaltens des Agenten ein. So ist beispielsweise in Abbildung 5.3 aus den Graphen (c) und (d) ersichtlich, dass um die 200. Episode herum eine Anomalie auftritt und der Agent deutlich weniger der Strecke schafft als vormals. Auch die Geschwindigkeit ist stark verringert. Derlei Anomalien wurden mit dem fortwährenden Erkunden der Umgebung während des Trainings begründet und es wird für die Performanzdaten des DDPG-Agenten im Test erwartet, dass diese stabiler sind. Betrachtet man die Performanzdaten auf den Teststrecken in den Abbildungen E.1, E.2 und E.3, dann bestätigt sich diese Erwartung. Zwar gibt es noch kleinere Schwankungen, ein dauerhafter Abfall oder starke Schwankungen der Performanzdaten sind jedoch nicht beobachtbar. Eine Ausnahme bilden allerdings die Performanzwerte des Agenten auf der Teststrecke CG Speedway in Abbildung E.2. Diese fluktuieren sehr stark und häufig. Grund dafür könnte ein *Overfitting* auf der komplizierteren Strecke Forza sein. Der Agent nimmt dann z.B. die Kurven auf der einfacheren Strecke CG Speedway als deutlich enger an, als sie tatsächlich sind und manövriert den Fahrzeugbot abseits der Fahrbahn. Die empirische Feststellung der Stabilität im Fahrverhalten eines *Reinforcement Learning*-Agenten ist ein überaus wichtiges Argument für den perspektivischen Einsatz dieser Algorithmen in echten Fahrzeugen.

5.1.2 Training des DDPG-Agenten im *Quick Race*-Modus

Nachdem in Unterabschnitt 5.1.1 die Performanzdaten des DDPG-Agenten im *Practice*-Modus dargestellt und interpretiert wurden, folgt in diesem Unterabschnitt die Untersuchung und Analyse der Performanzdaten des DDPG-Agenten im *Quick Race*-Modus. Als Belohnungsfunktion wird \mathcal{R}_1 genutzt. Dabei wird der DDPG-Agent ausschließlich auf der Strecke CG Speedway trainiert, da diese, aufgrund ihrer Kürze, die meisten Möglichkeiten zur Interaktion mit anderen Fahrzeugen bietet. Wie in Unterabschnitt 4.2.7 genannt, ist primäre Motivation zur Durchführung eines Experimentes im *Quick Race*-Modus die Untersuchung des Interaktionsverhaltens des Agenten mit anderen Fahrzeugen. Da, nach Unterabschnitt 4.2.6, die Kollision mit anderen Fahrzeugen penalisiert wird, wird ein abschwellender Verlauf der Anzahl an Kollisionen pro Episode während des Trainings erwartet. Diese Metrik wird, nach Unterabschnitt 5.1.1, hier zusätzlich genutzt. Es ist zu erwarten, dass sich die Beobachtungen eines Testdurchlaufes des DDPG-Agenten im *Quick Race*-Modus denen

5 Ergebnisse und Interpretation

aus dem Testdurchlauf im *Practice*-Modus konzeptionell sehr ähneln würden. Aus diesem Grund steht das Training des DDPG-Agenten in diesem Unterabschnitt im Vordergrund. An dem Rennen teilnehmen werden sieben Fahrzeugbots, die sehr heterogen bezüglich ihrer Leistungsdaten sind. Das in Abbildung 4.6 dargestellte Fahrzeug des Agenten ist leistungstechnisch im oberen Mittelfeld zu verorten. Der Fahrzeugbot des Agenten startet von der letzten Position. Erwartet wird deshalb, dass der Agent im Laufe des Trainings einige der langsameren Rennteilnehmer überholen kann und so eine stetig verbesserte, durchschnittliche Position erzielen kann. Auch die Position der Rennteilnehmer wird deshalb als Performanzmetrik genutzt und Performanzdaten zu dieser Metrik werden aufgezeichnet. Die Hyperparametereinstellungen und der Aufbau des DDPG-Agenten sind identisch mit denen des Agenten aus Unterabschnitt 5.1.1.

In Abbildung G.1 in Anhang G sind die aufgezeichneten Performanzdaten des DDPG-Agenten während des Trainings im *Quick Race*-Modus auf der Strecke CG Speedway dargestellt. Aus den in Abbildung G.1 dargestellten Performanzdaten über die Episoden, können folgende interessante Beobachtungen gemacht werden:

1. Der Graph G.1 (e) in Anhang G, zur Anzahl der Kollisionen pro Episode, zeigt das erwartete und erhoffte Verhalten auf. Zunächst sind die Kollisionen gering, steigen dann an und schließlich scheint der Agent eine Kollisionsvermeidung erlernt zu haben.
2. Im Laufe des Trainings verbessert der DDPG-Agent seine durchschnittliche Position im Rennen stetig. Dies ist aus G.1 (f) ersichtlich.
3. Die Graphen G.1 (c) und (d) zeigen größere und häufigere Schwankungen in den aufgezeichneten Performanzwerten auf, als dies noch im *Practice*-Modus in z.B. 5.1 (c) und (d) der Fall war.

Der Graph G.1 (e) in Anhang G liegt der ersten Beobachtung zugrunde und beschreibt die Interaktion des vom Agenten gesteuerten Fahrzeugbot mit den anderen Fahrzeugen im Rennen. Die Ausgestaltung dieser Interaktion ist, in der vorliegenden Arbeit, denkbar einfach. Der Agent muss den anderen Fahrzeugen einfach ausweichen, um so möglichst wenig mit diesen zu kollidieren. Aus G.1 (e) lässt sich erkennen, dass der vom Agenten gesteuerte Fahrzeugbot in den ersten 50 Episoden nur sehr wenig mit den anderen Fahrzeugen kollidiert. Dies lässt sich leicht erklären: Da der Agent von der hintersten Position startet und in den ersten Episoden noch keine Fähigkeiten zur Quer- und Längsführung des Fahrzeugs erlangt hat, fahren die anderen Fahrzeuge dem Agenten davon. Nach etwa 50 Episoden hat der Agent die langsamsten Konkurrenten eingeholt und kollidiert mit diesen. Kollisionen werden nach Unterabschnitt 4.2.6 penalisiert und so lernt der Agent im Laufe der Trainingsepisoden anderen Fahrzeugen auszuweichen. Besonders gut lässt sich dies an der Anzahl der Kollisionen ab der 240. Episode erkennen. Hier wird deutlich, dass der Agent weniger kollidiert als zuvor und gleichzeitig, betrachtet man G.1 (b) und (c), hohe Distanzen zurücklegt. Die gemeinsame Betrachtung dieser beiden Graphen ist hier notwendig, da eine geringe zurückgelegte Distanz von der Startlinie zumeist mit wenig Kollisionen einhergeht. Will man also einen bedeutungsvollen Trend in Graph G.1 (e) erkennen, muss man gleichzeitig G.1 (b) betrachten. Alternativ könnte man eine Performanzmetrik definieren, die die Kollisionen bezogen auf die zurückgelegte Distanz misst. Diese Performanzmetrik wird für die Auswertung der Belohnungsfunktionen im nächsten Unterabschnitt auch genutzt. In Abbildung F.1 in Anhang F sind gelungene Überholmanöver des Agenten während des Trainings dargestellt. Positiv erstaunlich ist die Fähigkeit des DDPG-Agenten eine hochkomplexe Aufgabe, wie das kollisionsfreie Überholen anderer Verkehrsteilnehmer, selbstständig zu erlernen; insbesondere vor dem Hintergrund der simplen Belohnungsfunktion, der Ereignisse und Terminierungsbedingungen aus Unterabschnitt 4.2.4. Perspektivisch können so weitere Untersuchungen zu anderen Aufgaben des autonomen Fahrens, wie dem regelkonformen Fahren oder dem Ein- und Ausparken, motiviert werden, da der DDPG-Agent gute Ergebnisse beim Erlernen von Überholmanövern aufzeigt.

Die zweite Beobachtung geht aus den Performanzdaten in Graph G.1 (*f*) in Anhang G hervor. Es ist deutlich zu erkennen, dass der Agent mit zunehmender Anzahl an Trainingsepisoden stets bessere durchschnittliche Positionen bzw. Platzierungen im Rennen erreicht. Grund dafür ist die sich stets erhöhende durchschnittliche Geschwindigkeit, was aus G.1 (*d*) hervorgeht sowie die immer weniger werdenden Kollisionen, was aus G.1 (*e*) hervorgeht. Sehr schön ist dieser Zusammenhang ab Episode 280 zu erkennen: Die Anzahl der Kollisionen nimmt dort wieder zu und die durchschnittliche Platzierung des Agenten verschlechtert sich. Auch wird in Unterabschnitt 4.2.6 dargestellt, dass der Agent für erfolgreiche Überholmanöver belohnt wird, was ebenfalls Einfluss auf die sich stetig verbessernde durchschnittliche Platzierung haben kann.

Der dritten Beobachtung liegt ein Vergleich der Performanzdaten im *Quick Race*-Modus mit denen im *Practice*-Modus zugrunde. Betrachtet man die Graphen in G.1 (*c*) und (*d*) in Anhang G im Vergleich zu den korrespondierenden Graphen in Abbildung 5.1, dann wird erkenntlich, dass die Performanzwerte im *Quick Race*-Modus stärker und häufiger schwanken als im *Practice*-Modus. Die Begründung dafür ist offensichtlich, bringt aber weitreichende Folgen mit sich: Die Interaktion mit anderen Fahrzeugen scheint den vom Agenten gesteuerten Fahrzeugbot zu verlangsamen oder drängt diesen gar abseits der Fahrbahn. Würde man zusätzlich also noch weitere Verkehrsteilnehmer simulieren, wie z.B. Fußgänger, Fahrradfahrer oder Passanten, ist zu vermuten, dass die Performanzdaten mindestens genau so stark schwanken werden. Insbesondere scheint der Agent auch nach 350 Episoden noch nicht allen Kollisionen aus dem Weg gehen zu können, was aus Abbildung G.1 (*e*) hervorgeht. In Unterabschnitt 5.1.1 wurde die Instabilität noch durch die fortwährende Erkundung der Umgebung begründet, doch aus den dieser Beobachtung zugrundeliegenden Performanzdaten und ihrer Interpretation wird deutlich, dass Schwankungen in den Performanzwerten auch durch eine komplexere Umgebung induziert werden können. Eine reine Begründung der Schwankungen mittels des Erkundungsargumentes aus Unterabschnitt 5.1.1 ist hier nicht vorbringbar, da im *Practice*- und *Quick Race*-Modus die gleichen OU-Parameter genutzt werden. Hat ein Agent nun die Quer- und Längsführung erfolgreich erlernt, mag man diesen Agenten um die nächste Fähigkeit erweitern, wie die der Kollisionsvermeidung. Danach etwa um die Verkehrsregelkonformität. Wird die Komplexität der Umgebung also stets erhöht, um irgendwann dem realen Straßenverkehr zu entsprechen, dann kann man aus den in diesem Experiment gemessenen Performanzdaten ableiten, dass dies mit großen Schwankungen in den Performanzdaten während des Trainingsprozesses einhergeht. Die Frage die sich dann stellt ist, ob und nach wie vielen Trainingsepisoden der Agent alle gewünschten Fähigkeiten sicher beherrscht. In jedem Fall ist aus dieser Argumentation zu vermuten, dass eine komplexere Umgebung mit einem verlängerten Lernprozess einhergeht.

Die in diesem Abschnitt 5.1 empirisch gefolgerten Erkenntnisse dienen der Erreichung des in Kapitel 1 formulierten Ziels einer Performanzanalyse eines DDPG-Agenten. Mehr als die bloße Interpretation der im Experiment aufgezeichneten Performanzdaten, zeigt dieser Abschnitt vor allem auf, was bei der interpretativen Analyse der Performanz von *Reinforcement Learning*-Agenten beachtet werden kann. Dies betrifft etwa die Nutzung von Testverfahren nach dem Training oder die Bedeutung von zunehmenden Umgebungseinflüssen auf die Anzahl an Trainingsepisoden und die Stabilität der Agenten-Performanz. Insbesondere können so neue und interessante Forschungsfragen generiert werden. Eine Zusammenfassung der Erkenntnisse wird in Abschnitt 5.4 angestrengt.

5.2 Vergleich verschiedener Belohnungsfunktionen

Nachdem im vorherigen Abschnitt 5.1 die Experimentergebnisse des DDPG-Agenten im *Practice*- und *Quick Race*-Modus dargestellt und interpretiert wurden, um die erste Forschungsfrage dieser Arbeit zu beantworten, wird in diesem Abschnitt nun die zweite Zielsetzung bearbeitet. Dabei sollen die Auswirkungen verschiedener Belohnungsfunktionen dargestellt, untersucht und interpretiert werden. Nach Unterabschnitt 4.2.4 spielen Belohnungs-

5 Ergebnisse und Interpretation

funktionen für die Performanz eines *Reinforcement Learning*-Agenten eine herausragende Rolle. In diesem Abschnitt sollen deshalb die drei, in Unterabschnitt 4.2.4 vorgestellten, Belohnungsfunktionen \mathcal{R}_1 , \mathcal{R}_2 und \mathcal{R}_3 hinsichtlich der resultierenden Agenten-Performanz untersucht werden. Die Methodik und Rahmenbedingungen dieser Untersuchung ähneln der aus Abschnitt 5.1. Ein DDPG-Agent wird auf der Strecke CG Speedway im *Quick Race*-Modus mit den verschiedenen Belohnungsfunktionen trainiert. Der Grund für die Auswahl des *Quick Race*-Modus, anstatt des *Practice*-Modus, ist einfach erklärbar: Da die Belohnungsfunktionen unterschiedlich viel Belohnung generieren, ist ein direkter Vergleich des Ertrages nicht möglich. Um aber weiterhin möglichst viele Performanzdaten zur Analyse nutzen zu können, bietet es sich an, die Anzahl an Kollisionen mit einzubeziehen. Auch nutzt die Belohnungsfunktion \mathcal{R}_3 den Opponents-Sensor. Eine spannende Untersuchung ergibt sich aus dem Vergleich der Anzahl der Kollisionen des DDPG-Agenten unter \mathcal{R}_1 , \mathcal{R}_2 und \mathcal{R}_3 . Erwartet wird, dass \mathcal{R}_3 hier die beste Performanz aufweist. In Abbildung H.1 in Anhang H sind die Ergebnisse des Experimentes zum Vergleich der Belohnungsfunktionen dargestellt. Aus diesen können folgende Beobachtungen gemacht werden:

1. Aus den Performanzgraphen $(d)_{\mathcal{R}_i}$ und den in Anhang H genannten statistischen Kennzahlen, kann geschlossen werden, dass mit \mathcal{R}_3 als Belohnungsfunktion der Agent tatsächlich in die wenigsten Kollisionen gerät.
2. Aus den Performanzgraphen $(b)_{\mathcal{R}_i}$ wird deutlich, dass der Agent unter der Belohnungsfunktion \mathcal{R}_2 am längsten für die erstmalige volle Umrundung der Rennstrecke benötigt. Unter \mathcal{R}_2 geschieht dies nämlich erst nach ca. 125 Episoden, während der Agent hierfür unter \mathcal{R}_1 und \mathcal{R}_3 nur ca. 60 Episoden benötigt.
3. Werden die Graphen $(b)_{\mathcal{R}_3}$ und $(d)_{\mathcal{R}_3}$ im Zusammenhang betrachtet, dann scheinen Leistungsabfälle in $(b)_{\mathcal{R}_3}$ durch $(d)_{\mathcal{R}_3}$ erklärbar, was wiederum die Folgerung der zweiten Beobachtung aus Unterabschnitt 5.1.2 bestätigt.

Die erstgenannte Beobachtung wird vor allem aus der Betrachtung der statistischen Kennzahlen der Graphen $(c)_{\mathcal{R}_1}$ und $(c)_{\mathcal{R}_3}$ aus Abbildung H.1 deutlich. So ist der arithmetische Mittelwert $\mu_{\mathcal{R}_1} = 9,58$ der Performanzdaten aus $(c)_{\mathcal{R}_1}$ höher als der arithmetische Mittelwert $\mu_{\mathcal{R}_3} = 7,74$ der Performanzdaten aus $(c)_{\mathcal{R}_3}$. Auch die Streuung der Werte um den Mittelwert ist mit $\sigma_{\mathcal{R}_1} = 9,91$ beim Agenten unter \mathcal{R}_1 höher als beim Agenten unter \mathcal{R}_3 mit $\sigma_{\mathcal{R}_3} = 8,91$. Dies macht sich auch im Performanzgraph $(d)_{\mathcal{R}_3}$ bemerkbar, der weniger fluktuiert als $(d)_{\mathcal{R}_1}$. Dies ist ein Indiz dafür, dass der Opponents-Sensor in \mathcal{R}_3 Auswirkung auf die Anzahl an Kollisionen des Agenten hat. Eine Erklärung, warum dieses Indiz plausibel ist, ist leicht herleitbar: Dadurch dass der Agent unter \mathcal{R}_3 in jedem Zeitschritt Informationen über den Abstand des eigenen Fahrzeuges zu den anderen Fahrzeugen aus der Belohnungsfunktion erhält, kann er diese Informationen in Verbindung mit den Opponents-Sensordaten bringen und eine Korrelation feststellen. Dieser, in den Gewichten der neuronalen Netze versteckte, Zusammenhang erlaubt es dem Agenten Aktionen auszuwählen, die verringern auf die Anzahl der Kollisionen wirken. Diese empirische Feststellung hat weitreichende Folgen für die Modellierung der Belohnungsfunktion. Zwar wird dem Agenten, nach Algorithmus 6, in jedem Fall eine negative Belohnung bei einer Kollision durch die Umgebung gemeldet, jedoch scheint es lohnender, die Information über den Abstand zu den anderen Fahrzeugen direkt in die Belohnungsfunktion aufzunehmen, falls Kollisionen vermieden werden sollen. Aus dieser Feststellung kann für andere Aufgaben des autonomen Fahrens perspektivisch gefolgert werden, dass beispielsweise eine Zustandsinformation über das verkehrsregelkonforme Verhalten in die Belohnungsfunktion miteinfließen sollte, anstatt nur im Fall einer Verkehrsregelverletzung dem Agenten negative Belohnung zuzuführen.

