

Evaluierung von Reinforcement Learning Algorithmen zur Erweiterung eines bestehenden Trajektorienfolgeregelungskonzeptes

Masterarbeit

eingereicht von: Laura Jasmin Witt

eingereicht am: 04.12.2019

Erstprüfer: Prof. Dr. Daniel Göhring

Zweitprüfer: Prof. Dr. Dr. (h.c.) habil. Raúl Rojas

Rechtliche und Eidesstattliche Erklärung

Die Ergebnisse, Meinungen und Schlüsse dieser Arbeit sind nicht notwendigerweise die der Volkswagen AG.

Ich versichere hiermit, dass ich die vorliegende Masterarbeit selbstständig verfasst und keine anderen als die im Literaturverzeichnis angegebenen Quellen benutzt habe. Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder noch nicht veröffentlichten Quellen entnommen sind, sind als solche kenntlich gemacht. Diese Arbeit ist in gleicher oder ähnlicher Form noch bei keiner anderen Prüfungsbehörde eingereicht worden.

Wolfsburg, 4. Dezember 2019

Laura Jasmin Witt

Kurzfassung

Aktuelle Trajektorienfolgeregelungskonzepte bestehend aus einer modellbasierten Vorsteuerung und einem modellfreien Regler ermöglichen bereits das automatisierte Fahren. Aufgrund von Ungenauigkeiten in den Systemmodellen und äußereren, sich ändernden Störeinflüssen werden zusätzlich vermehrt adaptive Regelungsverfahren eingesetzt, um eine kontinuierlich hohe Regelgüte zu garantieren.

Im Rahmen dieser Masterarbeit sollen verschiedene Reinforcement Learning Algorithmen bezüglich ihrer Eignung zur Erweiterung eines bestehenden Trajektorienfolgeregelungskonzeptes evaluiert werden. Mithilfe von künstlichen neuronalen Netzen werden sogenannte Agenten abgebildet, welche als additives Regelglied ein System bestehend aus Vorsteuerung und PID-Regler ergänzen sollen, um die Querdynamikregelung eines automatisierten Fahrzeugs zu optimieren.

In einer fahrzeughnahen Simulationsumgebung werden Fahrversuche durchgeführt, bei welchen Agenten sowohl mit verschiedenen Hyperparametern als auch verschiedenen Reinforcement Learning Algorithmen trainiert werden. Im Rahmen dieser Untersuchungen konnte eine Methode gefunden werden, welche in Kombination mit dem bestehenden Regelungskonzept zu einer signifikanten Verbesserung des Fahrzeugverhaltens führt. Zur Bewertung der Regelung wird vorwiegend die Abweichung zwischen Soll- und Isttrajektorie betrachtet, welche deutlich reduziert werden konnte.

Inhaltsverzeichnis

I Abbildungsverzeichnis	IV
II Tabellenverzeichnis	VII
III Abkürzungsverzeichnis	VIII
IV Symbolverzeichnis	IX
1 Einleitung	1
1.1 Motivation	1
1.2 Stand der Technik	2
1.3 Ziel und Aufbau der Arbeit	3
2 Grundlagen	4
2.1 Reinforcement Learning	4
2.1.1 Elemente von Reinforcement Learning Systemen	5
2.1.2 Markov-Entscheidungsprozesse	6
2.1.3 Approximierung der Value-Function	9
2.1.4 Policy-based Verfahren	13
2.2 Künstliche neuronale Netze	16
2.2.1 Aufbau von künstlichen neuronalen Netzen	16
2.2.2 Lernen in künstlichen neuronalen Netzen	18
3 Algorithmen	22
3.1 Auswahl der Algorithmen	22
3.2 Deep Deterministic Policy Gradient	23
3.3 Policy Gradient and Q-Learning	26
3.4 Actor-Critic using Kronecker-Factored Trust Region	28
4 Validierung der Algorithmen	32
4.1 Netzwerkarchitekturen und Hyperparameter	33
4.2 Simulationsergebnisse	35
4.3 Vergleich der Ergebnisse und Fazit	42
5 Verwendung der Algorithmen zur Erweiterung eines Trajektorienfolgeregelungskonzeptes	44
5.1 Aufbau	44
5.1.1 Aufbau der Simulationsumgebung	44
5.1.2 Aufbau des RL-Agenten	47
5.2 Einbindung des RL-Agenten in die Simulationsumgebung	48
5.3 Referenzfahrt	51
5.4 Simulationsergebnisse	54
5.5 Vergleich der Ergebnisse und Fazit	67
6 Zusammenfassung und Ausblick	69
Literatur	71

I Abbildungsverzeichnis

1.1	Aufgabenbereiche des autonomen Fahrens nach [1, 2]	1
2.1	Bereiche von maschinellem Lernen angelehnt an [3]	4
2.2	Elemente von Reinforcement Learning	5
2.3	Interaktion zwischen RL-Agent und Umwelt nach [4]	7
2.4	Vergleich der Backupdiagramme für DP, MC und TD nach [5]	9
2.5	Ablauf von Policy Iteration nach [4]	10
2.6	Policy Iteration und Monte-Carlo Policy Iteration nach [4]	12
2.7	Einteilung von RL-Verfahren angelehnt an [6]	15
2.8	Künstliches Neuron	16
2.9	Aktivierungsfunktionen	17
2.10	MLP mit n Eingängen, m Ausgängen und l versteckten Schichten	17
3.1	Approximation der inversen FIM mittels K-FAC nach [7]	30
4.1	Umgebung des Continuous Mountain Cars	32
4.2	Netzarchitekturen für Actor-KNN	33
4.3	Netzarchitekturen für Critic-KNN	34
4.4	Verlauf des Gewichtungsfaktors n für das externe Rauschen bei DDPG	35
4.5	Vergleich von zwei DDPG-Agenten mit verschiedenen Eingangsdaten	36
4.6	Vergleich von zwei PGQL-Agenten mit verschiedener Gewichtsinitialisierung	37
4.7	Ausgaben des Actor-KNN bei Gewichtsinitialisierung 1	38
4.8	Gaußverteilung für das Sampling der Actions bei verschiedenen Gewichtsinitialisierungen	38
4.9	Ausgaben des Actor-KNN bei Gewichtsinitialisierung 2	39
4.10	Vergleich von ACKTR- und PGQL-Agent mit ReLU-Aktivierung	40
4.11	Vergleich von zwei ACKTR-Agenten mit verschiedenen Aktivierungsfunktionen	41
4.12	Vergleich der finalen Agenten für verschiedene Anfangszustände des CMC	42

5.1	Fahrzeugkoordinatensystem angelehnt an [8,71]	45
5.2	Simulationsumgebung zur automatisierten Fahrzeugführung nach [9]	46
5.3	Komponenten des Querdynamikregelungskonzeptes nach [9]	46
5.4	Vereinfachte Darstellung der Prozesse des RL-Agenten	47
5.5	Gaußkurve zur Bestimmung des Rewardanteils der Querablage	50
5.6	Gefahrene Strecke in der Fahrzeugsimulation	52
5.7	Querablage bei der Referenzfahrt	52
5.8	Stellwinkel in Runde 7 der Referenzfahrt	53
5.9	Lateraler Reward bei der Referenzfahrt	53
5.10	Lateraler Reward von drei DDPG-Agenten mit allen nach Tabelle 5.1 verfügbaren Eingangsgrößen	54
5.11	Rewardanteile von drei DDPG-Agenten mit allen nach Tabelle 5.1 verfügbaren Eingangsgrößen	55
5.12	Lateraler Reward von drei DDPG-Agenten mit 15 der verfügbaren Eingangsgrößen	56
5.13	Stellwinkel und Querablage des DDPG-Agenten mit Architektur 3 und 15 der verfügbaren Eingangsgrößen in Runde 1	57
5.14	Rewardanteile von drei DDPG-Agenten mit 15 der verfügbaren Eingangsgrößen	57
5.15	Vergleich der Stellwinkel und der Querablage von zwei verschiedenen DDPG-Agenten mit 15 der verfügbaren Eingangsgrößen	59
5.16	Lateraler Reward von drei PGQL-Agenten mit 15 der verfügbaren Eingangsgrößen	60
5.17	Rewardanteile von drei PGQL-Agenten mit 15 der verfügbaren Eingangsgrößen	61
5.18	Zusammenhang zwischen Stellwinkel und Reward des PGQL-Agenten mit Architektur 1	61
5.19	Lateraler Reward von drei ACKTR-Agenten mit verschiedenen Netzarchitekturen	63
5.20	Rewardanteile von drei ACKTR-Agenten mit verschiedenen Netzarchitekturen	63
5.21	Prädizierte Standardabweichung des ACKTR-Agenten mit Architektur 1 . .	64

5.22 Stellwinkel des ACKTR-Agenten mit Architektur 1 in Runde 1 und 120	65
5.23 Rundenbasierte Standardabweichung für das Training eines ACKTR-Agenten	65
5.24 Lateraler Reward von einem ACKTR-Agenten mit Architektur 1	66
5.25 Rewardanteile von einem ACKTR-Agenten mit Architektur 1	66
5.26 Lateraler Reward des besten Agenten für jeden Algorithmus	67

II Tabellenverzeichnis

4.1 Bestandteile und Wertebereiche für Zustand und Aktion beim Continuous Mountain Car	32
4.2 Algorithmenspezifische Hyperparameter	34
4.3 Verglichene Zustände von ACKTR- und PGQL-Agent mit ReLU-Aktivierung	40
4.4 Ergebnisse des Agenten-Tests beim Continuous Mountain Car	43
5.1 Verfügbare Fahrzeuggrößen zum Erlernen des Lenkwinkels, der Gaspedal- sowie der Bremspedalstellung in der Fahrzeugsimulation	49
5.2 Erlernbare Actions der RL-Agenten	49
5.3 Verwendete Neuronenanzahlen in der Simulation	50

III Abkürzungsverzeichnis

A3C	Asynchronous Advantage Actor-Critic
ACKTR	Actor-Critic using Kronecker-Factored Approximation
CMC	Continuous Mountain Car
DP	Dynamische Programmierung
DDPG	Deep Deterministic Policy Gradient
ELU	Exponential Linear Unit
FIM	Fisher-Information-Matrix
GPU	Grafikprozessor
KI	Künstliche Intelligenz
KL	Kullback-Leibler
KNN	Künstliches Neuronales Netz
MC	Monte-Carlo
MDP	Markov-Entscheidungsprozess
ML	Maschinelles Lernen
MLP	Multi-Layer-Perzepron
MP	Markov-Prozess
MRP	Markov-Reward-Prozess
OU	Ornstein-Uhlenbeck
PG	Policy Gradient
PGQL	Policy Gradient and Q-Learning
PPO	Proximal Policy Optimization
ReLU	Rectifier Linear Unit
RL	Reinforcement Learning
RMS	Root Mean Square, quadratisches Mittel
SGD	Stochastic Gradient Descent
TD	Temporal-Difference
TRPO	Trust Region Policy Optimization

IV Symbolverzeichnis

Symbol	Einheit	Beschreibung
a	-	Aktion
a	m/s^2	Beschleunigung
A_t	-	Aktion zum Zeitpunkt t
$A(s, a)$	-	„Vorteil“ der Wahl einer Aktion a gegenüber allen anderen im Zustand s
\mathcal{A}	-	Aktionsraum
$\mathcal{A}(s)$	-	Menge aller möglichen Aktionen im Zustand s
b	-	Bias in einem KNN
e_y	m	Querablage
\mathbb{E}	-	Erwartungswert
F^{-1}	-	inverse Fisher-Information-Matrix
g	-	Gradient gespeichert in einem Neuron
G_t	-	Return zum Zeitpunkt t
h_{ia}	-	Neuronenanzahl in der i-ten versteckten Schicht des Actor-KNN
h_{ic}	-	Neuronenanzahl in der i-ten versteckten Schicht des Critic-KNN
H^π	-	Entropie der Policy π
$J(\theta)$	-	Güte-Maß
L	-	Loss-Funktion
m	-	Parameter bei Adam, laufender Mittelwert des Gradienten
\mathcal{N}	-	Normalverteilung
\mathcal{N}	-	Rauschprozess (bei DDPG)
o	-	Ausgabe eines Neurons
P	-	Übergangswahrscheinlichkeitsfunktion
q_π	-	State-Action-Value-Function
$q_\pi(s, a)$	-	State-Action-Value, wenn Aktion a im Zustand s gewählt wird
q_*	-	optimale State-Action-Value-Function
$Q(S_t, A_t)$	-	Value des Zustand-Aktion-Paares (S, A) zum Zeitpunkt t
Q	-	Critic-KNN (bei DDPG)
Q'	-	Target-Critic-KNN (bei DDPG)
r	-	Reward
R_t	-	Reward zum Zeitpunkt t
\mathcal{R}	-	Menge aller möglichen Rewards
\mathcal{R}	-	Replaybuffer
s	-	Zustand, State
S_t	-	Zustand zum Zeitpunkt t
\mathcal{S}	-	Menge aller möglichen Zustände
t	-	Zeitpunkt

Symbol	Einheit	Beschreibung
$V(S_t)$	-	Value des Zustands S zum Zeitpunkt t
v	-	Parameter bei Adam, laufender Mittelwert des quadratischen Gradienten
v_π	-	State-Value-Function
$v_\pi(s)$	-	State-Value von Zustand s
v_*	-	optimale State-Value-Function
$v_*(s)$	-	optimaler State-Value von Zustand s
v_x	m/s	Geschwindigkeit in Längsrichtung
v_y	m/s	Geschwindigkeit in Querrichtung
w	-	Gewicht in einem KNN
y	-	Zielwert, Target
α	-	Lernrate
β	rad	Schwimmwinkel
β_1	-	Hyperparameter von Adam
β_2	-	Hyperparameter von Adam
γ	-	Gewichtungsfaktor bei der Berechnung des Returns
δ	-	rückpropagierter Fehler
δ	-	Temporal Differenz Error
δ	-	Trust Region Hyperparameter (bei ACKTR)
δ_H^{ges}	rad	Lenkwinkel des gesamten Regelungskonzeptes
δ_H^R	rad	Additiver Stellwinkel des PID-Reglers
δ_H^{RL}	rad	Additiver Stellwinkel des RL-Agenten
δ_H^{soll}	rad	Soll-Lenkwinkel
δ_H^V	rad	Additiver Stellwinkel der Vorsteuerung
δ_t	-	
η	-	Lernrate in KNN
η	-	Parameter zur Gewichtung der Updates (bei PGQL)
Θ	-	Fahrzustandsgrößen
θ^μ	-	Gewichte/Parameter des Actor-KNN (bei DDPG)
$\theta^{\mu'}$	-	Gewichte/Parameter des Target-Actor-KNN (bei DDPG)
θ^π	-	Gewichte/Parameter des Actor-KNN (bei PGQL und ACKTR)
θ^Q	-	Gewichte/Parameter des Critic-KNN (bei DDPG)
$\theta^{Q'}$	-	Gewichte/Parameter des Target-Critic-KNN (bei DDPG)
θ^V	-	Gewichte/Parameter des Critic-KNN (bei PGQL und ACKTR)
κ	m^{-1}	Kurvenkrümmung
μ	-	Mittelwert
μ	-	Actor-KNN, deterministische Policy (bei DDPG)
μ'	-	Target-Actor-KNN (bei DDPG)
π	-	Policy

Symbol	Einheit	Beschreibung
π_*	-	optimale Policy
σ	-	Standardabweichung
τ	-	Parameter für Target-Network-Updates (bei DDPG)
ψ	rad	Gierwinkel
$\dot{\psi}$	rad/s	Gierwinkelgeschwindigkeit
$\ddot{\psi}$	rad/s ²	Gierwinkelbeschleunigung

1 Einleitung

1.1 Motivation

Autonomes Fahren ist einer der wichtigsten Technik-Trends der letzten Jahre [10]. Während aktuelle Fahrzeuge bereits eine Vielzahl an Fahrerassistenzsystemen bieten, werden diese kontinuierlich weiterentwickelt, um hochautomatisiertes und letztendlich autonomes Fahren zu ermöglichen. Für die Umsetzung einer autonomen Fahrfunktion ist eine vollständige Automatisierung der bisher weitestgehend vom Menschen übernommenen Fahraufgabe notwendig. Diese kann beispielsweise nach [11] in eine Drei-Ebenen-Hierarchie eingeteilt und soll exemplarisch am Beispiel eines Überholmanövers erläutert werden. Nach Beobachtung seiner Umgebung inklusive der umliegenden Fahrspuren sowie aller anderen Verkehrsteilnehmer muss der Fahrer einen realisierbaren und sicheren Spurwechsel planen. Anhand dieser geplanten Route muss der Fahrer Führungsgrößen, wie beispielsweise die Solllinie und -geschwindigkeit festlegen. Im weiteren Fahrtverlauf müssen Abweichungen zwischen Soll- und Istgrößen durch den Fahrer kompensiert werden, um entsprechend ein sicheres und stabiles Verhalten des Fahrzeugs zu garantieren. Für die automatisierte Fahrzeugführung muss ein Fahrzeug gleichbedeutende Aufgaben aus den Bereichen Wahrnehmung, Planung und Regelung erfolgreich absolvieren, wie Abbildung 1.1 zeigt.