Interessanterweise scheint eine niedrigere Anzahl an Kollisionen mit einem langsameren Anstieg in der Geschwindigkeit des Agenten einherzugehen. Dies geht aus dem Vergleich von $(a)_{\mathcal{R}_1}$ und $(c)_{\mathcal{R}_1}$ mit $(a)_{\mathcal{R}_3}$ und $(c)_{\mathcal{R}_3}$ hervor. Während die Geschwindigkeit des Agenten unter \mathcal{R}_1 bereits nach ca. 250 Episoden den Wert 0,5 erreicht, dauert dies beim Agenten

unter \mathcal{R}_3 bis zur ca. 300. Episode. Der Zusammenhang einer langsameren Geschwindigkeit und weniger Kollisionen ist trivial, aber dennoch ist es erstaunlich, dass auch der DDPG-Agent diesen entdeckt hat und sich zunutze macht.

Ein Kritikpunkt dieser Auswertung mag jedoch die Annahme gleichverteilter Performanzdaten in den Performanzgraphen $(c)_{\mathcal{R}_i}$ sein. Sind diese nicht gleichverteilt, was möglicherweise der Fall ist, sollte der Median anstatt des arithmetischen Mittels genutzt werden. Eine einordnende Betrachtung der statistischen Kennzahlen von \mathcal{R}_2 , $\mu_{\mathcal{R}_2} = 6,40$ und $\sigma_{\mathcal{R}_2} = 8,28$, bestätigt dies.

Der zweitgenannten Beobachtung dieses Abschnittes liegen die Performanzgraphen $(b)_{\mathcal{R}_i}$, zur prozentual erreichten Streckenweite des Agenten, zugrunde. Während die Agenten unter \mathcal{R}_1 und \mathcal{R}_3 spätestens ab der 60. Episode eine volle Streckenumrundung schaffen, benötigt der Agent unter \mathcal{R}_2 hierfür fast 125 Episoden. Der Grund dafür ist ersichtlich, wenn sich zugegen geführt wird, dass der Agent auf der Strecke CG Speedway in einer Kurve startet, was aus Abbildung 4.2 ersichtlich ist. Da \mathcal{R}_2 , im Gegensatz zu \mathcal{R}_1 und \mathcal{R}_3 , niemals für den Querweg belohnt, wird der Agent nicht direkt für gelungende Querwegmanöver belohnt oder für missglückte Querwegmanöver bestraft werden. Diese Informationen scheinen für das schnelle Erlernen von Kurvenfahrten aber wichtig zu sein. Dem Agenten mag es unter \mathcal{R}_2 schwerer fallen, einen Zusammenhang zwischen Querführung des Fahrzeugs und Belohnung herzustellen. Diese Feststellung und Argumentation schließt sich an die der vorherigen Beobachtung an und folgert die selben Konsequenzen: Soll der Agent schnell und zuverlässig bestimmte Fahraufgaben lösen, dann sollte eine Rückmeldung über das Gelingen oder Misserfolg der Fahraufgabe in die Belohnungsfunktion aufgenommen werden.

Die drittgenannte Beobachtung lässt sich durch Betrachtung der Performanzgraphen $(b)_{\mathcal{R}_3}$ und $(d)_{\mathcal{R}_3}$ feststellen. Ist die Anzahl der Kollisionen bezogen auf eine Distanzeinheit pro Episode hoch, dann erreicht der Agent oftmals keine volle Umrundung. Sehr schön ist dieser Zusammenhang um die 150. und die 250. Episode zu erkennen. Diese Beobachtung unterstützt die Folgerungen aus der zweiten Beobachtung aus Unterabschnitt 5.1.2. Demnach folgen aus mehr Kollisionen eine verringerte zurückgelegte Distanz des Agenten.

5.3 Vergleich DDPG-Agent mit DQN-Agent

In diesem Abschnitt sollen nun die bekannten Experimentergebnisse des DDPG-Agenten mit denen eines DQN-Agenten verglichen und interpretiert werden. Der DQN-Agenten-Algorithmus zählt zu den *value-based Reinforcement Learning*-Algorithmen, da er dem *Generalized Policy Iteration*-Verfahren, das in Unterabschnitt 3.3.1 beschrieben wurde, unterliegt. Dabei wird der langfristig zu erwartende Ertrag einer Aktion in einem Zustand berechnet und die Aktion mit dem höchsten Ertrag, in diesem Zustand, wird vom Agenten ausgewählt. Da der Zustand- und Aktionenraum einer Umgebung sehr groß sein kann, parametriert der DQN-Algorithmus die *Action-Value*-Funktion, die die Erträge von Aktionen in einem Zustand berechnet. Der *actor-critic* DDPG-Agenten-Algorithmus parametriert zusätzlich die *Policy*-Funktion und kann, im Gegensatz zum DQN-Algorithmus, auch für Aktionen mit realem Wertebereich genutzt werden. Ein Vergleich dieser beiden Agenten-Algorithmen ist also auch ein Vergleich der vorherrschenden Paradigmen im *model-free Reinforcement Learning*. El Sallab et al. führen in [EAPY16] einen ähnlichen Vergleich an, der aber auf dem Vergleich des DQN-Agenten mit dem von Silver et al. stammenden DDAC-Algorithmus aus [SLH⁺14] beruht. Der DDAC-Algorithmus umfasst weder einen OU-Prozess, noch ein *Experience Replay Memory* zur Trainingsdatendekorrelation. Nach bestem Wissensstand ist zum Zeitpunkt der Abgabe der vorliegenden Arbeit in der bisherigen Literatur zum autonomen Fahren mit *Reinforcement Learning*-Algorithmen in Simulationsumgebungen noch kein direkter Vergleich von DQN- und DDPG-Algorithmus angestrengt worden. Beide Agenten werden auf allen drei Strecken unter der Belohnungsfunktion \mathcal{R}_1 im *Practice*-Modus trainiert. Die Experimentergebnisse sind in den Abbildungen I.1, I.2 und I.3 in Anhang I einzusehen. Aus diesen Ergebnissen lassen sich folgende Beobachtungen machen:

5 Ergebnisse und Interpretation

1. Der DQN-Agent braucht weniger Episoden als der DDPG-Agent, um eine Konvergenz der Geschwindigkeit zu erreichen. Dies geht aus einem Vergleich der Performanzgraphen $(d)_{DQN}$ und $(d)_{DDPG}$ in Abbildung I.1, Abbildung I.2 und I.3 hervor. Dabei ist die erreichte Höchstgeschwindigkeit des DQN-Agenten auf allen drei Strecken allerdings niedriger als die des DDPG-Agenten.
2. Aus den Performanzgraphen $(c)_{DDPG}$ in Abbildung I.1, I.2 und I.3 ist erkennlich, dass der DDPG-Agent auf allen drei Strecken eine volle Umrundung schafft. Der DQN-Agent schafft dies nur auf der Strecke CG Speedway, was aus den Performanzgraphen $(c)_{DQN}$ der drei Abbildungen hervorgeht. Auch erreicht der DDPG-Agent eine volle Umrundung in früheren Episoden als der DQN-Agent, was aus Abbildung I.1 erkennbar ist. In Abbildung I.3 ist zudem eine starke Fluktuation in den Performanzwerten des DQN-Agenten zu erkennen.

Werden die Performanzdaten der beiden Agenten-Algorithmen hinsichtlich der Entwicklung der Fahrzeuggeschwindigkeit untersucht, fällt auf, dass die Geschwindigkeit des DQN-Agent schneller gegen einen bestimmten Wert, der dann zumeist die Höchstgeschwindigkeit ist, konvergiert. So benötigt der DQN-Agent auf der Strecke CG Speedway etwa 100 Episoden zur Konvergenz, während der DDPG-Agent etwa 150 Episoden benötigt. Allerdings liegt die Höchstgeschwindigkeit des DQN-Agenten zumeist unter der des DDPG-Agenten. So erreicht der DQN-Agent auf der Strecke CG Speedway eine Höchstgeschwindigkeit von ca. 0,40, während der DDPG-Agent auf der selben Strecke eine Höchstgeschwindigkeit von ca. 0,55 erreicht. Ähnliche Beobachtungen lassen sich auf den anderen Strecken machen. Gleichwohl hätte der DQN-Agent nach Tabelle 4.3 die Möglichkeit einer Höchstbeschleunigung von 0,95, also nur knapp unter der maximal möglichen Beschleunigung. Grund für die geringere Höchstgeschwindigkeit des DQN-Agenten mag die geringere Auswahl an Lenkwerten für die Querführung des Fahrzeuges sein. So hat der DQN-Agent nach Tabelle 4.3 nur Fünf verschiedene Lenkwerte zur Auswahl, was eine erratische Fahrweise fördert und damit auch die potentielle Gefahr einer der Terminierungsbedingungen zu begegnen. Ein ähnlicher Zusammenhang wurde in Abschnitt 5.2 zwischen der Anzahl an Kollisionen und der Geschwindigkeit dargestellt. Auch hier könnte der DQN-Agent also versucht haben, die Geschwindigkeit zugunsten einer höheren Distanz zu verringern. Für diese Argumentation spricht der, dem Ertrag sehr ähnelnde, Performanzgraph zur erreichten Distanz $(b)_{DQN}$ in den Abbildungen in Anhang I sowie die Neigung des DQN-Agenten schnell gegen die Höchstgeschwindigkeit zu konvergieren und bei dieser zu verharren. Der DDPG-Agent erreicht höhere Höchstgeschwindigkeiten auf den drei Strecken als der DQN-Agent und kann, aufgrund seiner deterministischen *Policy*-Funktion, reellwertige Aktionen auswählen. Die Gefahr einer erratischen Fahrweise und damit einhergehenden Terminierung der Episode wird somit verringert. El Sallab et al. gehen in [EAPY16] spezifisch auf die weiteren Vorteile reeller Aktionenwerte, wie eine sanftere Kurvenfahrt, ein.

Der zweiten Beobachtung liegt ein Vergleich der Performanzdaten zur prozentual erreichten Streckenweite von DQN- und DDPG-Agent zugrunde. Dabei sticht hervor, dass der DQN-Agent nur auf der relativ einfachen Strecke CG Speedway eine volle Umrundung schafft, während der DDPG-Agent eine volle Umrundung auf allen drei Strecken bewältigt. Auch geschieht die erstmalige volle Umrundung der Strecke CG Speedway durch den DDPG-Agenten in früheren Episoden als beim DQN-Agenten, was aus einem Vergleich von $(c)_{DQN}$ mit $(c)_{DDPG}$ in Abbildung I.1 hervorgeht. Überdies fluktuiieren die Performanzwerte des DQN-Agenten stärker als die des DDPG-Agenten, was insbesondere aus Abbildung I.3 hervorgeht. Ein Grund für die deutlich schlechtere Performanz des DQN-Agenten gegenüber dem DDPG-Agenten, was die erreichte Streckenweite und Stabilität betrifft, kann das zu kleinen und flachen neuronale Netz des DQN-Agenten sein. Dieses hat nach Abbildung C.1 nur drei verdeckte Schichten mit insgesamt 1.600 Neuronen. Neuronale Netze mit nur wenig verdeckten Schichten und wenig Neuronen lernen komplexe Zusammenhänge aus den Eingangsdaten schlechter und tendieren zum *Underfitting*. Dieser Mangel an Abstraktionsvermögen eines

neuronalen Netzes kann sich im *Deep Reinforcement Learning* durch schlechte Performanzergebnisse des Agenten ausdrücken. Der DQN-Agent versucht nach Unterabschnitt 3.3.4 mit einer parametrierten *Action-Value*-Funktion die wahre *Action-Value*-Funktion der Umgebung zu approximieren, $\hat{q}(s, a, \mathbf{w}) \approx q_\pi(s, a)$. Ist die wahre *Action-Value*-Funktion der Umgebung zu komplex für das neuronale Netz des DQN-Agenten, dann wird der Agent diese nie ausreichend gut approximieren können. [SB17, S. 323] Die schlechte Performanz des DQN-Agenten hinsichtlich der prozentual erreichten Streckenweite und hinsichtlich der hohen Varianz der Performanzwerte, könnte ein Indiz dafür sein. Die neuronalen Netze des DDPG-Agenten sind ebenfalls mit nur wenig verdeckten Schichten ausgestattet, jedoch wird statt der *Action-Value*-Funktion die *Policy*-Funktion parametriert. Die optimale *Policy*-Funktion kann durchaus einfacher als die wahre *Action-Value*-Funktion sein und so sollte man nach Sutton und Barto [SB17, S. 323] für jede Umgebung eine empirisch fundierte Abwägung zwischen *value-based*- und *actor-critic/policy-based*-Algorithmen treffen. Im vorliegenden Anwendungsfall der autonomen Quer- und Längsführung eines Fahrzeuges in der Simulationsumgebung TORCS, stellt sich eine Approximation der *Policy*-Funktion durch den *actor-critic* DDPG-Algorithmus als performanter heraus.

5.4 Zusammenfassung und Erkenntnisse im Vergleich

Die in den vorherigen Abschnitten dieses Kapitels gemachten Erkenntnisse sollen nun, hinsichtlich der in Kapitel 1 genannten Ziele dieser Arbeit, zusammengefasst werden:

(1.) Auswertung der Performanzmetriken des DDPG-Agenten

In Abschnitt 5.1 sind die Ergebnisse und Ergebnisinterpretationen der Performanzdaten des DDPG-Agenten dargestellt. Es wurde deutlich, dass auf allen drei Strecken ein positiver Lernfortschritt des DDPG-Agenten festzustellen ist. Performanzeinbrüche während des Trainings können im *Practice*-Modus mit der fortwährenden Erkundung der Umgebung erklärt werden und im *Quick Race*-Modus zusätzlich mit den Kollisionen des Agenten mit anderen Fahrzeugen. Dabei sticht heraus, dass der Agent selbst mit der simplen Belohnungsfunktion \mathcal{R}_1 in der Lage ist, komplexe Ausweichmanöver zu erlernen, was aus der sinkenden Anzahl von Kollisionen pro Episode während des Trainings hervorgeht. Auch wurde vorgeschlagen, dass Performanzmetriken für die Auswertung genutzt werden sollten, die nicht nur den aggregierten Effekt der Agenten-Performanz zeigen, sondern spezifisch auf die, in der Belohnungsfunktion befindlichen, Einflussgrößen ausgerichtet sind. Als Beispiel einer solchen spezifischen Performanzmetrik wurde der Längsweg aus \mathcal{R}_1 oder die Anzahl an Kollisionen aus \mathcal{R}_3 genannt. Spezifische Performanzmetriken erleichtern die Analyse und Interpretation der so wichtigen Ertrag-pro-Episode-Performanzmetrik.

Überdies wurde im Test festgestellt, dass der DDPG-Agent *overfitted*, da er während des Trainings gute Performanzdaten aufweist, aber im Test deutlich versagt. Dies geht insbesondere aus den Performanzgraphen zur prozentual erreichten Streckenweite hervor. Für die frühzeitige Erkennung von *Over-* oder *Underfitting* wurde deshalb das *Cross-Validation*-Verfahren vorgeschlagen.

(2.) Auswirkung der unterschiedlichen Belohnungsfunktionen

Der Vergleich unterschiedlicher Belohnungsfunktionen in Abschnitt 5.2 hat gezeigt, dass eine stetige Rückmeldung über das Gelingen oder Misserfolg von Fahraufgaben in der Belohnungsfunktion zu besseren Ergebnissen für diese Fahraufgaben führt. So wurde anhand eines Vergleiches von \mathcal{R}_1 und \mathcal{R}_3 , die sich nur in der stetigen Penalisierung von zu nahem Auffahren unterscheiden, gezeigt, dass der DDPG-Agent unter \mathcal{R}_3 in weniger Kollisionen verwickelt wird. Weiterhin wurde mit \mathcal{R}_2 gezeigt, dass durch das Fehlen der Rückmeldung über den Querweg in der Belohnungsfunktion \mathcal{R}_2 , der Agent länger für das Erlernen einer Kurvenfahrt braucht.

(3.) Vergleich DDPG- mit DQN-Agent

In Abschnitt 5.3 wurde der DDPG-Agent mit einem DQN-Agenten, hinsichtlich der gemessenen Performanzdaten aus dem Experiment, verglichen. Die Interpretation der Geschwindigkeitsdaten hat zur Vermutung geführt, dass der DQN-Agent aufgrund der wenigen, diskreten Lenkwerte vorsichtshalber langsamer als der DDPG-Agent fährt. Dies offenbart auch die Schwierigkeit des Vergleichs von einem DQN- mit einem DDPG-Agenten, da sich die Performanzunterschiede aus den konzeptionellen Eigenheiten der Algorithmen, wie der Diskretisierung des Aktionenraumes beim DQN-Agenten, ergeben. Werden der Performanzuntersuchung gleiche Rahmenbedingungen zugrunde gelegt und die neuronalen Netze der Agenten ähnlich aufgebaut, dann geht aus dem Experiment dieser Arbeit hervor, dass der DDPG-Agent bessere Performanzergebnisse erzielt. Dies wurde mit den besseren Messergebnissen zur Performanzmetrik der prozentual erreichten Streckenweite pro Episode empirisch dargelegt. Als Grund für die bessere Performanz des DDPG-Agenten wurde die einfachere Approximation der quasi-optimalen *Policy*-Funktion für die Quer- und Längsführung eines Fahrzeugs in TORCS vermutet.

Weitere Erkenntnisse

In Unterabschnitt 5.1.1 wurde dargelegt, dass eine Interpretation der Performanzmetrik zum Ertrag pro Episode vor allem im Zusammenhang mit spezifischen Performanzmetriken Sinn ergibt. Erst so wird erkennbar, wie sich der Ertrag zusammensetzt und in welchen Situationen ein Agent Schwächen und Stärken aufweist. Da sich der Ertrag aus der Belohnungsfunktion ergibt, stellt sich die Frage, welche Einflussgrößen, also Sensordaten, in die Performanzmetrik einfließen sollen. Nach Unterabschnitt 5.1.1 sollten alle Einflussgrößen, die eine Auswirkung auf die durchzuführenden Fahraufgaben des Agenten haben, mit einfließen. Wird die Umgebung immer realistischer und damit komplexer, dann steigt die Anzahl an Einflussgrößen schnell an. So müssten dann etwa die Einflussgrößen der Quer- und Längsführung, des korrekten Abbiegeverhaltens, der Einhaltung von Geschwindigkeitsbegrenzungen, des Anhaltens vor Fußgängerüberwegen und vieler anderer Fahraufgaben in die Belohnungsfunktion aufgenommen werden. Die Einbeziehung von Einflussgrößen von all diesen Fahraufgaben in die Belohnungsfunktion würde sich als sehr aufwendig herausstellen. Deshalb wurde in Unterabschnitt 5.1.1 ein *Inverse Reinforcement Learning*-Ansatz vorgeschlagen, bei dem aus der *Policy* eines menschlichen Fahrers und den Umgebungsständen auf eine Belohnungsfunktion geschlossen wird. Die handische Modellierung einer Belohnungsfunktion entfällt somit. In der Folge könnte ein *Reinforcement Learning*-Algorithmus, möglicherweise sogar als ein DDPG-Agent, unter dieser Belohnungsfunktion eine *Policy* erlernen, die dann der des menschlichen Fahrer entsprechen sollte.

Abschließend sollen die Erkenntnisse dieses Experiments kurz mit den Erkenntnissen der bisherigen Forschung zum autonomen Fahren mit *Reinforcement Learning* aus Kapitel 2 verglichen werden. Jaritz et al. kommen in [JdT⁺18] beim Vergleich unterschiedlicher Belohnungsfunktionen zu den gleichen Erkenntnissen wie diese Arbeit. Demnach folgt aus mehr Kollisionen eine verringerte, durchschnittliche Geschwindigkeit des Agenten.

Wie bereits erwähnt, kommen El Sallab et al. in [EAPY16] zur Erkenntnis, dass ein deterministischer *Policy Gradient*-Algorithmus zu einer ruhigeren Kurvenfahrt führt. Diese Erkenntnis ist auch für den Vergleich von DQN- mit DDPG-Agent anzunehmen, wird jedoch in der vorliegenden Arbeit nicht weiter untersucht.

Wolf et al. nutzen in [WHW⁺17] ebenfalls einen Trainings- und ein Testdatensatz, legen den Fokus ihrer Schlussfolgerungen diesbezüglich aber eher auf die praktische Evaluation ihrer Untersuchungsgegenstände, ohne näher auf *Overfitting* einzugehen.

6 Ausblick

In diesem Kapitel soll ein kurzer Ausblick gegeben werden. Sowohl über weiterführende Forschungsfragen, als auch über die Einbettung der Erkenntnisse in einen ethischen Kontext.

6.1 Weiterführende Forschungsfragen

Die gekonnte Durchfahrt enger Kurven oder das antizipatorische Ausweichen von Kollisionen des vom *Reinforcement Learning*-Agenten gesteuerten Fahrzeugbots scheinen bemerkenswert, jedoch bringen die, in dieser Arbeit vorgestellten, Algorithmen auch Nachteile und Übersimplifizierungen mit. So liegen den Agenten-Algorithmen sehr simple und händisch erstellte Belohnungsfunktionen zugrunde, die in der Hauptsache eine gute Quer- und Längsführung belohnen und höchstens noch ein zu dichtes Auffahren bestrafen. Für einen Einsatz in realen Fahrzeugen ist diese Belohnungsfunktion zu simpel. Als mögliche Lösung wurde die Nutzung eines *Inverse Reinforcement Learning*-Algorithmus vorgeschlagen, der aus einer *Policy* eine Belohnungsfunktion ableitet. Auch Talpaert et al. schlagen in [TSK⁺19] diesen Ansatz vor und Vasquez et al. [VYKL14] evaluieren, ob dieser Ansatz zu einer Replikation von menschlichem Fahrverhalten führt. Eine dedizierte Untersuchung in TORCS hierzu wäre interessant. Problem dabei ist allerdings, dass das zweistufige Vorgehen, bestehend aus dem Erlernen einer Belohnungsfunktion und dem Erlernen einer darauf basierenden *Policy*, eine gut getarnte Art des *Supervised Learning* sein dürfte und sich damit die Frage stellt, ob nicht gleich ein bewährter *Supervised Learning*-Ansatz, wie aus [BTD⁺16], vorzuziehen ist. In jedem Fall wird sich aber immer die Frage nach den zu verwendenden Sensorik stellen, insbesondere vor den wirtschaftlichen Hintergründen der Materialkosten der Sensoren. Deshalb wäre auch eine Untersuchung über die Performanz von Agenten-Algorithmen, bei Nutzung unterschiedlicher Sensoren, interessant.

Ein weiteres Problem, mit den in dieser Arbeit genutzten Algorithmen, ist das sehr simple Erkundungsverfahren für den Zustand- und Aktionenraum. So wird über das abschwellende ϵ -Parameter festgelegt, ob der Agent eine zufällige Aktion zur Erkundung auswählt oder wie stark die, von der *Policy* bestimmte, Aktion rauschbehaftet ist. Praktisch wird am Anfang des Trainings sehr viel erkundet und am Ende des Trainings kaum bis gar nicht. Diese Modellierung der Erkundung neuer Zustände entspricht nicht dem natürlichen Verhalten von Menschen und Tieren bei der Erkundung ihrer Umgebung. Deshalb haben Pathak et al. in [PAED17] das Konzept einer von der Neugierde geleiteten Erkundung (engl. *curiosity-driven exploration*) entwickelt, das von Burda et al. in [BESK18] verbessert wurde. Dabei sagt ein neuronales Netz den nächsten Zustand der Umgebung voraus und das Fehlermaß, bezogen auf den tatsächlich nächsten Zustand, wird berechnet. Ist das Fehlermaß hoch, hat sich der Agent noch nicht häufig in diesem nächsten Zustand befunden und der Gradient der *Policy*-Funktion wird entsprechend stark skaliert. *Curiosity-driven Exploration* nimmt allerdings sehr spärliche Belohnungen an. In der vorliegenden Arbeit wird dem Agenten jedoch in jedem Zeitschritt eine Belohnung zugeführt. So stellt sich die Frage, inwiefern die Vorteile der *curiosity-driven Exploration* mit denen der in dieser Arbeit vorgestellten Algorithmen in Verbindung gebracht werden können.

Überdies ergeben sich weiterführende Fragestellungen aus der Anwendung eines Training-

6 Ausblick

Test-Paradigmas für die Auswertung von Agenten-Algorithmen. Das Testen von trainierten Agenten wird nach [ZSSH19] in der bisherigen Forschung nicht häufig genug umgesetzt. In Unterabschnitt 5.1.1 wurde hierfür das *Cross-Validation*-Verfahren vorgeschlagen.

6.2 Ethische Aspekte und Sicherheitsfragen

Die bisher rein technische Vorstellung autonomer Fahrzeuge soll an dieser Stelle kurz um ethische, soziale und sicherheitstechnische Aspekte und Fragestellungen ergänzt werden. Grund dafür ist das notwendige Mitwirken von Ingenieuren und Informatikern bei der Implementierung von Lösungsansätzen zu derlei Fragestellungen.

Eine bekannte ethische Fragestellung wird durch das *Trolley Problem* aufgeworfen, bei der ein autonomes Fahrzeug die Entscheidung über das Überleben des Fahrer oder einer Fußgängergruppe treffen muss. [HDP18] Die Frage die sich dabei stellt ist, wie und welche moralischen Prinzipien zur Entscheidungsfindung in diesem Dilemma genutzt werden können. Im *Reinforcement Learning* könnten einem Agenten moralische Prinzipien über eine entsprechende Belohnungsfunktion vorgegeben werden. [KHJ⁺18] Problem dabei ist aber, dass diese Belohnungsfunktion zumeist wieder menschengemacht ist und damit lediglich die Wertvorstellungen ihrer Entwickler abbildet. [HDP18] Holstein et al. argumentieren in [HDP18], dass derlei idealisierte ethische Dilemma die eigentlichen Fragestellungen verschleieren und der Fokus auf die praktische Ethik gelegt werden sollte. Demnach sollten Algorithmen für das autonome Fahren nicht als *Black Box* behandelt werden, sondern vorher vereinbarten, transparenten ethischen Richtlinien folgen. Eine, auch für die vorliegende Arbeit, interessante Feststellung, da der Ende-Zu-Ende-Ansatz mittels *Reinforcement Learning* doch genau diese *Black Box* zum Ziel hat. Zwar lassen sich durch Hyperparametereinstellungen und die Belohnungsfunktion Einfluss auf die Algorithmen ausüben, jedoch müssten die Algorithmen dann stets auf durchgängige Konformität mit den ethischen Richtlinien getestet werden. Die Frage nach einem transparenten Testverfahren für die Richtlinienkonformität ist die Folgerung dieser Argumentation. Eine Möglichkeit Transparenz in kamerabasierten Fahrzeugen herzustellen, wird in [JdT⁺18] dargestellt und beruht auf dem Visualisierungsalgorithmus wichtiger Bildobjekte von Simonyan et al. aus [SVZ13]. Für die in dieser Arbeit vorgestellten *Reinforcement Learning*-Agenten wird vorgeschlagen, die getroffenen ethischen Vereinbarungen in die Belohnungsfunktion miteinfließen zu lassen und die Agenten anschließend umfänglich auf Konformität zu testen. Wird, wie in Kapitel 5 und Abschnitt 6.1 vorgeschlagen, ein *Inverse Reinforcement Learning*-Algorithmus zur Erstellung einer Belohnungsfunktion genutzt, gibt der menschliche Fahrer die Ethik vor.

Auch die klassischen Schutzziele der Informationssicherheit: Vertraulichkeit, Integrität und Verfügbarkeit müssen für autonome Fahrsysteme gewährleistet werden. In [HDP18] werden die Anforderungen an und Lösungsansätze für die *Safety* und die *Security* von autonomen Fahrzeugen übersichtlich aufgelistet. Darunter fällt etwa die Vertraulichkeit der Fahrtdata der Passagiere und ebenso die Vertraulichkeit der, aus den Sensordaten entnehmbaren, Bewegungs- und Aufenthaltsdaten von Passanten und anderen Verkehrsteilnehmern. In [WHW⁺17] untersuchen Wolf et al. explizit eine Belohnungsfunktion, die ein Fahrverhalten des Agenten ermöglicht, das den Agenten wie ein öffentlichen Verkehrsteilnehmer fahren lässt. Die Notwendigkeit einer solchen sicherheitsgarantierenden Belohnungsfunktion ist im Kontext der Diskussion um Performanzstabilität und *Overfitting* aus Kapitel 5 von besonderer Bedeutung. Von autonomen Fahrzeugen wird schließlich ein Rückgang der Anzahl der Verkehrsunfälle und -opfer erwartet. [Sch17] Vergleiche zwischen einem menschlichen Fahrer und einem *Reinforcement Learning*-Agenten hinsichtlich bestimmter Sicherheitsmetriken, wie der Anzahl der Unfälle, sind nach derzeitigem Kenntnisstand und bestem Wissen in der Forschung noch unzureichend untersucht. Die Auswertungen in Unterabschnitt 5.1.2 und Abschnitt 5.3 geben einen Einblick in die möglicherweise zu verwendenden Performanzmetriken und zeigen auf, dass ein *Reinforcement Learning*-Agent mit der Zeit lernen kann, Unfälle zu vermeiden.

7 Fazit

The greatest deception men suffer
is from their own opinions.

L. Da Vinci

In der vorliegenden Arbeit wurden die in Kapitel 1 genannten Ziele bearbeitet. Diese wurden vor dem Hintergrund einer Evaluation von *Reinforcement Learning*-Algorithmen im Kontext autonomer Fahrzeuge formuliert.

Zunächst wurde deshalb ein DDPG-Agenten-Algorithmus, hinsichtlich bestimmter Performanzmetriken, untersucht. So konnte gezeigt werden, dass der DDPG-Agent die Quer- und Längsführung des Fahrzeuges erfolgreich erlernen kann. Außerdem konnte gezeigt werden, dass der Agent lernt, Kollisionen mit anderen Fahrzeugen zu vermeiden. Da der Agent gute Trainingsergebnisse auf komplexen Strecken aufweist, aber auf einfachen Strecken im Test versagt, wurde zwecks der Vermeidung von *Overfitting* die Nutzung des *Cross-Validation*-Verfahrens vorgeschlagen.

Als nächstes wurde die Performanz des DDPG-Agenten unter verschiedenen Belohnungsfunktionen untersucht. Es wurde gezeigt, dass eine stetige Rückmeldung über das Gelingen oder Mislingen einer Fahraufgabe in der Belohnungsfunktion zu besseren Ergebnissen für diese Fahraufgabe führt. Konkret konnte durch das Hinzufügen einer stetigen Rückmeldung über den Fahrzeugabstand des Agentenfahrzeugs zu anderen Fahrzeugen die Anzahl an Kollisionen gesenkt werden. Wird dagegen die stetige Rückmeldung über den Querweg des Agentenfahrzeugs aus der Belohnungsfunktion entfernt, braucht der Agent länger zum Erlernen der Kurvenfahrt.

Ein direkter Vergleich des *actor-critic* DDPG-Agenten mit einem *value-based* DQN-Agenten, im Kontext des autonomen Fahrens innerhalb der Simulationsumgebung TORCS, hat ergeben, dass der DDPG-Agent bessere Performanzergebnisse erzielt als der DQN-Agent. Als eine mögliche Begründung hierfür wurde die einfachere Approximation der quasi-optimalen *Policy*-Funktion angeführt. Demnach sind *actor-critic Reinforcement Learning*-Algorithmen, für die Aufgabenstellung des autonomen Fahrens in Simulationsumgebungen, vorzuziehen. Diese Arbeit ist nach bestem Wissen die zum heutigen Zeitpunkt erste, die diese beiden Agenten-Algorithmen im Kontext des autonomen Fahrens verglichen hat.

Durch diese Untersuchungen wurden nicht nur konkrete Antworten zu den anfänglich formulierten Forschungsfragen gefunden, sondern es wurde auch aufgezeigt, was bei einer Evaluation von *Reinforcement Learning*-Algorithmen zu beachten ist. Darunter fällt etwa die Evaluation des trainierten Agenten auf Teststrecken, die durchdachte Ausgestaltung der Belohnungsfunktion und die Definition von nützlichen Performanzmetriken.

Überdies ergeben sich aus den Erkenntnissen dieser Arbeit weitere Fragestellungen, wie die nach einer effizienteren Erkundung des Zustand- und Aktionenraumes und der Nutzung eines *Inverse Reinforcement Learning*-Algorithmus zur Erstellung der Belohnungsfunktion. In jedem Fall ist zu vermuten, dass die Erforschung des *Reinforcement Learning* für das autonome Fahren weitergeht. Ob sich der *Reinforcement Learning*-Ansatz unter den in Kapitel 2 dargestellten Verfahren und Ansätzen durchsetzt, hängt auch von den empirischen Ergebnissen der weiteren Forschung ab.

Literaturverzeichnis

- [AAB⁺15] ABADI, Martín ; AGARWAL, Ashish ; BARHAM, Paul ; BREVDO, Eugene ; CHEN, Zhifeng ; CITRO, Craig ; CORRADO, Greg S. ; DAVIS, Andy ; DEAN, Jeffrey ; DEVIN, Matthieu ; GHEMAWAT, Sanjay ; GOODFELLOW, Ian ; HARP, Andrew ; IRVING, Geoffrey ; ISARD, Michael ; JIA, Yangqing ; JOZEFOWICZ, Rafal ; KAISER, Lukasz ; KUDLUR, Manjunath ; LEVENBERG, Josh ; MANÉ, Dan-delion ; MONGA, Rajat ; MOORE, Sherry ; MURRAY, Derek ; OLAH, Chris ; SCHUSTER, Mike ; SHLENS, Jonathon ; STEINER, Benoit ; SUTSKEVER, Ilya ; TALWAR, Kunal ; TUCKER, Paul ; VANHOUCKE, Vincent ; VASUDEVAN, Vijay ; VIÉGAS, Fernanda ; VINYALS, Oriol ; WARDEN, Pete ; WATTENBERG, Martin ; WICKE, Martin ; YU, Yuan ; ZHENG, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. <https://www.tensorflow.org/>. Version: 2015. – Software erhältlich unter tensorflow.org.
- [AGR⁺16] ALAHI, Alexandre ; GOEL, Kratarth ; RAMANATHAN, Vignesh ; ROBICQUET, Alexandre ; FEI-FEI, Li ; SAVARESE, Silvio: Social LSTM: Human Trajectory Prediction in Crowded Spaces. In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, S. 961–971.
- [BESK18] BURDA, Yuri ; EDWARDS, Harrison ; STORKEY, Amos ; KLIMOV, Oleg: Exploration by Random Network Distillation. In: *arXiv e-prints* (2018), Oktober, S. arXiv:1810.12894.
- [Bim15] BIMBRAW, K.: Autonomous cars: Past, present and future a review of the developments in the last century, the present scenario and the expected future of autonomous vehicle technology. In: *2015 12th International Conference on Informatics in Control, Automation and Robotics (ICINCO)* Bd. 01, 2015, S. 191–198.
- [BL17] BUDUMA, Nikhil ; LOCASCIO, Nicholas: *Fundamentals of Deep Learning: Designing Next-Generation Machine Intelligence Algorithms*. Erste Auflage. O'Reilly Media, Inc., 2017. – ISBN 978-1-491-92561-4
- [BNVB13] BELLEMARE, M. G. ; NADDAF, Y. ; VENESS, J. ; BOWLING, M.: The Arcade Learning Environment: An Evaluation Platform for General Agents. In: *Journal of Artificial Intelligence Research* 47 (2013), S. 253–279.
- [BTD⁺16] BOJARSKI, Mariusz ; TESTA, Davide D. ; DWORAKOWSKI, Daniel ; FIRNER, Bernhard ; FLEPP, Beat ; GOYAL, Prasoon ; JACKEL, Lawrence D. ; MONFORT, Mathew ; MULLER, Urs ; ZHANG, Jiakai ; ZHANG, Xin ; ZHAO, Jake ; ZIEBA, Karol: End to End Learning for Self-Driving Cars. In: *CoRR* abs/1604.07316 (2016). <http://arxiv.org/abs/1604.07316>
- [BYC⁺17] BOJARSKI, Mariusz ; YERES, Philip ; CHOROMANSKA, Anna ; CHOROMANSKI, Krzysztof ; FIRNER, Bernhard ; JACKEL, Lawrence D. ; MULLER, Urs: Explaining how a Deep Neural Network Trained with End-to-End Learning Steers a Car. In: *CoRR* abs/1704.07911 (2017). <http://arxiv.org/abs/1704.07911>