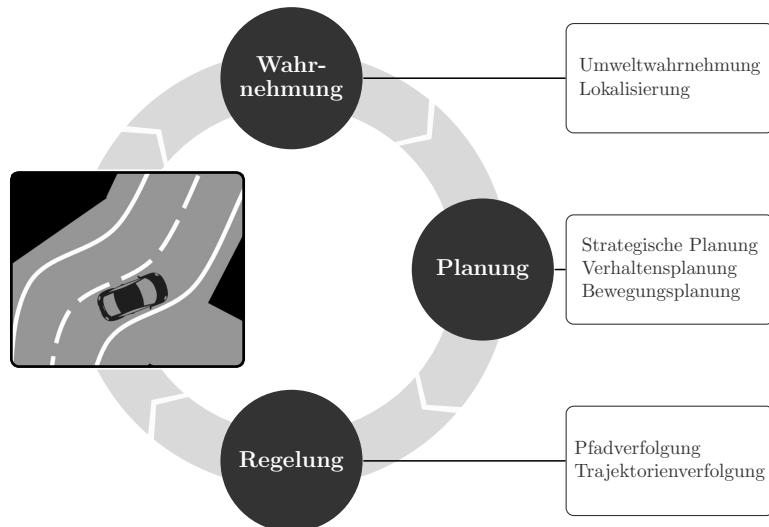


Abbildung 1.1: Aufgabenbereiche des autonomen Fahrens nach [1, 2]

Während der Bereich Wahrnehmung neben der Lokalisierung auch das Erfassen der Fahrzeugumwelt und des Fahrzeugzustandes beinhaltet, wird in der Planung basierend auf diesen Informationen beispielsweise die gewünschte Fahrtroute und die entsprechende Fahrtrajektorie definiert. Die Umsetzung dieses Wunschverhaltens in das tatsächliche Fahrzeugverhalten kann mithilfe verschiedener Regelungsansätze realisiert werden [2]. Unter der Voraussetzung einer fehlerfreien Wahrnehmung und Planung muss die Regelung jederzeit die Stabilität des Fahrzeugs und somit die Sicherheit der Fahrzeuginsassen garantieren. Um dementsprechend die Stabilität des Systems in den vorgegebenen Grenzen zu gewährleisten, wird eine hohe Regelgüte über den gesamten Dynamikbereich des Fahrzeugs notwendig. In traditionellen, modellbasierten Regelungsansätzen entstehen erste Schwierigkeiten bereits bei der Modellierung des Systems, da nichtlineare Dynamiken schwer abbildbar sind.

Zusätzlich kollidiert die häufig hohe Modellkomplexität mit der geforderten Randbedingung der Echtzeitfähigkeit des Systems. Hinzu kommen äußere, sich ändernde Störeinflüsse und Verschleißerscheinungen am Fahrzeug, welche nur bedingt identifizierbar und entsprechend nur mit hohem Aufwand in das Regelungsmodell integrierbar sind. Um diese Probleme zu lösen, werden vermehrt Kombinationen aus modellbasierten und adaptiven Regelungssystemen eingesetzt. Adaptive Systeme lösen komplexe Regelungstechnische Aufgaben auf Basis von Messdaten ohne ein zugrundeliegendes physikalisches Modell zu benötigen. Aufgrund der kontinuierlich fortschreitenden Technik und damit verbunden einer deutlichen Steigerung der Rechenleistung ergeben sich neue Einsatzmöglichkeiten für verschiedenste Verfahren, welche Adaptionen in Echtzeit ermöglichen. Durch die steigende Effizienz von Algorithmen der künstlichen Intelligenz finden diese vermehrt Einsatz in der datengetriebenen Optimierung von Regelungssystemen [12].

1.2 Stand der Technik

Maschinelles Lernen (ML) und künstliche Intelligenz (KI) werden im Bereich des automatisierten Fahrens bereits seit mehreren Jahren eingesetzt. Dabei existiert eine Vielzahl an Ansätzen zur Übernahme von Aufgaben im Bereich der Umweltwahrnehmung, mit welchen der Fahrer unterstützt werden soll. In [13, 14] wird beispielsweise die Eignung von künstlichen neuronalen Netzen (KNN) zur Erkennung von Ampeln und in [15, 16] zur Erkennung bzw. Klassifizierung von Verkehrszeichen gezeigt. Dabei werden überwiegend Verfahren des Supervised Learnings eingesetzt.

Ein relevanter Bestandteil der derzeitigen Forschung ist Reinforcement Learning (RL), das neben Supervised Learning und Unsupervised Learning ein Teilgebiet des maschinellen Lernens ist. Nachfolgend wird sich auf die Verwendung von modellfreien RL-Algorithmen fokussiert. Im Bereich des automatisierten Fahrens wurden entsprechende Verfahren bereits in der Planung sowie in der Regelung eingesetzt.

In [17] wird mithilfe von Verfahren des Reinforcement Learnings simulativ die optimale Fahrstrategie eines autonomen Fahrzeugs auf einer mehrspurigen Straße erlernt. Dabei wird basierend auf einer State-Repräsentation in Form von Zellenarrays eine von fünf diskreten Aktionen gewählt.

Weiterhin wird RL in verschiedener Literatur im Bereich der Fahrzeugregelung eingesetzt. Basierend auf Bilddaten existieren mit Beispielen wie [18, 19] bereits mehrere Ansätze zur Vorhersage von Lenkwinkeln in Fahrzeugsimulationen. In [20] wird statt des direkten Kameralbildes ein Wahrnehmungsmodul mit einer Encoder-Decoder-Architektur verwendet. Die Ausgabe des Encoder-Netzwerk stellt entsprechend eine niedrigdimensionalere Repräsentation der Umwelt dar, welche zum Erlernen eines diskretisierten Lenkwinkels für ein Fahrzeug in der Simulation verwendet wird. In [21] wird ein Wahrnehmungsmodul in Form eines Convolutional Neural Networks verwendet, um Streckeninformation aus den Bilddaten eines Rennsimulators zu extrahieren. Diese werden anschließend mit Informationen über den Fahrzustand, wie beispielsweise der Fahrzeuggeschwindigkeit und -ausrichtung, kombiniert, um auf dieser Basis einen kontinuierlichen Lenkwinkel zu prädizieren.

In [22] wird ein Reinforcement Learning Algorithmus verwendet, um ein Fahrzeug in einem Rennsimulator zu steuern. Abweichend von den vorherigen Ansätzen werden keine Informationen über den aktuellen Zustand des Fahrzeugs aus Bilddaten extrahiert. Stattdessen werden verschiedene Sensorinformationen, wie beispielsweise die Geschwindigkeit, die Abweichung zwischen der Fahrzeulgängsachse und der idealen Spur sowie die Abstände zu den Streckenrändern, für die Aktionsauswahl des Agenten verwendet. Basierend auf diesen werden Entscheidungen für das Bremsen, die Beschleunigung sowie das Lenken getroffen.

Die drei Aktionen können dabei ausschließlich diskrete Werte annehmen, die entweder kein Gas bzw. Vollgas, kein Bremsen bzw. Vollbremsung oder voller Lenkeinschlag nach links bzw. rechts bedeuten.

Mit [23–26] existieren vermehrt Anwendung im Bereich der Robotik außerhalb der Simulation, wobei beispielsweise Roboter das Laufen oder Modellhubschrauber das autonome Fliegen erlernen. In [27] lernt ein Auto durch Interaktion mit seiner Umwelt das Stellen eines Lenkwinkels, wobei die Längsführung des Fahrzeugs von einem menschlichen Fahrer übernommen wird. Als Repräsentation des Fahrzustands stehen fünf Messgrößen zur Verfügung, welche den Fahrzustand beschreiben und entsprechend als Basis für die Aktionsauswahl dienen. Vor Ausführung des prädierten Lenkwinkels wird dieser weiterverarbeitet. Um die Sicherheit des Fahrzeugs zu garantieren, wird in diesem Fahrversuch ein Korrekturverhalten angewendet, welches übernimmt, wenn sich das Fahrzeug in einen unsicheren Zustand begibt.

Alle der vorgestellten Ansätze übernehmen die komplette Regelung der Fahrzeugquerführung und bilden keine Regelungsstrukturen bestehend aus mehreren additiven Komponenten.

1.3 Ziel und Aufbau der Arbeit

Die Regelung eines Fahrzeuges stellt eine komplexe Aufgabe mit einer Vielzahl an Zuständen und Aktionen dar, weshalb im Rahmen dieser Masterarbeit verschiedene modellfreie Reinforcement Learning Algorithmen gewählt und bezüglich ihrer Eignung zur unterstützenden Regelung eines automatisierten Fahrzeugs evaluiert werden sollen.

Auf Basis einer umfangreichen Literaturrecherche wurde im vorherigen Abschnitt der aktuelle Stand von Reinforcement Learning im Bereich der Robotik und im Speziellen in der Fahrzeugführung aufgezeigt. Nachfolgend sollen in Kapitel 2 zuerst die Grundlagen von Reinforcement Learning und künstlichen neuronalen Netzen erläutert werden, welche in RL-Agenten Anwendung finden können. Auf Basis der durchgeföhrten Literaturrecherche erfolgt eine Auswahl geeigneter RL-Verfahren, welche in Kapitel 3 vorgestellt werden. Die Implementierung der Algorithmen in Python wird anhand eines Beispiels mit kleinem, jedoch kontinuierlichem Aktions- und Zustandsraum, dem Continuous Mountain Car, validiert. Die entsprechenden Simulationsergebnisse sind in Kapitel 4 erläutert. Das fünfte Kapitel beschreibt die Fusion der implementierten Algorithmen mit der bestehenden Simulationsumgebung zur Trajektorienfolgeregelung. Die zur Verfügung gestellte Simulation beinhaltet bereits eine funktionstüchtige Regelung, welche einem Fahrzeug autonomes Agieren auf einem gegebenen Streckenprofil ermöglicht. Die RL-Verfahren sollen als zusätzliches Regelglied eingesetzt werden, um mittels datengetriebener Methoden das Führungs- und Störverhalten zu verbessern. Nach einer kurzen Zusammenfassung erfolgt abschließend ein Ausblick auf mögliche zukünftige Aufgaben.