- [C⁺15] CHOLLET, François u. a.: *Keras: The Python Deep Learning library*. <https://keras.io>, 2015.
- [CKH⁺18] COBBE, Karl ; KLIMOV, Oleg ; HESSE, Chris ; KIM, Taehoon ; SCHULMAN, John: Quantifying Generalization in Reinforcement Learning. In: *arXiv e-prints* (2018), S. arXiv:1812.02341.
- [CM17] CHI, Lu ; MU, Yadong: Deep Steering: Learning End-to-End Driving Model from Spatial and Temporal Visual Cues. In: *CoRR* abs/1708.03798 (2017). <http://arxiv.org/abs/1708.03798>
- [CSKX15] CHEN, Chenyi ; SEFF, Ari ; KORNHAUSER, Alain ; XIAO, Jianxiong: DeepDriving: Learning Affordance for Direct Perception in Autonomous Driving. In: *arXiv e-prints* (2015), Mai, S. arXiv:1505.00256.
- [DCH⁺16] DUAN, Yan ; CHEN, Xi ; HOUTHOOFDT, Rein ; SCHULMAN, John ; ABBEEL, Pieter: Benchmarking Deep Reinforcement Learning for Continuous Control. In: *arXiv e-prints* (2016), April, S. arXiv:1604.06778.
- [DRC⁺17] DOSOVITSKIY, Alexey ; ROS, German ; CODEVILLA, Felipe ; LOPEZ, Antonio ; KOLTUN, Vladlen: CARLA: An Open Urban Driving Simulator. In: *arXiv e-prints* (2017), November, S. arXiv:1711.03938.
- [EAPY16] EL SALLAB, Ahmad ; ABDOU, Mohammed ; PEROT, Etienne ; YOGAMANI, Senthil: End-to-End Deep Reinforcement Learning for Lane Keeping Assist. In: *arXiv e-prints* (2016), Dezember, S. arXiv:1612.04340.
- [EAPY17] EL SALLAB, Ahmad ; ABDOU, Mohammed ; PEROT, Etienne ; YOGAMANI, Senthil: Deep Reinforcement Learning framework for Autonomous Driving. In: *arXiv e-prints* (2017), April, S. arXiv:1704.02532.
- [Edw17] EDWARDS, Chris: *SnakeOil - SnakeOil Virtual Motor Sports Lubricants*. <http://xed.ch/project/snakeoil/>, 2017. – Zuletzt abgerufen am 26.03.2019.
- [FKPG03] FAHRMEIER, Ludwig ; KÜNSTLER, Rita ; PIGEOT, Iris ; GERHARD, Tutz: *Statistik: der Weg zur Datenanalyse*. Vierte Auflage. Berlin Heidelberg New York : Springer Verlag, 2003. – ISBN 3-540-44000-3
- [FZL⁺18] FAN, Haoyang ; ZHU, Fan ; LIU, Changchun ; ZHANG, Liangliang ; ZHUANG, Li ; LI, Dong ; ZHU, Weicheng ; HU, Jiangtao ; LI, Hongye ; KONG, Qi: Baidu Apollo EM Motion Planner. In: *CoRR* abs/1807.08048 (2018). <http://arxiv.org/abs/1807.08048>
- [GAA⁺12] GASSER, Tom M. ; ARZT, Clemens ; AYOUBI, Mihiar ; BARTELS, Arne ; BÜRKLE, Lutz ; EIER, Jana ; FLEMISCH, Frank ; HÄCKER, Dirk ; HESSE, Tobias ; HUBER, Werner ; LOTZ, Christine ; MAURER, Markus ; RUTH-SCHUMACHER, Simone ; SCHWARZ, Jürgen ; VOGT, Wolfgang ; STRASSENWESEN, Bundesanstalt für (Hrsg.): *Rechtsfolgen zunehmender Fahrzeugautomatisierung*. Heft F 83. Bremerhaven : Wirtschaftsverlag NW, 2012. <https://bast.opus.hbz-nrw.de/opus45-bast/frontdoor/deliver/index/docId/541/file/F83.pdf>
- [GBC16] GOODFELLOW, Ian ; BENGIO, Yoshua ; COURVILLE, Aaron: *Deep Learning*. MIT Press, 2016. – <http://www.deeplearningbook.org>
- [GCSX16] GANESH, Adithya ; CHARALEL, Joe ; SARMA, Matthew D. ; XU, Nancy: Deep Reinforcement Learning for Simulated Autonomous Driving. (2016)

LITERATURVERZEICHNIS

- [GKB⁺16] GURGHIAN, A. ; KODURI, T. ; BAILUR, S. V. ; CAREY, K. J. ; MURALI, V. N.: DeepLanes: End-To-End Lane Position Estimation Using Deep Neural Networks. In: *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, 2016. – ISSN 2160–7516, S. 38–45.
- [Gru14] GRUBER, Manfred: *Ornstein-Uhlenbeck-Prozesse*. Hochschule München Vorlesungsskript 7: Stochastic Processes in Risk and Finance. <http://gruber.userweb.mwn.de/14.stochproc/14.stochproc.v07.pdf>. Version: April 2014. – Zuletzt abgerufen am 05.04.2019.
- [HDP18] HOLSTEIN, Tobias ; DODIG-CRNKOVIC, Gordana ; PELLICCIONE, Patrizio: Ethical and Social Aspects of Self-Driving Cars. In: *arXiv e-prints* (2018), Februar, S. arXiv:1802.04103.
- [Hon16] HONKANEN, Jari: *MEMS and Sensors in Automotive Applications on the Road to Autonomous Vehicles: HUD and ADAS*. MicroVision Präsentation, November 2016.
- [HWT⁺15] HUVAL, Brody ; WANG, Tao ; TANDON, Sameep ; KISKE, Jeff ; SONG, Will ; PAZHAYAMPALLIL, Joel ; ANDRILUKA, Mykhaylo ; RAJPURKAR, Pranav ; MIGIMATSU, Toki ; CHENG-YUE, Royce ; MUJICA, Fernando ; COATES, Adam ; NG, Andrew Y.: An Empirical Evaluation of Deep Learning on Highway Driving. In: *CoRR* abs/1504.01716 (2015). <http://arxiv.org/abs/1504.01716>
- [JdT⁺18] JARITZ, Maximilian ; DE CHARETTE, Raoul ; TOROMANOFF, Marin ; PEROT, Etienne ; NASHASHIBI, Fawzi: End-to-End Race Driving with Deep Reinforcement Learning. In: *arXiv e-prints* (2018), Juli, S. arXiv:1807.02371.
- [JSD⁺14] JIA, Yangqing ; SHELHAMER, Evan ; DONAHUE, Jeff ; KARAYEV, Sergey ; LONG, Jonathan ; GIRSHICK, Ross ; GUADARRAMA, Sergio ; DARRELL, Trevor: Caffe: Convolutional Architecture for Fast Feature Embedding. In: *arXiv preprint arXiv:1408.5093* (2014)
- [KHJ⁺18] KENDALL, Alex ; HAWKE, Jeffrey ; JANZ, David ; MAZUR, Przemyslaw ; REDA, Daniele ; ALLEN, John-Mark ; LAM, Vinh-Dieu ; BEWLEY, Alex ; SHAH, Amar: Learning to Drive in a Day. In: *CoRR* abs/1807.00412 (2018). <http://arxiv.org/abs/1807.00412>
- [Kie18] KIERFELD, Jan: *Thermodynamik und Statistik*. Technische Universität Dortmund Vorlesungsskript zum Modul Thermodynamik und Statistik. http://t1.physik.tu-dortmund.de/files/kierfeld/teaching/LectureNotes/kierfeld_TuS.pdf. Version: Januar 2018. – Zuletzt abgerufen am 09.04.2019.
- [Lap18] LAPAN, Maxim: *Deep Reinforcement Learning Hands-On - Apply Modern RL Methods, with Deep Q-Networks, Value Iteration, Policy Gradients, TRPO, AlphaGo Zero and More*. Birmingham : Packt Publishing, 2018. – ISBN 978–1–788–83424–7
- [Lau16] LAU, Ben: *Using Keras and Deep Deterministic Policy Gradient to play TORCS*. <https://openai.com/blog/introducing-openai/>, Oktober 2016. – Zuletzt abgerufen am 15.04.2019.
- [LCL13] LOIACONO, Daniele ; CARDAMONE, Luigi ; LANZI, Pier L.: Simulated Car Racing Championship: Competition Software Manual. In: *arXiv e-prints* (2013), April, S. arXiv:1304.1672.

- [LHP⁺16] LILLICRAP, Timothy P. ; HUNT, Jonathan J. ; PRITZEL, Alexander ; HEESS, Nicolas ; EREZ, Tom ; TASSA, Yuval ; SILVER, David ; WIERSTRA, Daan: Continuous control with deep reinforcement learning. In: *CoRR* abs/1509.02971v5 (2016). <http://arxiv.org/abs/1509.02971v5>
- [LMB⁺05] LECUN, Yann ; MULLER, Urs ; BEN, Jan ; COSATTO, Eric ; FLEPP, Beat: Off-road Obstacle Avoidance Through End-to-end Learning. In: *Proceedings of the 18th International Conference on Neural Information Processing Systems*. Cambridge, MA, USA : MIT Press, 2005 (NIPS'05), S. 739–746.
- [LZZC18] LI, Dong ; ZHAO, Dongbin ; ZHANG, Qichao ; CHEN, Yaran: Reinforcement Learning and Deep Learning based Lateral Control for Autonomous Driving. In: *CoRR* abs/1810.12778 (2018). <http://arxiv.org/abs/1810.12778>
- [Mau16] MAURER, Markus ; MAURER, Markus (Hrsg.) ; GERDES, J. C. (Hrsg.) ; LENZ, Barbara (Hrsg.) ; WINNER, Hermann (Hrsg.): *Autonomous Driving: Technical, Legal and Social Aspects - Introduction*. Berlin, Heidelberg : Springer Berlin Heidelberg, 2016. – S. 1–7. – ISBN 978-3-662-48847-8
- [MBB⁺08] MONTEMERLO, Michael ; BECKER, Jan ; BHAT, Suhrid ; DAHLKAMP, Hendrik ; DOLGOV, Dmitri ; ETTINGER, Scott ; HAEHNEL, Dirk ; HILDEN, Tim ; HOFFMANN, Gabriel ; HUHNKE, Burkhard ; JOHNSTON, Doug ; KLUMPP, Stefan ; LANGER, Dirk ; LEVANDOWSKI, Anthony ; LEVINSON, Jesse ; MARCIL, Julien ; ORENSTEIN, David ; PAEFGEN, Johannes ; PENNY, Isaac ; THRUN, Sebastian: Junior: The Stanford Entry in the Urban Challenge. In: *Journal of Field Robotics* 25 (2008), September, S. 569 – 597. <http://dx.doi.org/10.1002/rob.20258>. – DOI 10.1002/rob.20258
- [MBM⁺16] MINIH, Volodymyr ; BADIA, Adrià P. ; MIRZA, Mehdi ; GRAVES, Alex ; LILLICRAP, Timothy P. ; HARLEY, Tim ; SILVER, David ; KAVUKCUOGLU, Koray: Asynchronous Methods for Deep Reinforcement Learning. In: *CoRR* abs/1602.01783 (2016). <http://arxiv.org/abs/1602.01783>
- [MKS⁺13] MINIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; GRAVES, Alex ; ANTONOGLOU, Ioannis ; WIERSTRA, Daan ; RIEDMILLER, Martin A.: Playing Atari with Deep Reinforcement Learning. In: *CoRR* abs/1312.5602 (2013). <http://arxiv.org/abs/1312.5602>
- [MKS⁺15] MINIH, Volodymyr ; KAVUKCUOGLU, Koray ; SILVER, David ; RUSU, Andrei A. ; VENESS, Joel ; BELLEMARE, Marc G. ; GRAVES, Alex ; RIEDMILLER, Martin A. ; FIDJELAND, Andreas ; OSTROVSKI, Georg ; PETERSEN, Stig ; BEATTIE, Charles ; SADIQ, Amir ; ANTONOGLOU, Ioannis ; KING, Helen ; KUMARAN, Dharshan ; WIERSTRA, Daan ; LEGG, Shane ; HASSABIS, Demis: Human-level control through deep reinforcement learning. In: *Nature* 518 (2015), S. 529–533.
- [Moo91] MOORE, Andrew: *Efficient Memory-based Learning for Robot Control*. Pittsburgh, PA, Carnegie Mellon University, Diss., März 1991.
- [MP10] MÜLLER, Michael ; POGUNTKE, Werner: *Basiswissen Statistik*. Herdecker Witten : W3L-Verlag, 2010. – ISBN 978-3-937137-83-4
- [NHTSA13] NATIONAL-HIGHWAY-TRAFFIC-SAFETY-ADMINISTRATION: *Preliminary Statement of Policy Concerning Automated Vehicles*. https://www.nhtsa.gov/staticfiles/rulemaking/pdf/Automated_Vehicles_Policy.pdf, 2013. – Zuletzt abgerufen am 07.03.2019.
- [Ope15] OPENAI: *Introducing OpenAI*. <https://openai.com/blog/introducing-openai/>, Dezember 2015. – Zuletzt abgerufen am 26.03.2019.

LITERATURVERZEICHNIS

- [PAED17] PATHAK, Deepak ; AGRAWAL, Pulkit ; EFROS, Alexei A. ; DARRELL, Trevor: Curiosity-driven Exploration by Self-supervised Prediction. In: *CoRR* abs/1705.05363 (2017). <http://arxiv.org/abs/1705.05363>
- [PCY⁺16] PADEN, Brian ; CÁP, Michal ; YONG, Sze Z. ; YERSHOV, Dmitry S. ; FRAZZOLI, Emilio: A Survey of Motion Planning and Control Techniques for Self-driving Urban Vehicles. In: *CoRR* abs/1604.07446 (2016). <http://arxiv.org/abs/1604.07446>
- [PGC⁺17] PASZKE, Adam ; GROSS, Sam ; CHINTALA, Soumith ; CHANAN, Gregory ; YANG, Edward ; DEVITO, Zachary ; LIN, Zeming ; DESMAISON, Alban ; ANTIGA, Luca ; LERER, Adam: Automatic differentiation in PyTorch. In: *NIPS-W*, 2017.
- [Pom89] POMERLEAU, Dean A.: Advances in Neural Information Processing Systems 1. Version: 1989. <http://dl.acm.org/citation.cfm?id=89851.89891>. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 1989. – ISBN 1–558–60015–9, Kapitel ALVINN: An Autonomous Land Vehicle in a Neural Network, S. 305–313.
- [PTA⁺17] PAULL, L. ; TANI, J. ; AHN, H. ; ALONSO-MORA, J. ; CARLONE, L. ; CAP, M. ; CHEN, Y. F. ; CHOI, C. ; DUSEK, J. ; FANG, Y. ; HOEHENER, D. ; LIU, S. ; NOVITZKY, M. ; OKUYAMA, I. F. ; PAZIS, J. ; ROSMAN, G. ; VARRICCHIO, V. ; WANG, H. ; YERSHOV, D. ; ZHAO, H. ; BENJAMIN, M. ; CARR, C. ; ZUBER, M. ; KARAMAN, S. ; FRAZZOLI, E. ; DEL VECCHIO, D. ; RUS, D. ; HOW, J. ; LEONARD, J. ; CENSI, A.: Duckietown: An open, inexpensive and flexible platform for autonomy education and research. In: *2017 IEEE International Conference on Robotics and Automation (ICRA)*, 2017, S. 1497–1504.
- [Rav18] RAVICHANDIRAN, Sudharsan: *Hands-On - Reinforcement Learning with Python*. Birmingham : Packt Publishing, 2018. – ISBN 978–1–78883–652–4
- [RDGF15] REDMON, Joseph ; DIVVALA, Santosh K. ; GIRSHICK, Ross B. ; FARHADI, Ali: You Only Look Once: Unified, Real-Time Object Detection. In: *CoRR* abs/1506.02640 (2015). <http://arxiv.org/abs/1506.02640>
- [SB17] SUTTON, Richard S. ; BARTO, Andrew G.: *Reinforcement Learning: An Introduction*. 2. Auflage. Cambridge, MA, USA und London, UK : MIT Press, 2017. <https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf>
- [SC18] STANFORD-CYBERLAW: *Summary of Levels of Driving Automation for On-Road Vehicles*. <https://cyberlaw.stanford.edu/files/blogimages/LevelsofDrivingAutomation.pdf>, 2018. – Zuletzt abgerufen am 07.03.2019.
- [Sch17] SCHOETTLE, Brandon: *Sensor Fusion: A Comparison of sensing capabilities of humand drivers and highly automated vehicles*. <http://umich.edu/~umtriswt/PDF/SWT-2017-12.pdf>, August 2017. – Zuletzt abgerufen am 27.03.2019.
- [SHM⁺16] SILVER, David ; HUANG, Aja ; MADDISON, Chris J. ; GUEZ, Arthur ; SIFRE, Laurent ; DRIESSCHE, George van d. ; SCHRITTWIESER, Julian ; ANTONOGLOU, Ioannis ; PANNEERSHELVAM, Veda ; LANCTOT, Marc ; DIELEMAN, Sander ; GREWE, Dominik ; NHAM, John ; KALCHBRENNER, Nal ; SUTSKEVER, Ilya ; LILLICRAP, Timothy ; LEACH, Madeleine ; KAVUKCUOGLU, Koray ; GRAEPEL, Thore ; HASSABIS, Demis: Mastering the Game of Go with Deep Neural Networks and Tree Search. In: *Nature* 529 (2016), Nr. 7587, S. 484–489.

<http://dx.doi.org/10.1038/nature16961>. – DOI 10.1038/nature16961. – ISSN 0028–0836

- [SI18] SAE-INTERNATIONAL ; SAE-INTERNATIONAL (Hrsg.): *Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. J3016-2018-06. Warrendale, USA, 2018.
- [Sil15a] SILVER, David: *Integrating learning and Planning*. University College London, Kurs *Reinforcement Learning*, Vorlesung 8. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/dyna.pdf. Version: Mai 2015.
- [Sil15b] SILVER, David: *Introduction to Reinforcement Learning*. University College London, Kurs *Reinforcement Learning*, Vorlesung 1. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/intro_RL.pdf. Version: Mai 2015.
- [Sil15c] SILVER, David: *Markov Decision Processes*. University College London, Kurs *Reinforcement Learning*, Vorlesung 2. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MDP.pdf. Version: Mai 2015.
- [Sil15d] SILVER, David: *Model-Free Control*. University College London, Kurs *Reinforcement Learning*, Vorlesung 5. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/control.pdf. Version: Mai 2015.
- [Sil15e] SILVER, David: *Model-Free Prediction*. University College London, Kurs *Reinforcement Learning*, Vorlesung 4. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/MC-TD.pdf. Version: Mai 2015.
- [Sil15f] SILVER, David: *Planning by Dynamic Programming*. University College London, Kurs *Reinforcement Learning*, Vorlesung 3. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/DP.pdf. Version: Mai 2015.
- [Sil15g] SILVER, David: *Policy Gradient*. University College London, Kurs *Reinforcement Learning*, Vorlesung 7. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/pg.pdf. Version: Mai 2015.
- [Sil15h] SILVER, David: *Value Function Approximation*. University College London, Kurs *Reinforcement Learning*, Vorlesung 6. http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching_files/FA.pdf. Version: Mai 2015.
- [SLH⁺14] SILVER, David ; LEVER, Guy ; HEESS, Nicolas ; DEGRIS, Thomas ; WIERSTRA, Daan ; RIEDMILLER, Martin: Deterministic Policy Gradient Algorithms. In: *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32*, JMLR.org, 2014 (ICML'14), I-387–I-395.
- [SLM⁺15] SCHULMAN, John ; LEVINE, Sergey ; MORITZ, Philipp ; JORDAN, Michael I. ; ABBEEL, Pieter: Trust Region Policy Optimization. In: *arXiv e-prints* (2015), Februar, S. arXiv:1502.05477.
- [SQAS15] SCHAUL, Tom ; QUAN, John ; ANTONOGLOU, Ioannis ; SILVER, David: Prioritized Experience Replay. In: *arXiv e-prints* (2015), S. arXiv:1511.05952.
- [SSSI11] STALLKAMP, Johannes ; SCHLIPSING, Marc ; SALMEN, Jan ; IGEL, Christian: The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In: *Proceedings of the International Joint Conference on Neural Networks*, 2011, S. 1453 – 1460.

LITERATURVERZEICHNIS

- [STK18] SHARMA, S. ; TEWOLDE, G. ; KWON, J.: Behavioral Cloning for Lateral Motion Control of Autonomous Vehicles Using Deep Learning. In: *2018 IEEE International Conference on Electro/Information Technology (EIT)*, 2018. – ISSN 2154–0373, S. 0228–0233.
- [SVZ13] SIMONYAN, Karen ; VEDALDI, Andrea ; ZISSERMAN, Andrew: Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps. In: *arXiv e-prints* (2013), Dezember, S. arXiv:1312.6034.
- [SWD⁺17] SCHULMAN, John ; WOLSKI, Filip ; DHARIWAL, Prafulla ; RADFORD, Alec ; KLIMOV, Oleg: Proximal Policy Optimization Algorithms. In: *arXiv e-prints* (2017), Juli, S. arXiv:1707.06347.
- [TMD⁺06] THRUN, Sebastian ; MONTEMERLO, Mike ; DAHLKAMP, Hendrik ; STAVENS, David ; ARON, Andrei ; DIEBEL, James ; FONG, Philip ; GALE, John ; HALPENNY, Morgan ; HOFFMANN, Gabriel ; LAU, Kenny ; OAKLEY, Celia ; PALATUCCI, Mark ; PRATT, Vaughan ; STANG, Pascal ; STROHBAND, Sven ; DUPONT, Cedric ; JENDROSSEK, Lars-Erik ; KOELEN, Christian ; MARKEY, Charles ; RUMMEL, Carlo ; NIEKERK, Joe van ; JENSEN, Eric ; ALESSANDRINI, Philippe ; BRADSKI, Gary ; DAVIES, Bob ; ETTINGER, Scott ; KAEHLER, Adrian ; NEFIAN, Ara ; MAHONEY, Pamela: Stanley: The Robot That Won the DARPA Grand Challenge: Research Articles. In: *J. Robot. Syst.* 23 (2006), September, Nr. 9, S. 661–692. <http://dx.doi.org/10.1002/rob.v23:9>. – DOI 10.1002/rob.v23:9. – ISSN 0741–2223
- [TSK⁺19] TALPAERT, Victor ; SOBH, Ibrahim ; KIRAN, B R. ; MANNION, Patrick ; YOGAMANI, Senthil ; EL-SALLAB, Ahmad ; PEREZ, Patrick: Exploring applications of deep reinforcement learning for real-world autonomous driving systems. In: *arXiv e-prints* (2019), Januar, S. arXiv:1901.01536.
- [vGS15] VAN HASSELT, Hado ; GUEZ, Arthur ; SILVER, David: Deep Reinforcement Learning with Double Q-learning. In: *arXiv e-prints* (2015), S. arXiv:1509.06461.
- [VN16] VITELLI, Matt ; NAYEBI, Aran: CARMA: A Deep Reinforcement Learning Approach to Autonomous Driving. (2016). https://web.stanford.edu/~anayebi/projects/CS_239_Final_Project_Writeup.pdf
- [VYKL14] VASQUEZ, D. ; YU, Y. ; KUMAR, S. ; LAUGIER, C.: An Open Framework for Human-Like Autonomous Driving Using Inverse Reinforcement Learning. In: *2014 IEEE Vehicle Power and Propulsion Conference (VPPC)*, 2014. – ISSN 1938–8756, S. 1–4.
- [WD92] WATKINS, Christopher J. C. H. ; DAYAN, Peter: Q-learning. In: *Machine Learning* 8 (1992), Mai, Nr. 3, 279–292. <http://dx.doi.org/10.1007/BF00992698>. – DOI 10.1007/BF00992698. – ISSN 1573–0565
- [WEG⁺15] WYMANN, Bernhard ; ESPIÉ, Eric ; GUIONNEAU, Christophe ; DIMITRAKAKIS, Christos ; COULOM, Rémi ; SUMNER, Andrew: TORCS: The open racing car simulator. <http://www.cse.chalmers.se/~chrdimi/papers/torcs.pdf>, März 2015.
- [WEG⁺19a] WYMANN, Bernhard ; ESPIÉ, Eric ; GUIONNEAU, Christophe ; DIMITRAKAKIS, Christos ; COULOM, Rémi ; SUMNER, Andrew: The TORCS Racing Board - Cars. http://www.berniw.org/trb/cars/car_view.php?viewcarid=5, 2019. – Zuletzt abgerufen am 23.03.2019.

- [WEG⁺19b] WYMANN, Bernhard ; ESPIÉ, Eric ; GUIONNEAU, Christophe ; DIMITRAKAKIS, Christos ; COULOM, Rémi ; SUMNER, Andrew: *The TORCS Racing Board - Track List.* <http://www.berniw.org/trb/tracks/tracklist.php>, 2019. – Zuletzt abgerufen am 22.03.2019.
- [Wen18] WENG, Lilian: *Webartikel: Policy Gradient Algorithms.* <https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html>, April 2018. – Zuletzt abgerufen am 14.04.2019.
- [WHW⁺17] WOLF, P. ; HUBSCHNEIDER, C. ; WEBER, M. ; BAUER, A. ; HÄRTL, J. ; DÜRR, F. ; ZÖLLNER, J. M.: Learning how to drive in a real world simulation with deep Q-Networks. In: *2017 IEEE Intelligent Vehicles Symposium (IV)*, 2017, S. 244–250.
- [Wil92] WILLIAMS, Ronald J.: Simple statistical gradient-following algorithms for connectionist reinforcement learning. In: *Machine Learning* 8 (1992), Mai, Nr. 3, S. 229–256. <http://dx.doi.org/10.1007/BF00992696>. – DOI 10.1007/BF00992696. – ISSN 1573–0565
- [WJW18] WANG, Sen ; JIA, Daoyuan ; WENG, Xinshuo: Deep Reinforcement Learning for Autonomous Driving. In: *CoRR* abs/1811.11329 (2018). <http://arxiv.org/abs/1811.11329>
- [WSH⁺15] WANG, Ziyu ; SCHAUL, Tom ; HESSEL, Matteo ; VAN HASSELT, Hado ; LANTOT, Marc ; DE FREITAS, Nando: Dueling Network Architectures for Deep Reinforcement Learning. In: *arXiv e-prints* (2015), S. arXiv:1511.06581.
- [Yi18] YI, Hongsuk: Deep Deterministic Policy Gradient for Autonomous Vehicle Driving. In: *Int. Conf. Artificial Intelligence (ICAI) 2018*, CSREA Press, 2018. – ISBN 1–60132–480–4, S. 191–194.
- [Yos17] YOSHIDA, Naoto: *Gym-TORCS.* <https://github.com/ugo-nama-kun/gym-torcs>, 2017.
- [ZHBH17] ZEILINGER, Marcel ; HAUK, Raphael ; BADER, Markus ; HOFMANN, Alexander: Design of an Autonomous Race Car for the Formula Student Driverless (FSD). In: *Proceedings of the OAGM and ARW*, 2017.
- [ZSSH19] ZHAO, Chenyang ; SIGAUD, Olivier ; STULP, Freek ; HOSPEDALES, Timothy M.: Investigating Generalisation in Continuous Deep Reinforcement Learning. In: *CoRR* abs/1902.07015 (2019). <http://arxiv.org/abs/1902.07015>
- [ZVMB18] ZHANG, Chiyuan ; VINYALS, Oriol ; MUNOS, Rémi ; BENGIO, Samy: A Study on Overfitting in Deep Reinforcement Learning. In: *CoRR* abs/1804.06893 (2018). <http://arxiv.org/abs/1804.06893>

A

Taxonomien zu autonomen Fahrzeugen

Level	Name	Narrative definition	Execution of steering and acceleration/deceleration	Monitoring of driving environment	Fallback performance of dynamic driving task	System capability (driving modes)	BASL Level	NHTSA Level
Human driver monitors the driving environment								
0 Automation	No Automation	the full-time performance by the <i>human driver</i> of all aspects of the <i>dynamic driving task</i> , even when enhanced by warning or intervention systems	Human driver	Human driver	Human driver	n/a	0	
1 Driver Assistance	Driver Assistance	the <i>driving mode-specific execution</i> by a driver assistance system of either steering or acceleration/deceleration using information about the driving environment and with the expectation that the <i>human driver</i> perform all remaining aspects of the <i>dynamic driving task</i>	Human driver and system	Human driver	Human driver	Some driving modes	1	
2 Partial Automation	Partial Automation	the <i>driving mode-specific execution</i> by one or more driver assistance systems of both steering and acceleration/deceleration using information about the driving environment and with the expectation that the <i>human driver</i> perform all remaining aspects of the <i>dynamic driving task</i>	System	Human driver	Human driver	Some driving modes	2	
Automated driving system ("system") monitors the driving environment								
3 Conditional Automation	Conditional Automation	the <i>driving mode-specific performance</i> by an <i>automated driving system</i> of all aspects of the <i>dynamic driving task</i> with the expectation that the <i>human driver</i> will respond appropriately to a <i>request to intervene</i>	System	System	Human driver	Some driving modes	3	
4 High Automation	High Automation	the <i>driving mode-specific performance</i> by an <i>automated driving system</i> of all aspects of the <i>dynamic driving task</i> , even if a <i>human driver</i> does not respond appropriately to a <i>request to intervene</i>	System	System	System	Some driving modes	4	
5 Full Automation	Full Automation	the <i>full-time performance</i> by an <i>automated driving system</i> of all aspects of the <i>dynamic driving task</i> under all roadway and environmental conditions that can be managed by a <i>human driver</i>	System	System	System	All driving modes	-	

Abbildung A.1: Dargestellt ist ein Vergleich der SAE-Taxonomie für autonome Fahrzeuge mit den Taxonomien der Bundesanstalt für Straßenwesen (BASf) und der National Highway Traffic Safety Administration (NHTSA) der USA. Bildquelle: [SC18].

B Literaturvergleich

Publikation	Verwendete Algorithmen	Auswertungsmetriken		Hypothesen/Vergleiche	Enthält AD?
Kendall et al. (2018)	DDPG mit CNN, DDPG mit VAE	Maximal erreichte Distanz über alle Episoden, Dauer des Lernens	Vergleich DDPG mit CNN und DDPG mit VAE		Ja
Wang et al. (2018)	DDPG	Mittlere Geschwindigkeit pro Episode, Ertrag pro Episode, gefahrene Distanz pro Episode, Mittlere Abweichung von Fahrbahnmitte pro Episode	Training im Practice Mode von TORCS und Testing im Competition Mode von TORCS. Lediglich Darstellung der Trainingsperformance pro Episode		Ja
Jaritz et al. (2018)	A3C	Distanz pro Anpassungsschritt, durchschnittliche Geschwindigkeit pro Anpassungsschritt, Anzahl der Unfälle pro Anpassungsschritt	Vergleich der Performance der Convolutional-Schicht, Reward-Funktion und Respawn-Strategie eines A3C-Agenten mit der Performance des A3C-Agenten aus Mnih et al. (2016)		Ja
Henderson et al. (2017)	DDPG, PPO, TRPO, ACKTR	Durchschnittlicher Ertrag pro Zeitschritt	Vergleich genannter Algorithmen innerhalb mehreren OpenAI-Gym-Umgebungen		Nein
Vitelli und Nayebi (2016)	CNN-RNN DQN, Q-Learning	Durchschnittlicher Ertrag über alle Episoden, Durchschnittliche Geschwindigkeit über alle Episoden, Maximale Geschwindigkeit über alle Episoden	Vergleich der genannten Algorithmen hinsichtlich der Auswertungskriterien		Ja
El Sallab et al. (2016)	DQN, DDAC	Anzahl der Rennrunden, Effekt von Diskretisierung auf Fahrverhalten	Vergleich DQN mit DDAC, Convergence Time vs. Termination Conditions		Ja
Mnih et al. (2016)	DQN, DDQN, Duel-DQN, Prioritized-DQN, A3C, n-Step Q-Learning	Spiel-Score der Atari2600-Spiele über die Trainingszeit, Vergleich mit menschlichen Spiel-Scores in TORCS. Mit und ohne Gegner.	Vergleich von Async 1-step Q-Learning, Async SARSA, Async n-step Q-Learning, A3C und Mensch in TORCS. Mit und ohne Gegner.		Ja
Mnih et al. (2013)	Random, SARSA, DQN	Durchschnittliche Belohnung pro Episode, Durchschnittliche Action-Value pro Episode	Vergleich von SARSA, DQN, menschlichem Probanden hinsichtlich der Auswertungsmetriken		Nein

Abbildung B.1: Dargestellt ist eine Übersicht über die Auswertungsmetriken und Hypothesen der für diese Arbeit bedeutenden und populären Publikationen. Aus dieser eigens erstellten Erhebung werden in Kapitel 4 die Performanzmetriken des Experimentes dieser Arbeit mitunter abgeleitet. Die letzte Spalte der Tabelle gibt an, ob in der Publikation das Thema des autonomen Fahren untersucht wurde, da dieser Umstand bei der Auswahl der Auswertungsmetriken mitberücksichtigt werden muss. Bildquellen: Eigens erstellt.

C Aufbau der neuronalen Netze

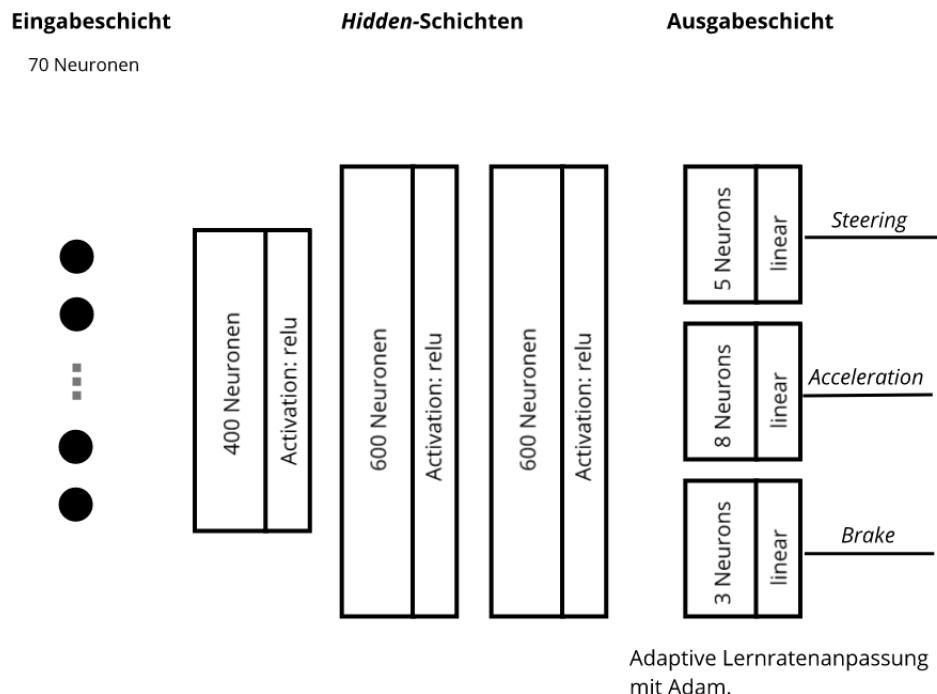


Abbildung C.1: Dargestellt ist der Aufbau des neuronalen Netzes des DQN-Agenten. Die Verbindungen zwischen den Neuronen sind, aus Gründen der Übersichtlichkeit, nicht dargestellt. Das neuronale Netz des DQN-Agenten ist ein *Feedforward*-Netz was bedeutet, dass jedes Neuron einer Schicht mit jedem Neuron der nachfolgenden Schicht verbunden ist. Das neuronale Netz besteht aus insgesamt $400 + 600 + 600 + 16 = 1.616$ Neuronen mit insgesamt 637.600 Gewichten, zählt man die *Bias* nicht hinzu. Für die verdeckten Schichten wird die Aktivierungsfunktion $\text{relu}(x) = \max(0, x) \in [0; \infty)$ genutzt, während die Ausgabeneuronen linear aktiviert werden. Bildquelle: Eigens erstellt.