2 Grundlagen

In diesem Kapitel sollen die notwendigen Grundlagen von Reinforcement Learning und deren Umsetzung mithilfe von künstlichen neuronalen Netzen erläutert werden.

In Kapitel 2.1 wird zunächst RL als Teilgebiet des maschinellen Lernens eingeordnet. Anschließend erfolgt die Erläuterung der einzelnen Elemente eines RL-Systems sowie die Beschreibung des Markov-Entscheidungsprozesses (MDP) als mathematisch idealisierte Darstellungsmöglichkeit. Auf Grundlage dessen werden mögliche Optimierungsmethoden vorgestellt, die basierend auf der Value-Function, der Policy oder einer Kombination von beidem agieren. Kapitel 2.2 befasst sich zuerst mit dem Aufbau von künstlichen neuronalen Netzen und deren Bestandteilen, den künstlichen Neuronen. Anschließend wird auf das Lernen in KNN eingegangen.

2.1 Reinforcement Learning

Reinforcement Learning ist bereits seit vielen Jahren aus den verschiedensten Bereichen, wie beispielsweise Statistik und Psychologie, bekannt. Seit etwa 30 Jahren beschäftigen sich ebenfalls vermehrt Forscher im Bereich des maschinellen Lernens und der künstlichen Intelligenz mit diesem Thema [28]. Als Teilgebiet der Informatik befasst sich Maschinelles Lernen mit dem Finden von Problemlösungen anhand von gesammelten Daten [29].

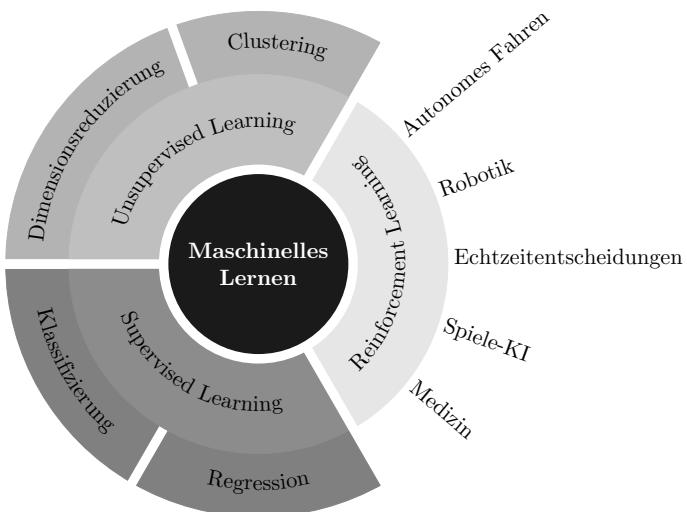


Abbildung 2.1: Bereiche von maschinellem Lernen angelehnt an [3]

Wie Abbildung 2.1 zeigt, ist maschinelles Lernen in drei große Bereiche einteilbar. Unsupervised Learning wird vorwiegend für Clustering oder Dimensionalitätsreduzierungen eingesetzt. Ziel des Clusterings ist es, Ähnlichkeiten bzw. Strukturen in einem vorliegenden, ungelabelten Datensatz zu finden und entsprechend zu Gruppen, sogenannten Clustern, zuzuordnen [4,29,30]. Dimensionsreduzierung vereinfacht eine Vielzahl an Aufgaben, indem Abhängigkeiten zwischen den Variablen erkannt und entsprechend Dimensionen ohne zusätzlichen Informationsgehalt entfernt werden. Ebenfalls wird die Visualisierung von hochdimensionalen Daten ermöglicht [30].

Bei Supervised Learning hingegen müssen zu den Daten die zugehörigen Label existieren. Für jeden Eingabevektor enthält der Datensatz einen Zielwert oder -vektor, dessen Zusammenhang durch das erlernte Modell abgebildet werden soll [29]. Ziel dieser Art von Lernen ist häufig das Generalisieren und dadurch das Extrapolieren von gesammelten Wissen auf unbekannte Daten [4]. Bei der Klassifizierung wird versucht anhand der vorhandenen Daten ein Modell zu lernen, welches alle Daten mit dem gleichen Label einer Klasse zuordnet. In der Vorhersagephase nach der Lernphase sollen ähnliche Datensätze der richtigen Klasse zugeordnet werden können, obwohl ihr Label unbekannt ist. Bei der Regression wird ein direkter Zusammenhang zwischen Ein- und Ausgabe erlernt [31].

Reinforcement Learning ist im Gegensatz zu den anderen Gebieten des maschinellen Lernens geprägt durch die Interaktion mit der Umwelt und somit angelehnt an das menschliche Lernverhalten. Vereinfacht ausgedrückt muss der sogenannte Agent in der Lage sein die Umwelt wahrzunehmen, relevante Informationen zu extrahieren, basierend auf diesen eine Handlung auszuwählen und diese auszuführen, um somit seine Umwelt zu beeinflussen. Der Agent versucht sein Verhalten kontinuierlich durch die Rückmeldung aus seiner Umgebung zu verbessern.

Reinforcement Learning wird in verschiedenen Bereichen für unterschiedlichste Anwendungen genutzt. In der Robotik ebenso wie im autonomen Fahren bietet RL beispielsweise Möglichkeiten für das Erlernen von schwer zu entwickelnden Verhaltensweisen. Anhand von Realdaten sind Systemanpassungen möglich, die anschließend zu einer verbesserten Verhaltensweise führen sollen. Besonders bekannt im Reinforcement Learning sind Erfolge aus dem Bereich der Spiele-KI, in welchen z.B. AlphaGo [32] fällt. In Kapitel 2.1.1 wird zunächst auf die Bestandteile eines RL-Systems eingegangen.

2.1.1 Elemente von Reinforcement Learning Systemen

Ein Reinforcement Learning System besteht aus zwei Hauptbestandteilen, dem Agenten und der Umwelt, welche unterschiedliche Elemente enthalten und über verschiedene Schnittstellen miteinander interagieren. Eine schematische Darstellung bietet Abbildung 2.2.

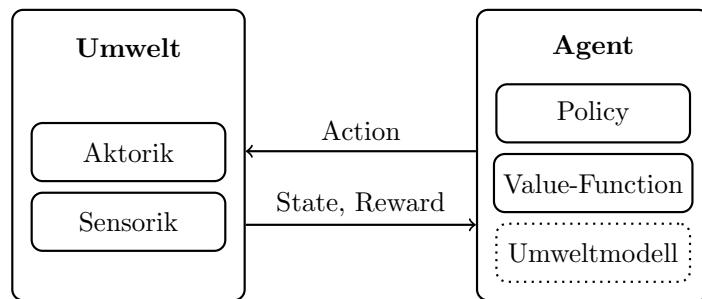


Abbildung 2.2: Elemente von Reinforcement Learning

Als Agent wird das System bezeichnet, welches in der Umgebung agiert. Dazu werden vorwiegend eine Policy und eine Value-Function approximiert. Optional kann ebenfalls ein Umweltmodell erlernt werden.

Eine Policy beschreibt eine Strategie bzw. Handlungsweise eines RL-Agenten ausgehend von einem bestimmten Zustand. Lernt der Agent eine deterministische Policy, so liefert

seine Policy eine explizite Aktion. Handelt der Agent nach einer stochastischen Policy, so liefert diese eine Verteilung über alle möglichen Aktionen basierend auf dem aktuellen Zustand [33]. Der Aktionsraum \mathcal{A} kann dabei je nach gegebener Umgebung diskret oder kontinuierlich sein.

Mithilfe einer Value Function bewertet der Agent seinen aktuellen Zustand bzw. das Wählen einer Aktion in diesem. Die State-Value-Function $v_\pi(s)$ gibt an, wie hoch die zu erwartende Belohnung ausgehend von dem Zustand s für den Agenten ist. Bewertet der Agent, wie gut es ist in einem Zustand s eine Aktion a zu wählen und anschließend seiner Policy zu folgen, nutzt dieser eine State-Action-Value-Function $q_\pi(s, a)$ [4]. Bei einigen RL-Algorithmen lernt der Agent zusätzlich ein Modell seiner Umwelt, um mithilfe von diesen Schlussfolgerungen über das Verhalten der Umwelt treffen zu können. Infolgedessen wird beispielsweise das Prädizieren des nachfolgenden Zustands oder des zu erwartenden Rewards für eine getätigte Aktion möglich. Solche Verfahren werden als modellbasierte Methoden bezeichnet. Bei modellfreien Algorithmen, welche in dieser Arbeit verwendet werden, erkundet der Agent seine Umwelt lediglich auf Basis von Trial-and-Error und lernt somit keine Dynamiken der Umwelt sondern direkt oder indirekt eine Policy, mit welcher der erhaltene Reward maximiert werden soll.