Hyperparameter	Erklärung	Wert
Startwert von ε	Faktor der bestimmt, ob der Agent die Umgebung erkunden soll oder die <i>Policy</i> die Aktion bestimmt.	1,0
Verfallrate von ε	Pro Zeitschritt wird $\varepsilon \leftarrow \varepsilon - \text{Verfallrate}$ neu berechnet.	0,00015
Mindestwert von ε	Grenzwert nachdem ε nicht weiter verfällt.	0,09
Diskontierungsfaktor γ	Faktor der den Ertrag des nächsten Zustandes abschwächt, z.B. nach Gleichung 3.33.	0,99
Lernrate α	Faktor für die Gewichtung des Gradienten im <i>gradient descent</i> -Verfahren.	0,002
Größe <i>Replay Memory</i>	Anzahl der im <i>Replay Memory</i> gespeicherten Tupel (s, a, r, s') .	50.000
Batch Size	Größe der Stichprobe aus dem <i>Replay Memory</i> für die Gewichtsanpassung.	128

Tabelle C.1: Tabellarische Übersicht über die Hyperparametereinstellungen des DQN-Agenten-Algorithmus. Von besonderer Wichtigkeit ist die Verfallrate von ε . Diese wurde durch Experimente auf den in der Tabelle angegeben Wert festgelegt. Ist die Verfallrate zu groß, hört der Agent zu früh auf die Umgebung zu erkunden, ist die Verfallrate zu klein, kann es sehr lange dauern, bis der Agent Performanzfortschritte macht.

C Aufbau der neuronalen Netze

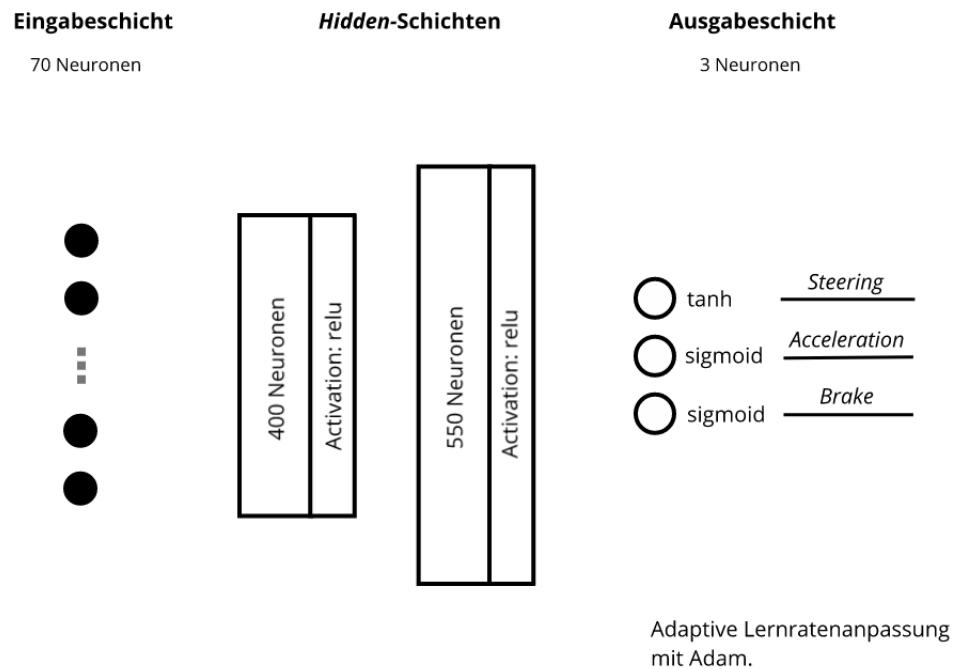


Abbildung C.2: Dargestellt ist der Aufbau des neuronalen Netzes des DDPG-*Actor*. Die Verbindungen zwischen den Neuronen sind, aus Gründen der Übersichtlichkeit, nicht dargestellt. Das neuronale Netz des DDPG-*Actor* ist ein *Feedforward*-Netz was bedeutet, dass jedes Neuron einer Schicht mit jedem Neuron der nachfolgenden Schicht verbunden ist. Das neuronale Netz besteht aus insgesamt $400 + 550 + 3 = 953$ Neuronen mit $70 \times 400 + 400 \times 550 + 550 \times 3 = 249.500$ trainierbaren Gewichten, zählt man die *Bias* nicht hinzu. Die Neuronen den verdeckten Schichten haben die Aktivierungsfunktion $\text{relu}(x) = \max(0, x) \in [0; \infty)$, während das Ausgabeneuron für die Querführung $\tanh(x) = (e^x + e^{-x}) / (e^x - e^{-x}) \in (-1; 1)$ und das Ausgabeneuron für die Längsführung $\sigma(x) = (1 + e^{-x})^{-1} \in (0; 1)$ als Aktivierungsfunktionen nutzen. Die 70 Eingabeneuronen entsprechen jeweils einem Sensorwert der in Tabelle 4.1 aufgeführten Sensoren. Bildquelle: Eigens erstellt.

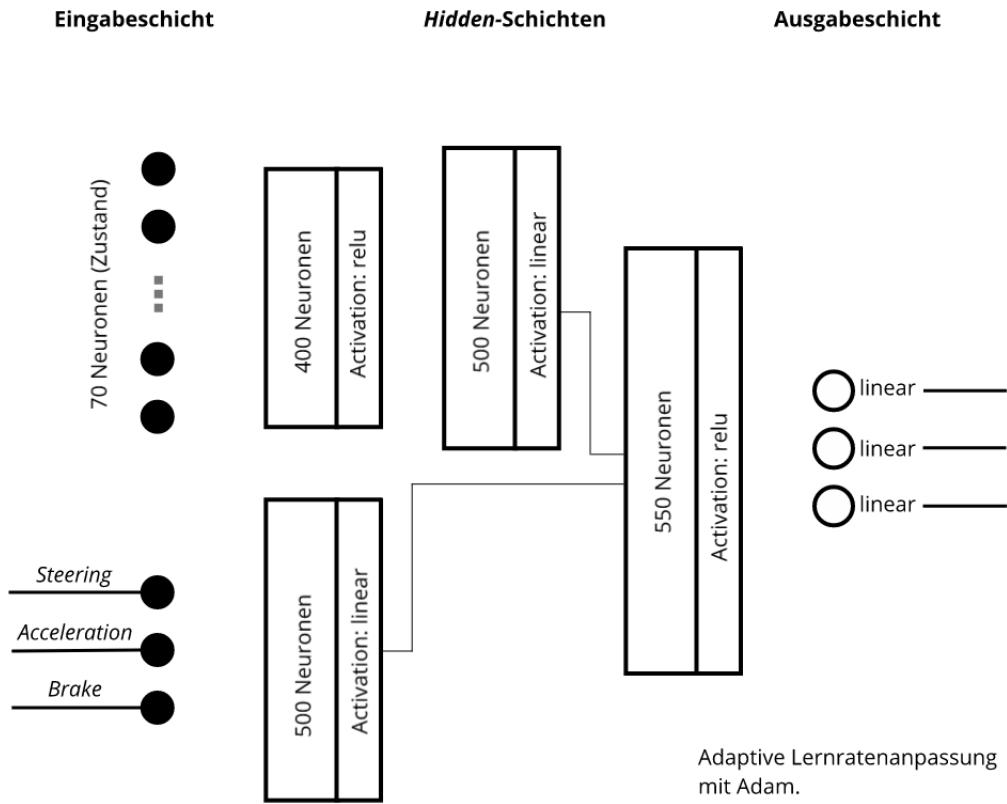


Abbildung C.3: Dargestellt ist der Aufbau des neuronalen Netzes des DDPG-*Critic*. So wie das DDPG-*Actor*-Netz, ist auch das neuronale Netz des *Critic* ein *Feedforward*-Netz. Anders als die bisherigen neuronalen Netze, nimmt das Netz des DDPG-*Critic* außer den Sensordaten noch die Ausgabe des DDPG-*Actors* entgegen. Diese beiden Eingaben werden in einer der verdeckten Schichten zusammengeführt. Ausgabe des DDPG-*Critic*-Netzes sind die aus den DDPG-*Actor*-Werten und Sensorwerten resultierenden Erträge. Diese können, nach der in Algorithmus 4 dargestellten Berechnungsweise, für die Bestimmung des Fehlers in *Actor*- und *Critic*-Netz genutzt werden. Der Backpropagation-Algorithmus ändert dann die Gewichte der Netze gemäß ihres Anteiles am Fehler. Wichtig bleibt noch festzuhalten, dass, da der DDPG-Algorithmus *off-policy* ist, *Actor*- und *Critic*-Netz für sowohl die *Policy*, die tatsächlich die Aktionen bestimmt, als auch für die *Policy*, die das TD-Target berechnet, jeweils einzeln existieren. Das neuronale Netz hat insgesamt 1.953 Neuronen und 782.800 Gewichte, zählt man die *Bias* nicht hinzu. Bildquelle: Eigens erstellt.

Hyperparameter	Erklärung	Wert
Startwert von ε	Faktor der bestimmt, ob der Agent die Umgebung erkunden soll oder die <i>Policy</i> die Aktion bestimmt.	1,0
Verfallrate von ε	Pro Zeitschritt wird $\varepsilon \leftarrow \varepsilon - \text{Verfallrate}$ neu berechnet.	0,00001
Mindestwert von ε	Grenzwert nachdem ε nicht weiter verfällt.	0,07
Diskontierungsfaktor γ	Faktor der den Ertrag des nächsten Zustandes abschwächt, z.B. nach Gleichung 3.33.	0,99
Lernrate <i>Actor</i>	Faktor für die Gewichtung des Gradien-ten im <i>gradient descent</i> -Verfahren beim <i>Actor</i> .	0,00011
Lernrate <i>Critic</i>	Faktor für die Gewichtung des Gradien-ten im <i>gradient descent</i> -Verfahren beim <i>Critic</i> .	0,0011
Größe <i>Replay Memory</i>	Anzahl der im <i>Replay Memory</i> gespei-cherten Tupel (s, a, r, s') .	50.000
Batch Size	Größe der Stichprobe aus dem <i>Replay Memory</i> für die Gewichtsanpassung.	64
v	Skalierungsfaktor für Target-Netz-Anpassung.	0.001
OU Lenkung θ	Steifigkeit des OU-Prozesses für die Lenkung.	0,55
OU Lenkung μ	Gleichgewichtsniveau des OU-Prozesses für die Lenkung.	0,0
OU Lenkung σ	Diffusion des OU-Prozesses für die Lenkung.	0,15
OU Beschleunigung θ	Steifigkeit des OU-Prozesses für die Be-schleunigung.	1,0
OU Beschleunigung μ	Gleichgewichtsniveau des OU-Prozesses für die Beschleunigung.	0,55
OU Beschleunigung σ	Diffusion des OU-Prozesses für die Be-schleunigung.	0,1
OU Bremse θ	Steifigkeit des OU-Prozesses für die Bremse.	1,0
OU Bremse μ	Gleichgewichtsniveau des OU-Prozesses für die Bremse.	0.25
OU Bremse σ	Diffusion des OU-Prozesses für die Bremse.	0.1

Tabelle C.2: Tabellarische Übersicht über die Hyperparametereinstellungen des DDPG-Agenten-Algorithmus. Ebenso wie beim DQN-Agenten, zeigen Experimente, dass die Verfallrate von ε großen Einfluss auf die Dauer des Lernprozesses hat. Auch die Einstellungen der OU-Parameter sind entscheidend für eine schnelle Konvergenz.

D Darstellung des Bremsvorganges

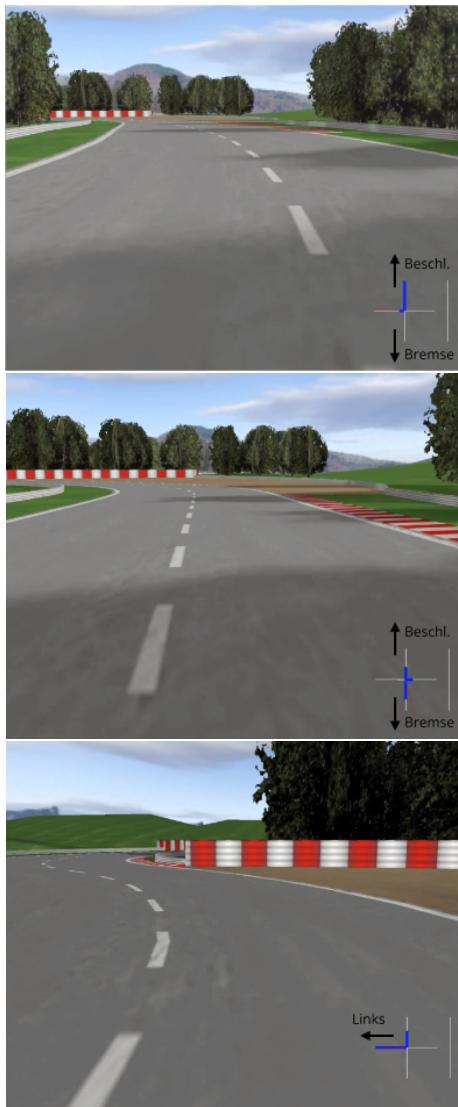


Abbildung D.1: Dargestellt ist der Bremsvorgang des DDPG-Agenten auf der Strecke E-Road. Im obersten Bild ist zu sehen, wie das Fahrzeug auf eine Kurve bei voller Beschleunigung zufährt. Im mittleren Bild sieht man dann, wie der Agent die Bremse betätigt, um die Geschwindigkeit zu reduzieren. Bemerkenswert ist dabei, dass der Agent auch gleichzeitig beschleunigt. Im untersten Bild wird gezeigt, dass eine starke Lenkbewegung entsprechend der Streckenrichtung vom Agenten durchgeführt wird. Bildquellen: Eigens erstellt.

E Testergebnisse DDPG-Agent im *Practice*-Modus

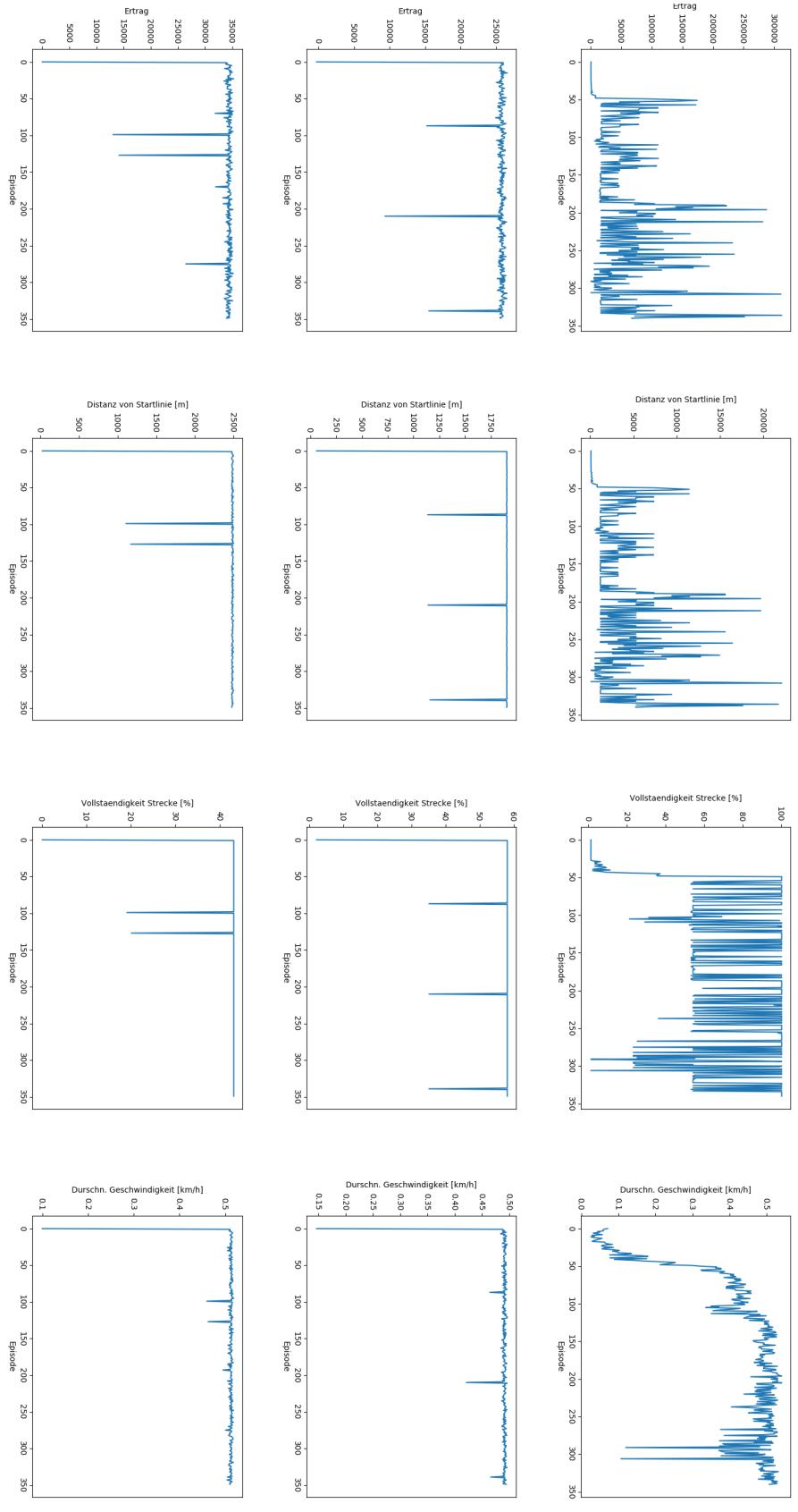


Abbildung E.1: Dargestellt sind die Performanzdaten des DDPG-Agenten während des Testens. Der Agent wurde auf der Strecke CG Speedway angelernt. Die oberste Reihe zeigt zur Referenz die Trainingsergebnisse auf CG Speedway und ist äquivalent zu Abbildung 5.1. Die mittlere Reihe zeigt die Testergebnisse auf der Strecke E-Road. Die unterste Reihe zeigt die Testergebnisse auf der Strecke Forza. Von links nach rechts: Ertrag pro Episode, erreichte Distanz des Agenten von der Startlinie pro Episode, Vollständigkeit der Rennstrecke pro Episode und durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.

E Testergebnisse DDPG-Agent im Practice-Modus

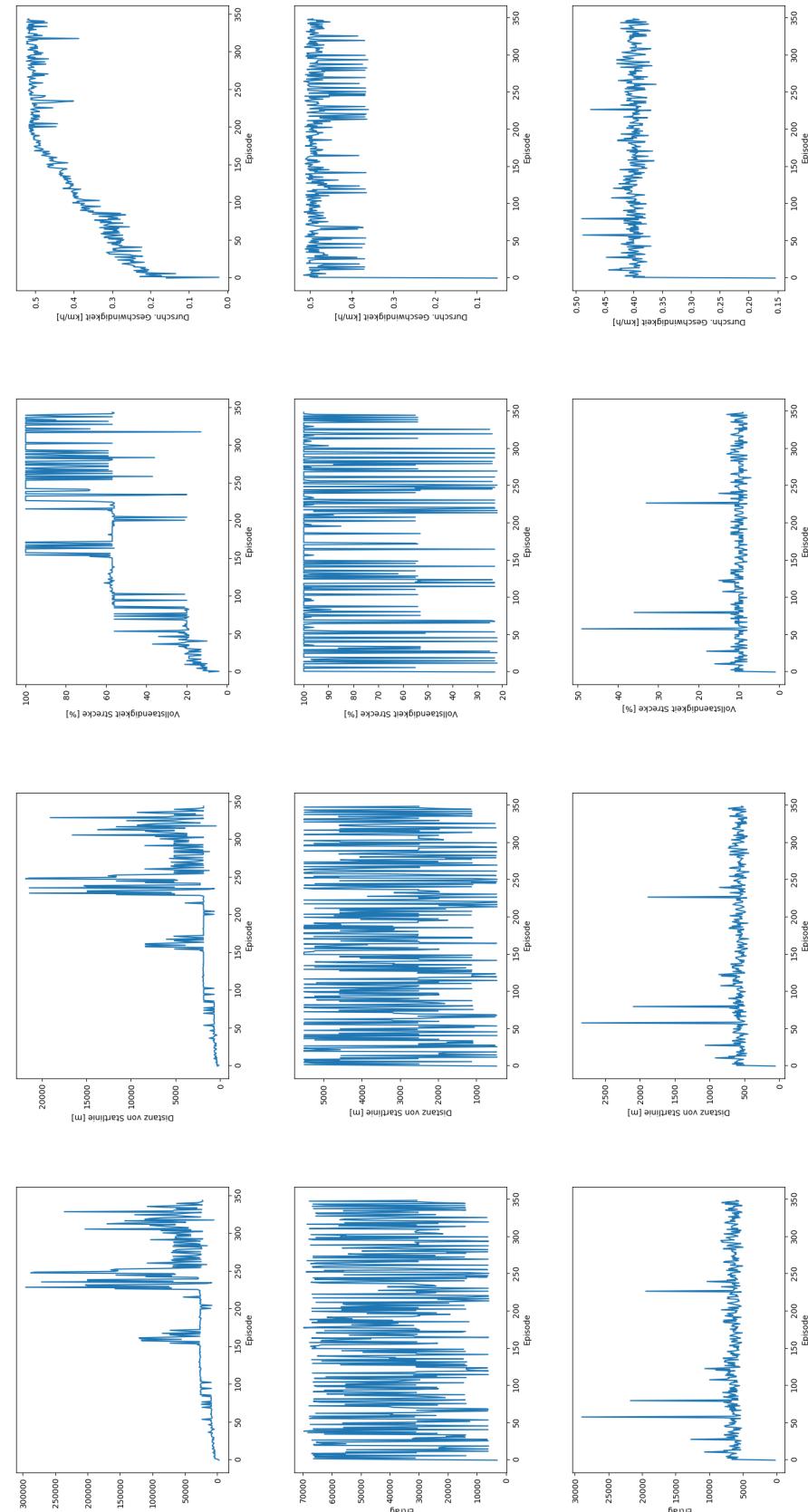


Abbildung E.2: Dargestellt sind die Performanzdaten des DDPG-Agenten während des Testens. Der Agent wurde auf der Strecke E-Road angelernt. Die oberste Reihe zeigt zur Referenz die Trainingsergebnisse auf E-Road und ist äquivalent zu Abbildung 5.2. Die mittlere Reihe zeigt die Testergebnisse auf der Strecke CG Speedway. Die unterste Reihe zeigt die Testergebnisse auf der Strecke Forza. Von links nach rechts: Ertrag pro Episode, Vollständigkeit der Rennstrecke pro Episode und durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.

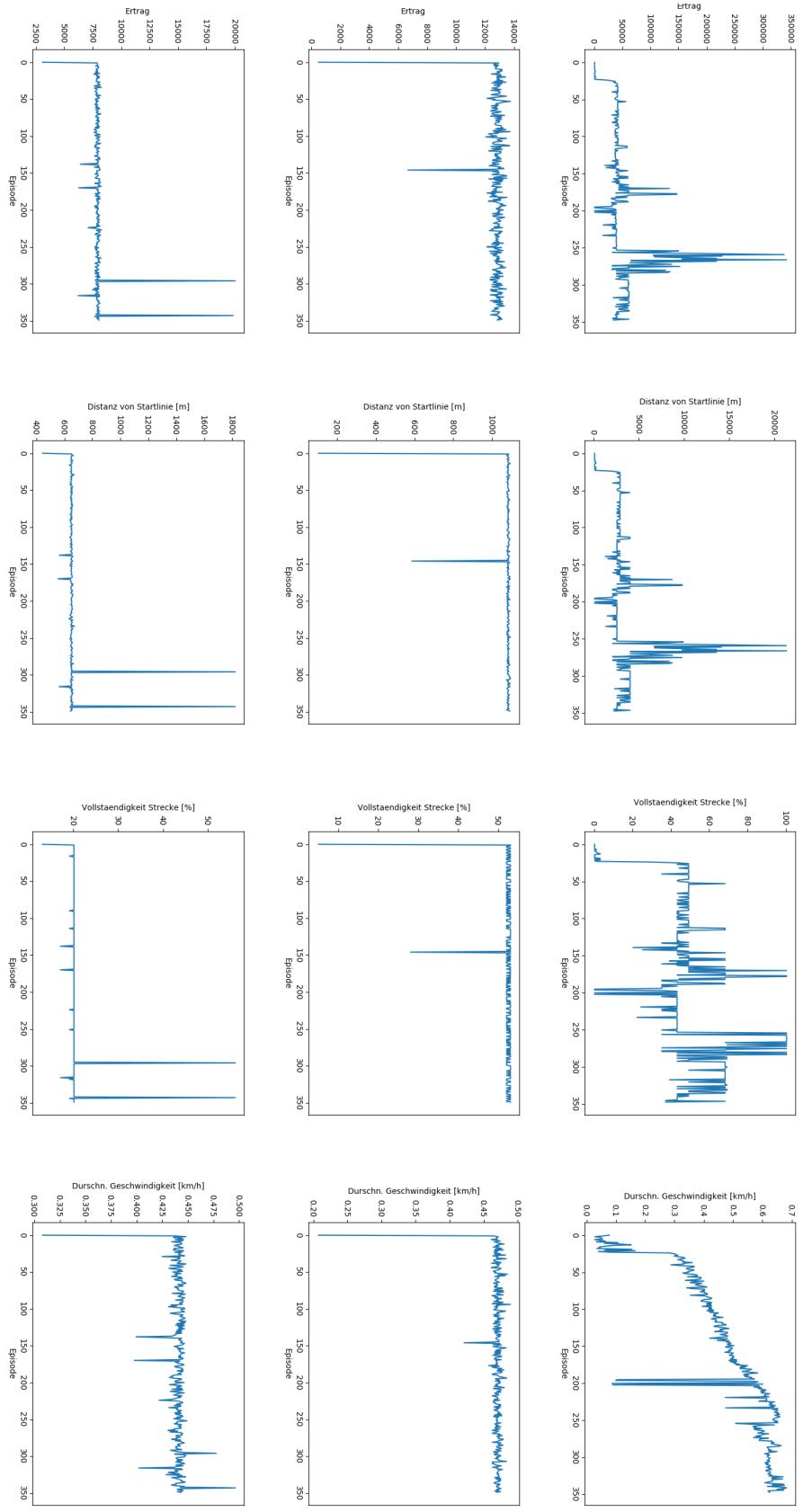


Abbildung E.3: Dargestellt sind die Performanzdaten des DDPG-Agenten während des Testens. Der Agent wurde auf der Strecke Forza angelernt. Die oberste Reihe zeigt zur Referenz die Trainingsergebnisse auf der Strecke Forza und ist equivalent zu Abbildung 5.3. Die mittlere Reihe zeigt die Testergebnisse auf der Strecke CG Speedway. Die unterste Reihe zeigt die Testergebnisse auf der Strecke E-Road. Von links nach rechts: Ertrag pro Episode, erreichte Distanz des Agenten von der Startlinie pro Episode und durchschnittliche Geschwindigkeit pro Episode.

F Darstellung von Überholmanövern

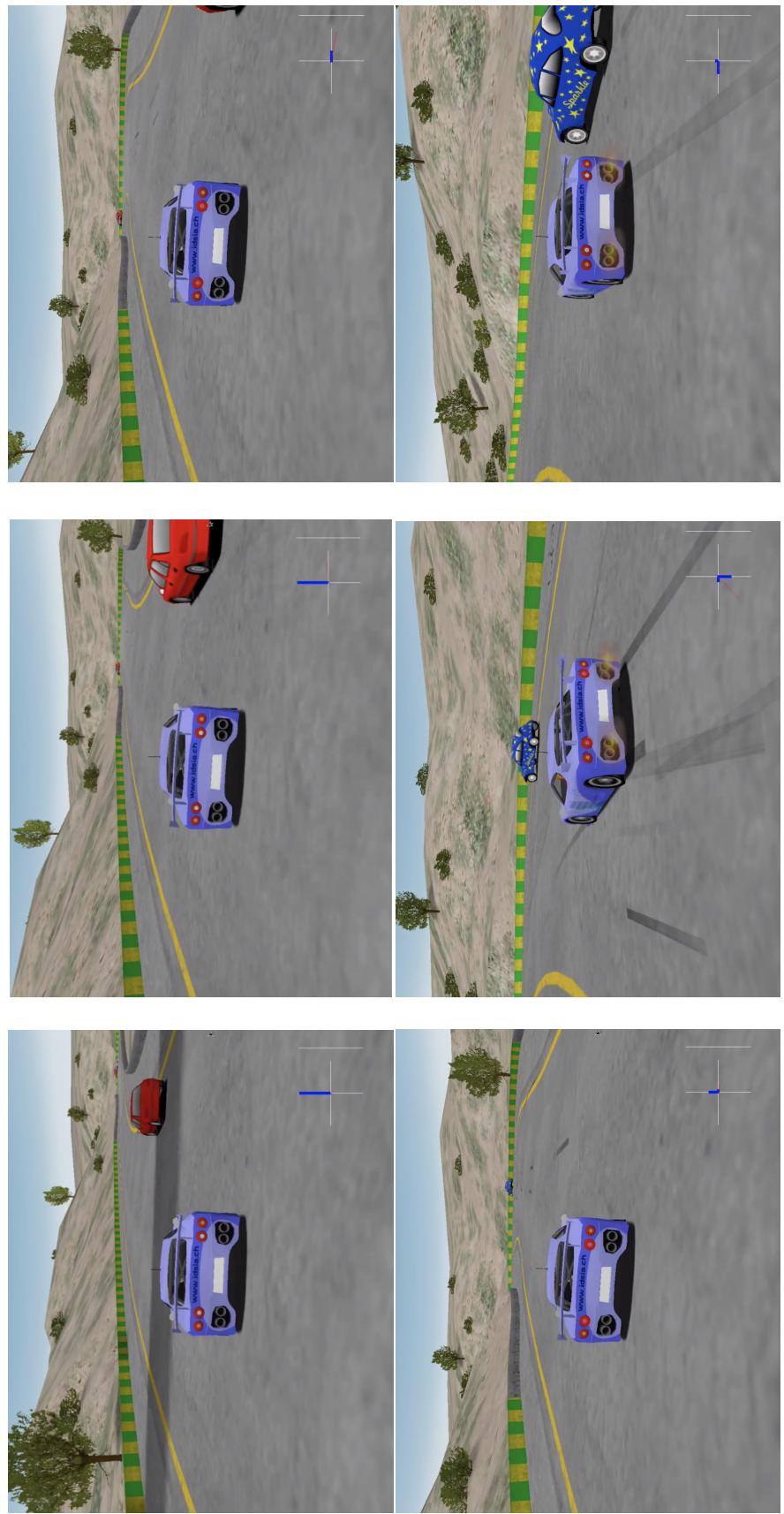


Abbildung F.1: Dargestellt sind zwei exemplarische Überholmanöver des vom DDPG-Agenten gesteuerten Fahrzeugbot auf der Strecke CG Speedway. In den oberen drei Abbildungen ist zu sehen, wie der Agent ein rotes Fahrzeug auf beinahe gerader Strecke ohne Kollision überholt. In den unteren drei Abbildungen ist zu sehen, wie der Agent ein blaues Fahrzeug der Konkurrenz ohne Kollision in einer Kurve überholt. Bildquellen: Eigens erstellt.

G Trainingsergebnisse des DDPG-Agenten im *Quick Race*-Modus

G Trainingsergebnisse des DDPG-Agenten im Quick Race-Modus

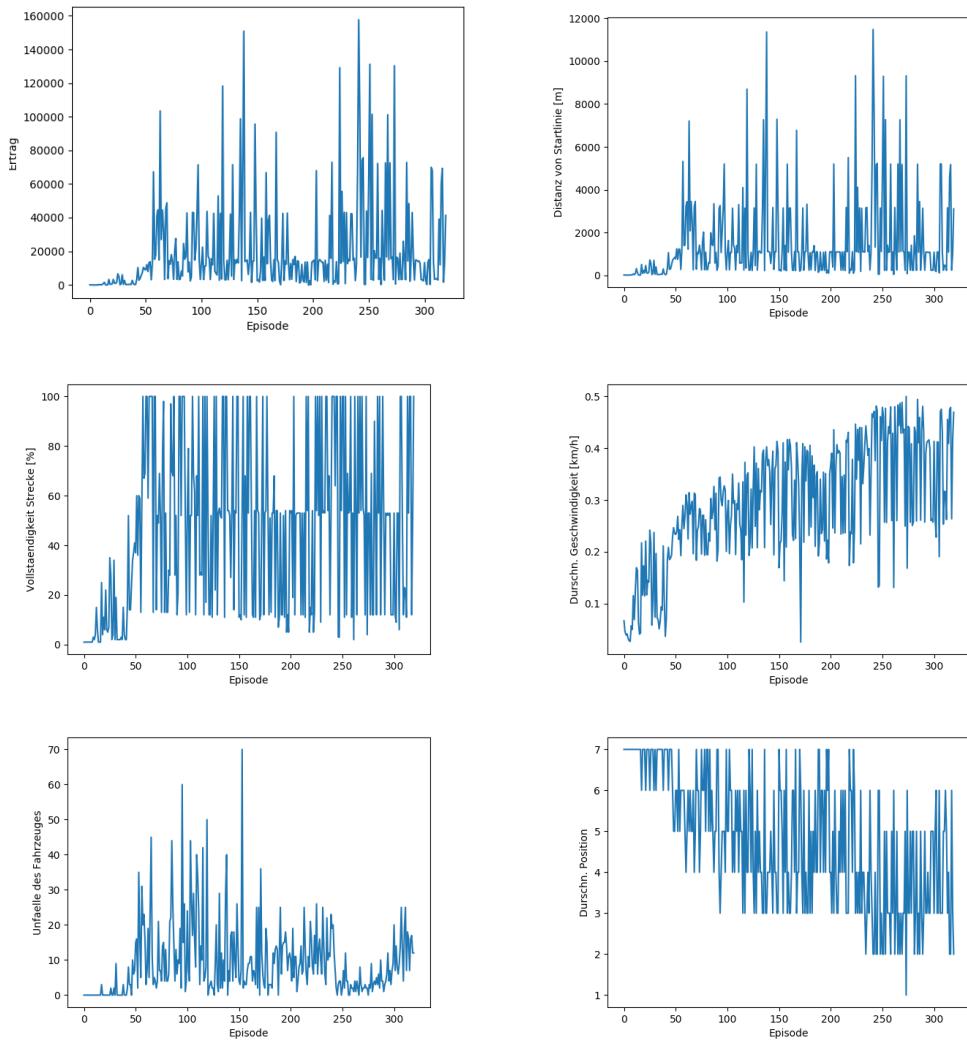


Abbildung G.1: Dargestellt sind die Performanzdaten des DDPG-Agenten während des Trainings im *Quick Race*-Modus auf der Strecke CG Speedway. Von l.o. nach r.u.: (a) Ertrag pro Episode, (b) erreichte Distanz des Agenten von der Startlinie pro Episode, (c) Vollständigkeit der Rennstrecke pro Episode, (d) durchschnittliche Geschwindigkeit pro Episode, (e) Kollisionen pro Episode und (f) durchschnittliche Position pro Episode. Bildquellen: Eigens erstellt.