Die Trennung zwischen Agent und Umwelt entspricht im Reinforcement Learning nicht zwingend den physikalischen Grenzen des Agenten. Der Begriff Umwelt bezeichnet in diesem Zusammenhang alle Komponenten außerhalb der Entscheidungsfindung des Agenten. Neben der die Aktion ausführenden Aktorik enthält die Umwelt Sensorik, mit deren Hilfe der aktuelle Zustand erfasst wird. Ist es dieser möglich die gesamte Umgebung zu beobachten und im State abzubilden, entspricht der Agenten-State dem Umwelt-State. Existieren weitere, nicht durch die Sensorik wahrnehmbare Einflüsse, unterscheiden sich die beiden Repräsentationen des Zustandes. Da der Agenten-State der relevante für das Lösen der RL-Aufgabe ist, wird dieser nachfolgend lediglich als State bzw. Zustand bezeichnet.

Der Zustandsraum \mathcal{S} kann genau wie der Aktionsraum \mathcal{A} diskret oder kontinuierlich sein. Der Reward ist ein skalarer Wert, der die Güte einer gewählten Aktion in einem gegebenen Zustand festlegt. Anhand dieses Bewertungssignals versucht der Agent seine Policy zu verbessern, um zukünftig situationsbedingt bessere und somit gewinnbringendere Aktionen zu wählen.

Ein bekanntes Problem bei Reinforcement Learning ist dabei der Trade-Off zwischen Exploration und Exploitation. Der Agent wird ihm bekannte, gute Aktionen für den gegebenen Zustand wählen, um den Reward zu maximieren. Infolgedessen werden möglicherweise bessere Aktionen nicht berücksichtigt und somit einige Zustände im Zustandsraum unbekannt bleiben. Der Agent muss also seine Umwelt erkunden, d. h. Aktionen abweichend der besten ihm bekannten Aktion wählen, um langfristig den höchstmöglichen Reward zu erhalten. Dies wird im kontinuierlichen Aktionsraum je nach verwendetem Algorithmus durch Sampling aus der Wahrscheinlichkeitsverteilung oder durch das Hinzufügen von zusätzlichem Rauschen erreicht.

Im nachfolgenden Abschnitt sollen die angesprochenen Elemente mithilfe einer mathematischen Beschreibungsmöglichkeit, dem Markov-Entscheidungsprozess, verdeutlicht werden.

2.1.2 Markov-Entscheidungsprozesse

Ein Markov-Prozess (MP) ist ein speicherloser, zufälliger Prozess, welcher durch ein Tupel bestehend aus Zustand S und Übergangsfunktion P beschrieben werden kann. Ein Markov-Reward-Prozess (MRP) enthält im Gegensatz zum MP ebenfalls eine Rewardfunktion, die angibt, wie hoch der zu erwartende Reward im Zustand S ist. Wird zusätzlich eine

Entscheidung getroffen, wird der Prozess als Markov-Decision-Prozess (engl. für Markov-Entscheidungsprozess) bezeichnet. Das beschreibende Tupel $(\mathcal{S}, \mathcal{A}, P, \mathcal{R}, \gamma)$ besteht aus einer Zustandsmenge \mathcal{S} , einer Aktionsmenge \mathcal{A} , der Übergangswahrscheinlichkeitsfunktion P , der Rewardfunktion \mathcal{R} und dem Gewichtungsfaktor γ [34]. MDPs können als mathematisch idealisierte Beschreibung für ein Reinforcement Learning Problem verwendet werden.

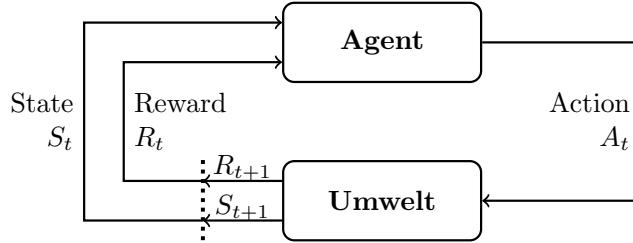


Abbildung 2.3: Interaktion zwischen RL-Agent und Umwelt nach [4]

Abbildung 2.3 verdeutlicht die Interaktion eines RL-Agenten mit seiner Umwelt. In jedem Zeitschritt t wählt der Agent auf Grundlage seines aktuellen Zustandes $S_t \in \mathcal{S}$ eine Aktion $A_t \in \mathcal{A}(s)$. Infolgedessen erhält er von seiner Umwelt einen Reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$ und einen neuen Zustand $S_{t+1} \in \mathcal{S}$. Dadurch erzeugt der Agent eine sogenannte Trajektorie der folgenden Form [4]:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots . \quad (2.1)$$

Ein Zustand besitzt die Markov-Eigenschaft, wenn dieser alle zukunftsbeeinflussenden Informationen der Vergangenheit enthält, d. h. ein Zustand S_t ist ausschließlich abhängig vom vorherigen Zustand S_{t-1} und der darin getätigten Aktion A_{t-1} . Im weiteren Verlauf der Arbeit wird die Markov-Eigenschaft als gegeben vorausgesetzt, wodurch sich die Komplexität der Umgebungsbeschreibung erheblich reduziert [4].

Die Wahrscheinlichkeit von einem Zustand s in einen anderen Zustand s' durch Ausführen einer Aktion a zu gelangen, wird durch die Übergangswahrscheinlichkeitsfunktion p beschrieben:

$$p(s'|s, a) \doteq \Pr \{ S_t = s' | S_{t-1} = s, A_{t-1} = a \} = p(s', r|s, a). \quad (2.2)$$

Der daraus folgende Reward r ergibt sich nach:

$$r(s, a) \doteq \mathbb{E} [S_{t-1} = s, A_{t-1} = a]. \quad (2.3)$$

Das Ziel eines Agenten ist es, den Erwartungswert des zu erhaltenden Rewards bis zum Erreichen eines Endzustandes zu vermehren. Formal ausgedrückt ist das Ziel das Maximieren des zu erwartenden Returns G_t , welcher im einfachsten Fall durch die Summe der Rewards definiert wird [4]:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \dots + R_T, \quad (2.4)$$

wobei T den letzten Zeitschritt und somit das Ende einer sogenannten Episode bezeichnet. Bei episodischen Aufgaben teilt sich die Interaktion mit der Umwelt selbstständig in einzelne, unabhängige Abschnitte. Aufgaben ohne Endzustände werden als kontinuierlich-

che Aufgaben bezeichnet. Um zu vermeiden, dass der durch Gleichung 2.4 berechenbare Return aufgrund des uneingeschränkten Zeithorizonts gegen unendlich strebt, werden die zukünftigen Rewards gewichtet:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}, \quad (2.5)$$

wobei $0 \leq \gamma \leq 1$ den Gewichtungsfaktor bezeichnet, welcher angibt, wie stark zukünftige Rewards einbezogen werden. Bei einem Gewichtungsfaktor von $\gamma = 0$ wird lediglich der aktuelle Reward berücksichtigt. Bei $\gamma = 1$ findet keine Gewichtung statt, weshalb alle zukünftigen sowie die aktuelle Situation gleich gewertet werden.

In nahezu jedem RL-Algorithmus wird die (State-)Value-Function v_π für einen State bzw. die (State-Action-)Value-Function q_π für ein State-Action-Paar approximiert, welche angibt, wie gut der entsprechende Zustand bzw. das Wählen einer bestimmten Aktion in diesem Zustand ist. Als Gütemaß wird der zu erwartende Return verwendet. Einer Value-Function liegt eine Handlungsweise eines Agenten zugrunde, welche als Policy π bezeichnet wird. Die Value-Function $v_\pi(s)$ eines Zustandes s mit gegebener Policy π gibt den zu erwartenden Return ausgehend von Zustand s an [4]:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right]. \quad (2.6)$$

Analog zu Gleichung 2.6 bezeichnet $q_\pi(s, a)$ den zu erwartenden Return, wenn in Zustand s Aktion a gewählt und anschließend der Policy π gefolgt wird [4]:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right]. \quad (2.7)$$

Der Value eines Endzustandes ist immer null, da ausgehend von diesem kein weiterer Reward zu erwarten ist. Die Value-Functions können durch Verwendung von beispielsweise Monte-Carlo (MC) oder Temporal Difference (TD) Methoden approximiert werden. TD kombiniert Ideen von Monte-Carlo Methoden und dynamischer Programmierung (DP), wobei letztere die sogenannten Bellman-Gleichungen lösen [4].

Eine Bellman-Equation bzw. Bellman-Gleichung besagt, dass die optimale Lösung eines Optimierungsproblems aus optimalen Teillösungen besteht. Durch Umformung von Gleichung 2.6 ergibt sich die Bellman-Gleichung für die Value-Funktion v_π :

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t | S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} | S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r | s, a) [r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s']] \\ &= \sum_a \pi(a|s) \sum_{s', r} p(s', r | s, a) [r + \gamma v_\pi(s')]. \end{aligned} \quad (\text{aus 2.6}) \quad (2.8)$$

Die Güte einer Policy kann mithilfe der Value-Function beurteilt werden. Eine Policy π ist besser als eine andere Policy π' , wenn $v_\pi(s) \geq v_{\pi'}(s)$ für alle Zustände $s \in \mathcal{S}$ gilt. Eine Policy wird als optimale Policy π_* bezeichnet, wenn der zugehörige Wert $v_\pi(s)$ größer als der aller anderen Policies ist. Alle optimalen Policies teilen die gleiche optimale State-Value

Function v_* :

$$v_*(s) \doteq \max_{\pi} v_{\pi}(s), \quad (2.9)$$

sowie die gleiche optimale Action-Value Function q_* :

$$q_*(s, a) \doteq \max_{\pi} q_{\pi}(s, a). \quad (2.10)$$

Das Erlernen einer optimalen Policy ist in der Praxis eingeschränkt, da der Agent durch die Verfügbarkeit von Rechenkapazitäten und Speicherplatz beschränkt wird [4].

2.1.3 Approximierung der Value-Function

Dynamische Programmierung, Monte-Carlo Methoden und Temporal Difference Learning sind die drei grundlegenden Klassen, in die Methoden zum Lösen von MDPs eingeteilt werden können [4]. Hierbei geht es vermehrt um das Approximieren von Value-Functions, von welchen anschließend die Handlungsweise eines Agenten abgeleitet werden kann.

Unter dynamischer Programmierung werden verschiedene Methoden zusammengefasst, mit welchen optimale Policies bei gegebenen Modellen der Umwelt bestimmt werden können. Im Gegensatz zur dynamischen Programmierung wird bei Monte-Carlo Methoden kein perfektes Modell der Umwelt benötigt. Stattdessen wird die komplette Episode bis zum Endzustand betrachtet, um basierend auf diesem sogenannten Rollout den Value eines Zustandes zu bestimmen. Temporal Difference Learning kombiniert Ideen der dynamischen Programmierung mit Monte-Carlo Methoden. Dabei wird der Zusammenhang der Values von aufeinanderfolgenden Zuständen ausgenutzt, um das komplette Rollout einer Episode zu umgehen. Abbildung 2.4 zeigt die Backup-Diagramme von Dynamischer Programmierung, Monte-Carlo- und Temporal Difference Learning.

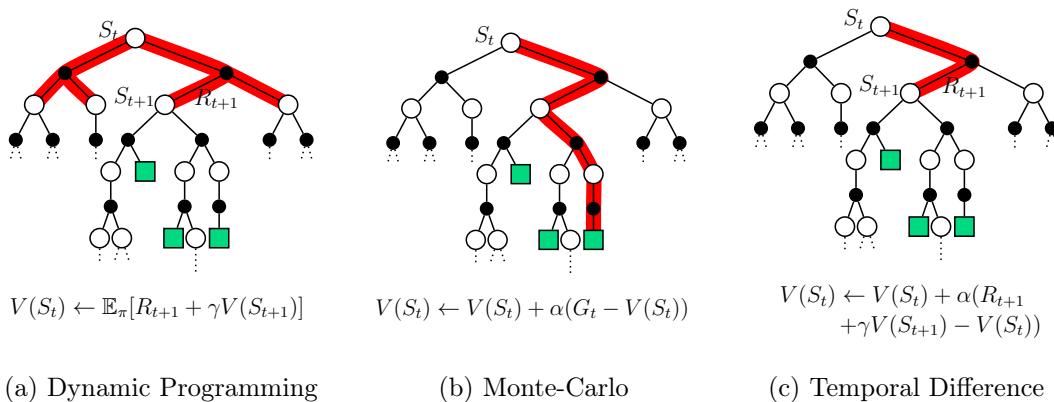


Abbildung 2.4: Vergleich der Backupdiagramme für DP, MC und TD nach [5]

Weisse Kreise markieren die Zustände, schwarze Kreise die Zustands-Aktionspaare und grüne Quadrate die Endzustände. Während bei DP alle möglichen Übergänge in Nachfolgezustände betrachtet und mit ihrer Auftrittswahrscheinlichkeit gewichtet werden, wird bei MC und TD nur eine mögliche Aktion berücksichtigt. Da bei MC-Methoden die Values der Zustände lediglich auf Basis des Returns berechnet werden, muss die gesamte Episode betrachtet werden. TD-Methoden nutzen wiederum die Zusammenhänge aufeinanderfolgender Zustände aus, um den Value eines Zustandes zu bestimmen.

Dynamische Programmierung

Da die von DP geforderten Modelle der Umwelt bei RL häufig nicht bekannt sind, sind DP-Verfahren nur bedingt anwendbar. Trotzdem stellen sie die Grundlage für den Großteil aller Reinforcement Learning Methoden dar, weshalb anschließend einige Grundkonzepte der dynamischen Programmierung erläutert werden sollen [4]. *Policy Evaluation* evaluiert die Value-Function, um die Güte einer gegebenen Policy zu bewerten. Die Values aller Endzustände werden mit null und alle anderen Zustände beliebig initialisiert. Anschließend wird die Bellman-Gleichung 2.8 als Update-Regel verwendet. Entsprechend gilt für alle $s \in \mathcal{S}$:

$$v_{k+1}(s) = \sum_{a \in A} \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] . \quad (2.11)$$

Mit fortschreitender Wiederholung, d. h. $k \rightarrow \infty$, werden die Values gegen ihre tatsächlichen Werte konvergieren. Um iterativ die nächste Value-Function v_{k+1} aus der vorherigen v_k bestimmen zu können, wird die Update-Regel auf jeden Zustand angewandt und somit der zugehörige neue Value berechnet.

Der zugehörige Pseudocode in den Zeilen 1 bis 10 von Algorithmus 1 terminiert durch das Abbruchkriterium $\Delta < \theta$. Unterschreitet die Differenz zwischen vorherigem und aktuellen Value des Endzustandes einen kleinen, positiven Grenzwert θ , wird das Ergebnis als ausreichend gut angesehen und die Suche beendet. Prinzipiell werden Value-Functions berechnet, um infolgedessen bessere Policies zu finden. *Policy Improvement* bezeichnet den Prozess der Policy-Verbesserung, bei welchem in einem Zustand s eine Aktion a abweichend der aktuellen Policy π gewählt wird. Überschreitet der entsprechende State-Action-Value $q_\pi(s,a)$ nach Gleichung 2.12 den State-Value $v_\pi(s)$, ist die Wahl der Aktion abweichend der aktuellen Policy besser. Der Q-Value für ein State-Action-Paar berechnet sich dabei nach:

$$\begin{aligned} q_\pi(s,a) &\doteq \mathbb{E}_\pi [R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = a] \\ &= \sum_{s'} p(s',r|s,a) [r + \gamma v_\pi(s')] . \end{aligned} \quad (2.12)$$

Policy Iteration evaluiert und verbessert die Policy abwechselnd, wie Abbildung 2.5 verdeutlicht.

$$\pi_0 \xrightarrow{\text{E}} v_{\pi_0} \xrightarrow{\text{I}} \pi_1 \xrightarrow{\text{E}} v_{\pi_1} \xrightarrow{\text{I}} \dots \xrightarrow{\text{E}} \pi_* \xrightarrow{\text{I}} v_{\pi_*} = v_*$$

Abbildung 2.5: Ablauf von Policy Iteration nach [4]

Der Agent startet mit einer zufälligen Policy und nutzt *Policy Evaluation* für ein Update der Value-Function. Anschließend wird mithilfe von *Policy Improvement* überprüft, ob eine bessere Policy existiert und möglicherweise diese übernommen. Falls keine bessere Policy gefunden werden kann, handelt es sich bei der aktuellen Policy um eine optimale Policy und entsprechend ebenfalls um eine optimale Value-Function, weshalb der Algorithmus 1 beendet wird.

Im Evaluationsschritt wird der Zustandsraum häufig mehrfach durchlaufen bis die Value-Funktion gegen v_π konvergiert. Oftmals reichen aber schon wenige Iterationen, damit im *Policy Improvement*-Schritt eine bessere Policy gefunden werden kann. *Value Iteration* ist eine Abwandlung von *Policy Iteration*, bei welcher der *Policy Evaluation*-Schritt nach ei-

Algorithmus 1 Policy Iteration zur Approximation von $\pi \approx \pi_*$

- 1: $V(s) \in \mathbb{R}$ für alle Zustände zufällig initialisieren
- 2: $\pi(s) \in \mathcal{A}(s)$ für alle Zustände zufällig initialisieren

Policy Evaluation

```

3: do
4:    $\Delta \leftarrow 0$ 
5:   for jeden  $s \in \mathcal{S}$  do
6:      $v \leftarrow V(s)$ 
7:      $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$ 
8:      $\Delta \leftarrow \max(\Delta, |v - V(s)|)$ 
9:   end for
10:  while  $\Delta < \theta$ 

```

Policy Improvement

```

11: stable  $\leftarrow$  true
12: for jeden  $s \in \mathcal{S}$  do
13:    $a_{\text{prev}} \leftarrow \pi(s)$ 
14:    $\pi(s) \leftarrow \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$ 
15:   if  $a_{\text{prev}} \neq \pi(s)$  then
16:     stable  $\leftarrow$  false
17:   end if
18: end for
19: if stable then
20:   return  $V \approx v_*, \pi \approx \pi_*$ 
21: else
22:   zurück zu Policy Evaluation
23: end if

```

ner Iteration, d. h. nach einem Durchlauf durch alle Zustände, beendet wird. Das neue Update für die Value-Function vereint den Evaluations- und Verbesserungsschritt in einer Gleichung:

$$\begin{aligned}
v_{k+1}(s) &\doteq \max_a \mathbb{E}[R_{t+1} + \gamma v_k(S_{t+1}) | S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma v_k(s')] .
\end{aligned} \tag{2.13}$$

Monte-Carlo Methoden

Bei MC-Methoden sind die Schätzungen der einzelnen Values der entsprechenden Zustände nicht von anderen Values abhängig, weshalb diese Verfahren für große Zustandsräume weniger geeignet sind. Im Gegensatz zu DP haben diese Verfahren jedoch den Vorteil, dass mit ihnen der Value eines einzelnen States bzw. eines Teils des Zustandsraum betrachtet werden kann, ohne alle Zustände approximieren zu müssen.