H Vergleich der Belohnungsfunktionen

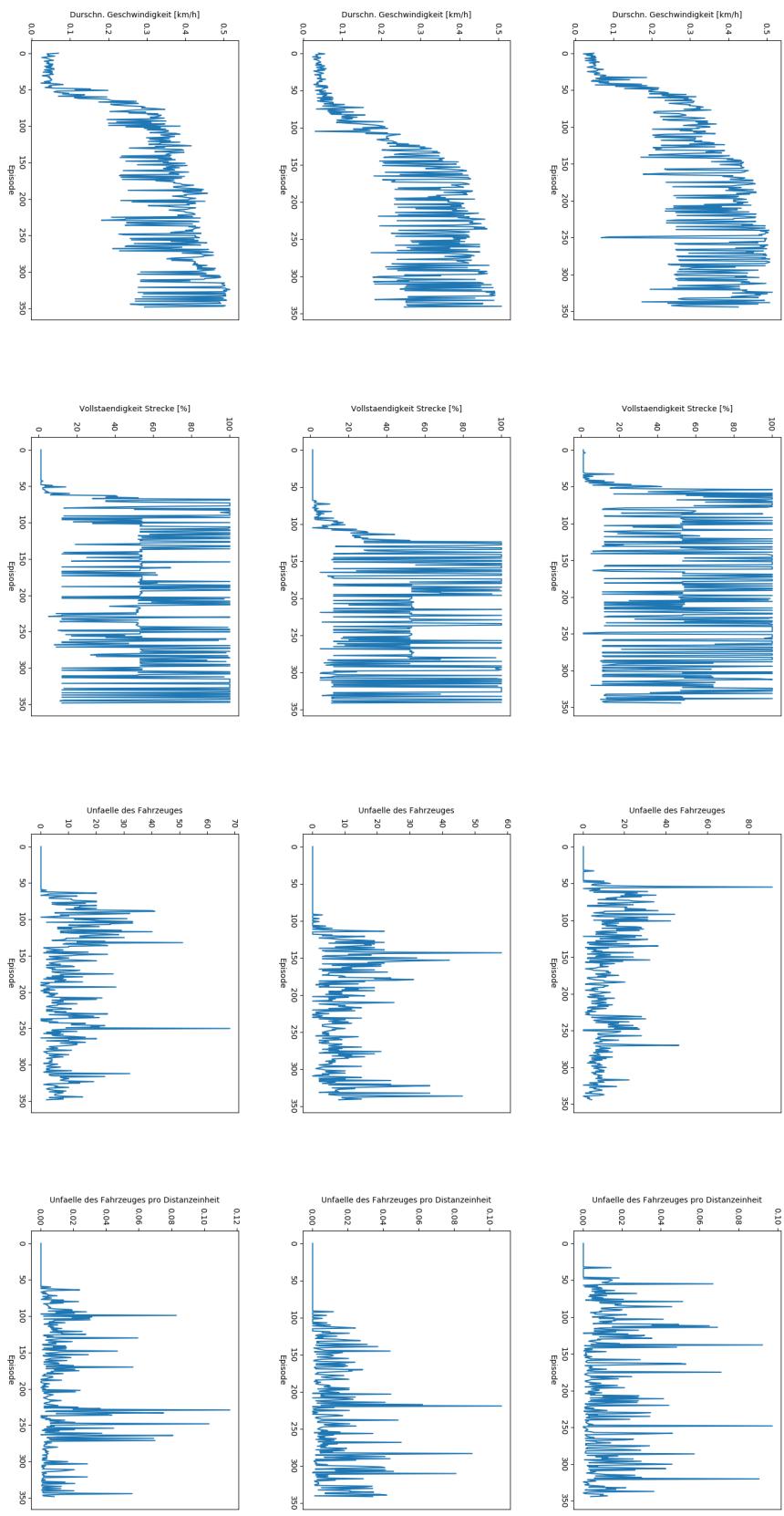
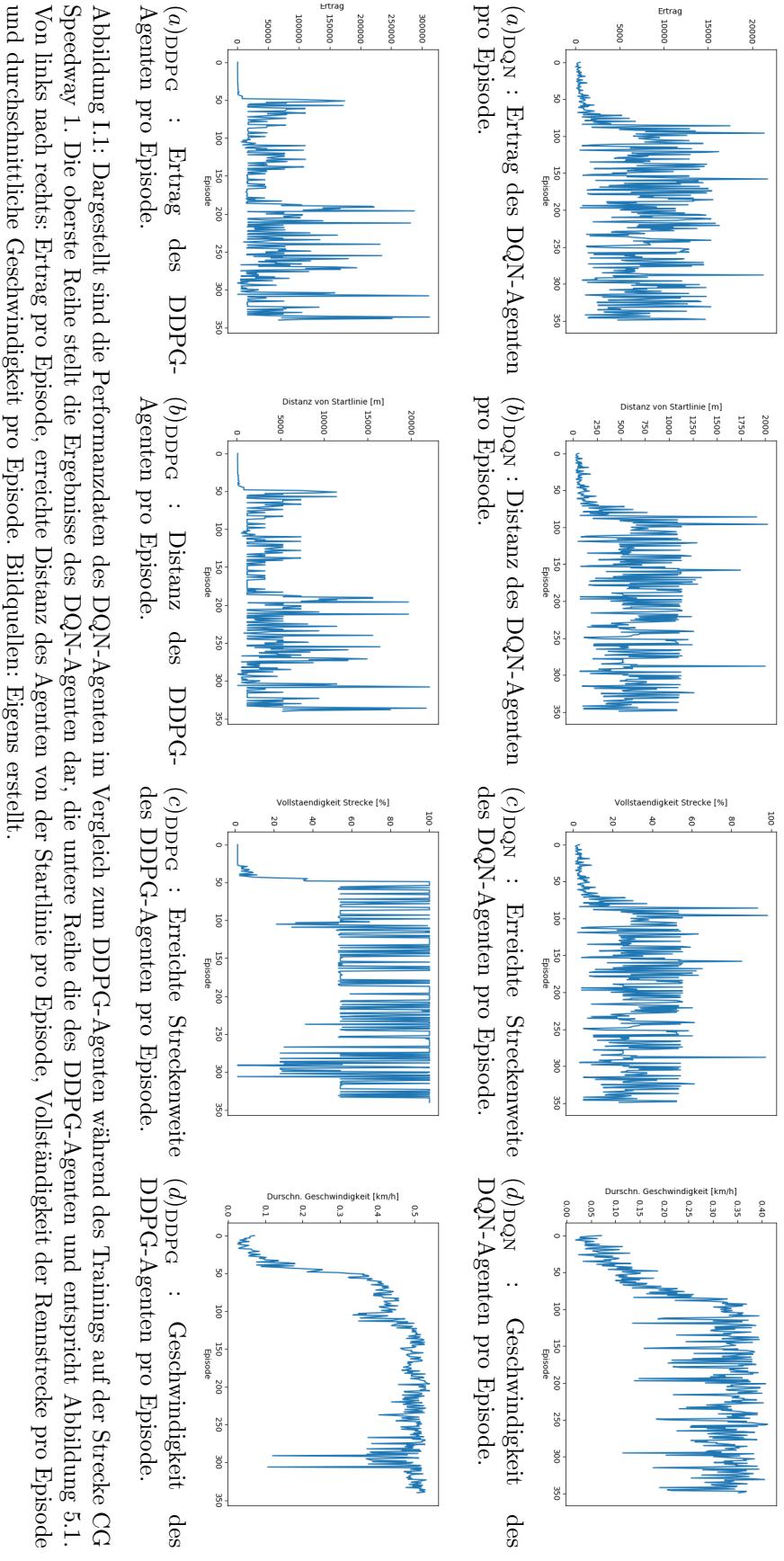


Abbildung H.1: Dargestellt ist der Vergleich der Performanzdaten des DDPG-Agenten während des Testens auf der Strecke CG Speedway unter den drei verschiedenen Belohnungsfunktionen. Die oberste Reihe stellt die Performanz des Agenten unter \mathcal{R}_1 dar. Die mittlere Reihe zeigt die Performanzdaten des Agenten unter \mathcal{R}_2 . Die unterste Reihe zeigt die Performanzdaten des Agenten unter \mathcal{R}_3 . Von links nach rechts: (a) $_{\mathcal{R}_i}$ durchschnittliche Geschwindigkeit pro Episode, (b) $_{\mathcal{R}_i}$ Vollständigkeit der Rennstrecke pro Episode, (c) $_{\mathcal{R}_i}$ Anzahl an Kollisionen pro Episode und (d) $_{\mathcal{R}_i}$ Anzahl an Kollisionen bezogen auf die erreichte Distanz pro Episode. Für die Performanzdaten (c) $_{\mathcal{R}_i}$ wurden überdies folgende statistische Kennzahlen errechnet: $\mu_{\mathcal{R}_1} = 9, 58$, $\sigma_{\mathcal{R}_1} = 9, 91$, $\mu_{\mathcal{R}_2} = 6, 40$, $\sigma_{\mathcal{R}_2} = 8, 28$, $\mu_{\mathcal{R}_3} = 7, 74$ und $\sigma_{\mathcal{R}_3} = 8, 91$. Da die Erträge unterschiedlicher Belohnungsfunktionen entspringen und somit nicht direkt vergleichbar sind, fehlt die Darstellung der Performanzdaten dieser Metrik in dieser Abbildung. Bildquellen: Eigens erstellt.

I Vergleich DQN- mit DDPG-Agent



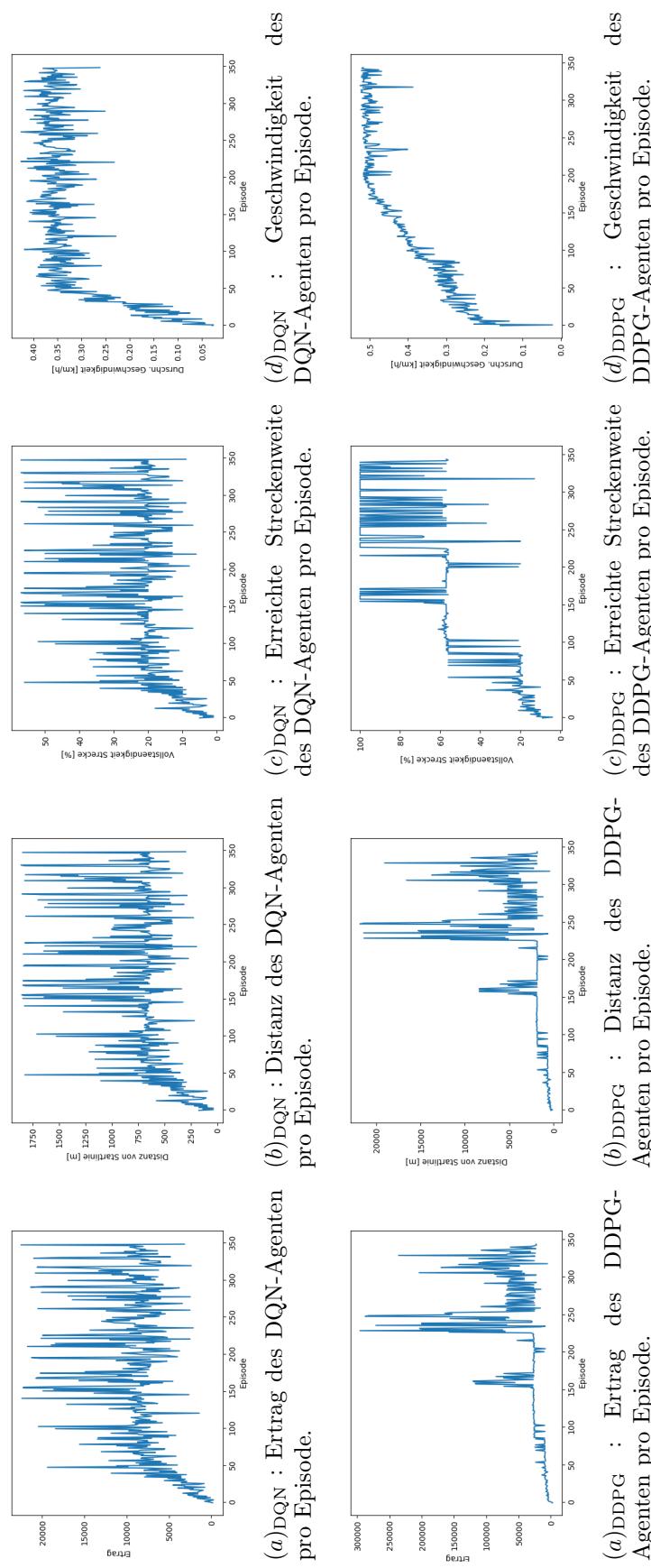
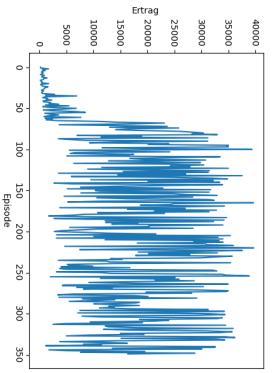
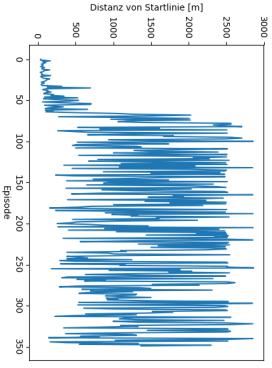


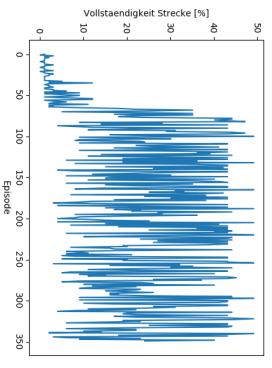
Abbildung I.2: Dargestellt sind die Performanzdaten des DQN-Agenten im Vergleich zum DDPG-Agenten während des Trainings auf der Strecke E-Road. Die oberste Reihe stellt die Ergebnisse des DQN-Agenten dar, die untere Reihe die des DDPG-Agenten und entspricht Abbildung 5.2. Von links nach rechts: Ertrag pro Episode, erreichte Distanz des Agenten von der Startlinie pro Episode, Vollständigkeit der Rennstrecke pro Episode und durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.



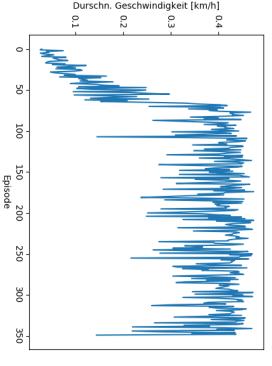
(a)DQN : Ertrag des DQN-Agenten pro Episode.



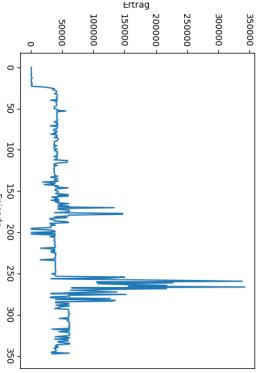
(b)DQN : Distanz des DQN-Agenten pro Episode.



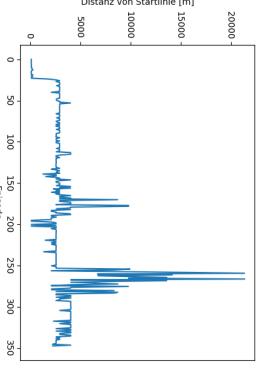
(c)DQN : Erreichte Streckenweite des DQN-Agenten pro Episode.



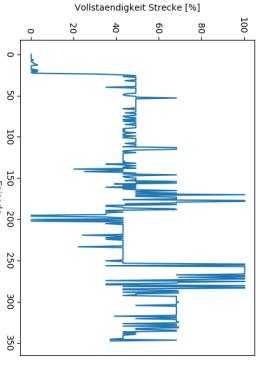
(d)DQN : Geschwindigkeit des DQN-Agenten pro Episode.



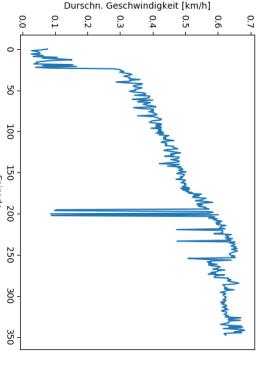
(a)DDPG : Ertrag des DDPG-Agenten pro Episode.



(b)DDPG : Distanz des DDPG-Agenten pro Episode.



(c)DDPG : Erreichte Streckenweite des DDPG-Agenten pro Episode.



(d)DDPG : Geschwindigkeit des DDPG-Agenten pro Episode.

Abbildung 1.3: Dargestellt sind die Performanzdaten des DQN-Agenten im Vergleich zum DDPG-Agenten während des Trainings auf der Strecke Forza. Die oberste Reihe stellt die Ergebnisse des DQN-Agenten dar, die untere Reihe die des DDPG-Agenten und entspricht Abbildung 5.3. Von links nach rechts: Ertrag pro Episode, erreichte Distanz des Agenten von der Startlinie pro Episode, Vollständigkeit der Rennstrecke pro Episode und durchschnittliche Geschwindigkeit pro Episode. Bildquellen: Eigens erstellt.

J Datenträger mit Implementierung

Dieser Arbeit beiliegend ist ein Datenträger, der die Python-Implementierungen der DQN- und DDPG-Algorithmus beinhaltet. Alternativ kann über https://github.com/koeller21/ma_code auf den Programmcode zugegriffen werden. Die beiden Algorithmen können über das Programm `pyrl.py` gestartet werden. Der Syntax hierzu ist:

```
python3 pyrl.py <dqn | ddpg> <train | test> <eroad | cgspeedway | forza>
```

Wissenswert ist zudem, dass im Test-Modus die trainierten Gewichte automatisch geladen werden. Außerdem muss die Streckenauswahl manuell erfolgen. Das bedeutet, dass VTORCS zunächst gestartet und dann eine Strecke, entweder im *Practice*- oder *Quick Race*-Modus, ausgewählt werden muss. In der Folge wird der TORCS-Bildschirm blau und das Programm `pyrl.py` kann gestartet werden.