Ohne bekanntes Umweltmodell kann es vorteilhaft sein die State-Action-Values q anstatt der State-Values v zu betrachten, da entsprechend nicht bekannt ist, welche Nachfolgezustände von einem Startzustand erreichbar sind. Wird für jede Aktion ausgehend von einem Zustand ein separater Mittelwert des Returns gebildet, konvergieren diese gegen

die jeweiligen State-Action-Values q . Abbildung 2.6 verdeutlicht den Unterschied zwischen *Policy Iteration* in der dynamischen Programmierung und der Monte-Carlo Version. Beide folgen dabei der Grundidee von Generalized Policy Iteration (GPI), welche allgemein das Interagieren eines *Policy Improvement*- und eines *Policy Evaluation*-Prozesses bezeichnet.

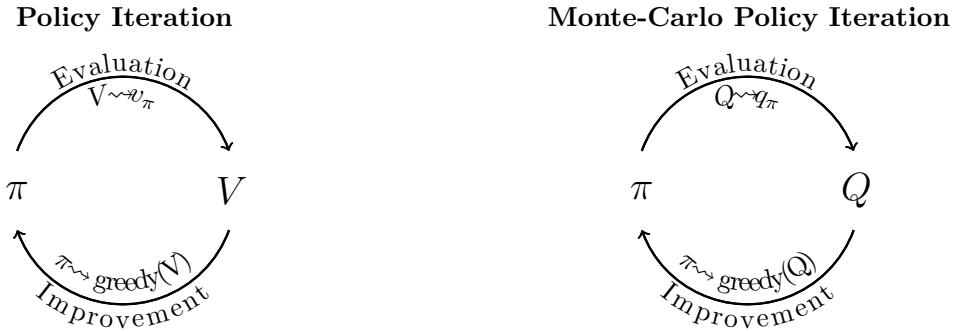


Abbildung 2.6: Unterschied zwischen Policy Iteration und Monte-Carlo Policy Iteration nach [4]

Durch abwechselnde Evaluations- und Verbesserungsschritte wird mithilfe von *MC Policy Iteration* letztendlich eine optimale Policy und eine optimale Q- bzw. State-Action-Value Function ermittelt. Der Evaluationsschritt ist identisch mit dem der *DP Policy Iteration*. *Policy Improvement* wird durch „Gierigkeit“ der Policy im Bezug auf die Q-Values erreicht, d. h., dass in jedem Zustand die Aktion mit dem höchsten Q-Wert gewählt wird:

$$\pi(s) \doteq \arg \max_a q(s, a). \quad (2.14)$$

Temporal Difference Learning

Im Gegensatz zu MC-Methoden nutzen TD-Methoden für Updates nicht den Return der kompletten Episode, sondern lediglich den beobachteten Reward im nächsten Zeitschritt. Das einfachste TD-Verfahren TD(0) nutzt für das Update der Value-Function lediglich den aktuellen Value, den Value des Nachfolgezustandes und den durch die Transition erhaltenen Reward:

$$V(S_{t+1}) = V(S_t) + \alpha [R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]. \quad (2.15)$$

Der Term $R_{t+1} + \gamma V(S_{t+1})$ wird als TD-Target bezeichnet, der Teil der Gleichung 2.15 in eckigen Klammern entspricht dem TD-Fehler δ_t . Dieser gibt die Differenz zwischen der aktuellen Schätzung des Returns ausgehend von Zustand S_t und der verbesserten Schätzung $R_{t+1} + \gamma V(S_{t+1})$ an.

Q-Learning [35] ist einer der ersten TD-Algorithmen, bei welchem die optimale Q- bzw. State-Action-Funktion q_* approximiert wird. Der Algorithmus wird durch das folgende Update definiert:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \quad (2.16)$$

Bei *Q-Learning* nutzt der Agent für die Berechnung des Targets $R_{t+1} + \gamma \max_a Q(S_{t+1}, a)$ eine andere Policy als für die Auswahl der ausführten Aktion. Da für das Update die Aktion mit dem höchsten Q-Wert gewählt wird, handelt es sich um ein gieriges Verhalten in

Bezug auf $Q(s, a)$. Um weitere Zustands-Aktions-Paare zu besuchen, ist für die tatsächliche Aktionsauswahl aber beispielsweise eine ϵ -greedy Policy vorteilhafter, bei welcher abhängig vom Parameters ϵ zufällige Aktionen gewählt werden. Das Update nach Gleichung 2.16 approximiert q_* folglich unabhängig der verfolgten Policy. Unter der Annahme, dass alle State-Aktion-Paare aktualisiert werden, ist die Konvergenz von Q gegen q_* in [36] bewiesen. Der prinzipielle Ablauf von *Q-Learning* ist Algorithmus 2 zu entnehmen [4, 37].

Algorithmus 2 Q-Learning zur Approximation von $\pi \approx \pi_*$ nach [4]

```

 $Q(s, a)$  für alle Zustand-Aktion-Paare zufällig initialisieren
 $Q(s, a)$  für alle Endzustand-Aktion-Paare mit 0 initialisieren
for jede Episode do
     $S$  initialisieren
    do
        Aktion  $A$  in Zustand  $S$  nach der Policy abgeleitet aus  $Q$  (z.B.  $\epsilon$ -greedy) wählen
        Aktion  $A$  ausführen,  $R$  und  $S'$  beobachten
         $Q(S, A) \leftarrow Q(S, A) + \alpha[R + \gamma \max_a Q(S', a) - Q(S, A)]$ 
         $S \leftarrow S'$ 
    while  $S$  kein Endzustand ist
end for

```

2.1.4 Policy-based Verfahren

Die im vorherigen Kapitel vorgestellten Methoden approximieren vorwiegend die Value-Function. Aus dieser wird anschließend die Handlungsweise des Agenten abgeleitet, indem beispielsweise immer die Aktion mit dem höchsten Q-Value gewählt wird. Da in diesen Verfahren die Policy nur implizit anhand der ermittelten Value-Function abgeleitet wird, spricht man hierbei von sogenannten value-based Methoden. Neben RL-Verfahren dieser Gruppe existieren Methoden, die direkt eine parametrisierte Policy erlernen. Bei policy-based RL-Verfahren optimiert der Agent durch die Interaktion mit der Umwelt direkt eine parametrisierte Policy. Dabei kann eine Value-Function zum Erlernen der Policy-Parameter verwendet werden. Zum Lernen der Parameter kann der Gradient eines Güte-Maßes $J(\theta)$ bezogen auf die Policy-Parameter θ genutzt werden. Für episodische Aufgaben kann dieses durch

$$J(\theta) \doteq v_\pi(s_0) \quad (2.17)$$

beschrieben werden, wobei der Value $v_\pi(s_0)$ des Episodenstartzustands s_0 nach Gleichung 2.6 berechnet wird.

Das vorliegende Optimierungsproblem kann mithilfe verschiedener Ansätze gelöst werden. Nachfolgend werden lediglich Verfahren basierend auf dem Gradienten für die Optimierung der Policy betrachtet. Policy Gradient (PG)-Methoden suchen das lokale Maximum der Zielfunktion $J(\theta)$, indem die Parameter θ nach der folgenden Gleichung angepasst werden [38]:

$$\Delta\theta = \alpha \nabla_\theta J(\theta), \quad (2.18)$$

wobei $\nabla_\theta J(\theta)$ der namensgebende Policy Gradient und α eine Lernrate ist. Der Policy Gradient ist ein Vektor, welcher alle partiellen Ableitungen der Zielfunktion J nach den einzelnen Policy-Parametern enthält [38].

Es ist schwierig zu gewährleisten, dass die Policy-Parameteränderungen zur Verbesserung der Gütfunktion führen, weil diese neben der Aktionsauswahl auch von der Verteilung der

besuchten Zustände abhängt. Da der Einfluss der Policy auf die Zustandsverteilung von der Transitionsfunktion abhängig und somit dem Agenten unbekannt ist, wird die Schätzung des Policy Gradienten erschwert. Das Policy Gradient Theorem beschreibt den Gradienten der Güte-Funktion nach den Policy-Parametern ohne dabei die Ableitung der Verteilung der besuchten Zustände zu benötigen. Für den episodischen Fall gilt

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a|s, \theta), \quad (2.19)$$

wobei μ die On-Policy-Verteilung ist, welche die „verbrachte Zeit“ in einem Zustand ins Verhältnis zu der Gesamtzeit setzt und somit die Auftrittshäufigkeit des Zustandes festlegt. Der Beweis für die Gültigkeit des Theorems wird an dieser Stelle nicht aufgeführt, ist jedoch in [4] zu finden.

Folgt der Agent der Policy π kann die Gewichtung über die Zustandsverteilung durch den Erwartungswert ersetzt werden, wodurch sich Gleichung 2.19 zu

$$\nabla J(\theta) = \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta) \right] \quad (2.20)$$

verändert. Wird dieser Gradient für ein Policy-Parameterupdate genutzt, handelt es sich um einen all-actions Algorithmus, da alle möglichen Aktionen berücksichtigt werden. Gleichung 2.20 kann um den Term $\pi(a|S_t, \theta)$ erweitert werden, welcher angibt, mit welcher Wahrscheinlichkeit eine Aktion im Zustand S_t gewählt wird. Wird lediglich die im Zustand S_t tatsächlich gewählte Aktion A_t betrachtet, ergibt sich der Policy Gradient, der für das Parameter-Update des *REINFORCE*-Algorithmus verwendet wird.

$$\begin{aligned} \nabla J(\theta) &= \mathbb{E}_\pi \left[\sum_a \pi(a|S_t, \theta) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t, \theta)}{\pi(a|S_t, \theta)} \right] \\ &= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\ &= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t, \theta)}{\pi(A_t|S_t, \theta)} \right] \\ &= \mathbb{E}_\pi [G_t \nabla \ln \pi(A_t|S_t, \theta)] \end{aligned} \quad (2.21)$$

Somit ist *REINFORCE* eine Monte-Carlo Methode, da der Return der gesamten Episode G_t berücksichtigt wird. Da möglicherweise in einigen Zuständen mehrere Aktionen einen hohen Q-Wert haben, ist das Zufügen einer sogenannten Baseline hilfreich, um das Lernen zu beschleunigen. Wird beispielsweise der State-Value verwendet, wird die Verbesserung durch die Wahl der Aktion gegenüber diesem betrachtet. Der Value des Zustandes entspricht dabei der gewichteten Summe aller Q-Werte in diesem, weshalb dieser nicht von der aktuell gewählten Aktion abhängig ist. Infolgedessen eignet sich die Schätzung $\hat{v}(S_t, w)$ als Baseline. Diese Schätzung kann mithilfe von value-based Methoden erlernt werden. Algorithmus 3 zeigt die Vorgehensweise unter Berücksichtigung der mittels des Faktor γ gewichteten zukünftigen Rewards.

Algorithmus 3 REINFORCE with Baseline nach [4]

```

 $\theta$  zufällig initialisieren
 $w$  zufällig initialisieren
for jede Episode do
    Trajektorie  $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$  entsprechend Policy  $\pi(\cdot | \cdot, \theta)$  erzeugen
    for  $t = 0, 1, \dots, T-1$  do
         $G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k$ 
         $\delta \leftarrow G - \hat{v}(S_t, w)$ 
         $w \leftarrow w + \alpha^w \delta \nabla \hat{v}(S_t, w)$ 
         $\theta \leftarrow \theta + \alpha^\theta \gamma^t \delta \nabla \ln \pi(A_t | S_t, \theta)$ 
    end for
end for

```

Neben der Einteilung in value- und policy-based können alle RL-Verfahren in On- bzw. Off-Policy Methoden eingeteilt werden.

Bei On-Policy Algorithmen lernt der Agent basierend auf Erfahrungstupeln, welche mit der aktuellen Policy gesammelt wurden. Entsprechend existiert nur eine Policy, mit welcher die Parameterupdates berechnet und die Handlungen des Agenten ausgewählt werden. Ein Beispiel für Off-Policy Algorithmen ist *Q-Learning*, da verschiedene Policies verwendet werden. Der Agent wählt Aktionen basierend auf einer Policy, wohingegen die Daten, die zum Anpassen der Policy oder der Value-Function verwendet werden, von einer anderen Policy stammen können. Abbildung 2.7 visualisiert die bereits vorgestellte Einteilung von Reinforcement Learning Verfahren.

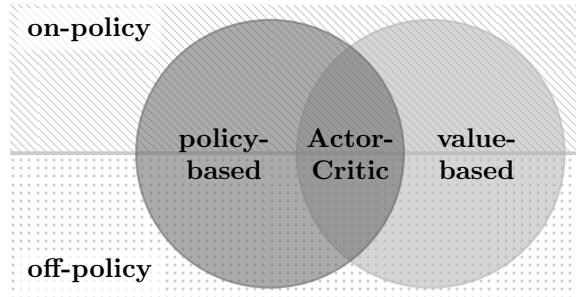


Abbildung 2.7: Einteilung von RL-Verfahren angelehnt an [6]

Methoden in der Schnittmenge von value- und policy-based approximieren sowohl eine Value-Function als auch eine Policy und werden unter dem Begriff Actor-Critic-Methoden zusammengefasst. Dabei wird mithilfe einer parametrisierten Funktion die Policy, der sogenannte Actor, abgebildet. Mithilfe eines zweiten Parametersets wird die Value-Function, die sogenannte Critic, erlernt. Der Nachteil von policy-based Methoden, wie beispielsweise *REINFORCE*, ist, dass aufgrund der Verwendung des Returns die Parameter erst am Ende einer Episode aktualisiert werden können. Actor-Critic-Methoden lösen dieses Problem, indem mithilfe der Critic der aktuelle State- bzw. State-Action-Value approximiert und stellvertretend für den Return verwendet wird. Infolgedessen sind Updates nach jeden Zeitschritt möglich. Im Allgemeinen wird die Critic verwendet, um den Actor zu verbessern, durch die Interaktion der beiden Funktionsapproximatoren verbessern diese sich jedoch

gegenseitig. Actor-Critic-Algorithmen vereinen entsprechend value-based RL in der Critic und policy-based RL in dem Actor.

2.2 Künstliche neuronale Netze

Künstliche neuronale Netze, häufig auch nur Neuronale Netze (NN) genannt, wurden mit der Motivation der Nachbildung des menschlichen Gehirns entwickelt. Durch Adaption bekannter Bestandteile des menschlichen Gehirns entstehen Netzwerke, welche Lösungen für unterschiedlichste Probleme erlernen können.

2.2.1 Aufbau von künstlichen neuronalen Netzen

Künstliche neuronale Netze bestehen aus verschiedenen Schichten, welche mehrere künstliche Neuronen miteinander verknüpfen.

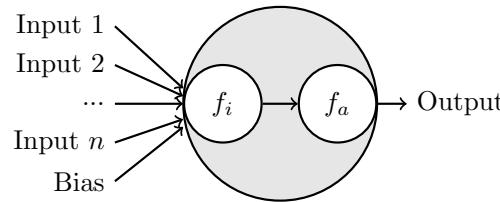


Abbildung 2.8: Künstliches Neuron

Abbildung 2.8 zeigt den prinzipiellen Aufbau eines künstlichen Neurons. Die n verschiedenen Eingaben in ein Neuron, welche häufig als Features bezeichnet werden, werden mithilfe der Integrationsfunktion f_i verarbeitet, bevor diese an die Aktivierungsfunktion f_a übergeben werden. Für die Zusammenfassung der Neuroneingaben wird dabei überwiegend die Summenfunktion verwendet. Mithilfe des Bias kann ein Schwellenwert vorgegeben werden, der überschritten werden muss, damit das Neuron eine positive Ausgabe liefert. Die endgültige Aktivierung eines Neurons wird mithilfe der Aktivierungsfunktion berechnet. Eine Auswahl an Aktivierungsfunktionen ist Abbildung 2.9 zu entnehmen.

Die Identitätsfunktion oder auch lineare Funktion (Abbildung 2.9a), die die einfachste Aktivierungsfunktion darstellt, da sie die Eingänge in ein Neuron nicht weiter manipuliert, wird vorwiegend in der Ausgabeschicht von mehrschichtigen KNN verwendet. In anderen Schichten werden überwiegend nichtlineare Funktionen genutzt, da durch deren Kombination beliebig komplexe Funktionen repräsentiert werden können. Die Sigmoidfunktion (Abbildung 2.9b) beschränkt den Eingabewert auf einen Bereich zwischen 0 und 1. Das Problem der nicht gegebenen Nullzentrierung wird durch den Tangenshyperbolicus (Abbildung 2.9c) gelöst, der die Eingabewerte auf einen Bereich zwischen -1 und 1 beschränkt. Das im maschinellen Lernen bekannte Problem der „verschwindenden Gradienten“ existiert jedoch bei beiden bereits genannten nichtlinearen Aktivierungsfunktionen und tritt auf, wenn aufgrund der Sättigung des Neurons in bestimmten Bereich der Gradient gegen null strebt. Um dieses zu umgehen, ist aktuell Rectifier Linear Unit (ReLU) (Abbildung 2.9d) eine weit verbreitete Aktivierungsfunktion. Durch die Linearfunktion im positiven Bereich „verschwinden“ die Gradienten im positiven Eingabebereich nicht [39]. Softplus (Abbildung 2.9e) wird häufig ebenfalls als Smooth-ReLU bezeichnet, da ein glatterer Übergang vom negativen in den positiven Eingabebereich vorliegt, wobei jedoch das bei ReLU auftretende Problem nicht gelöst wird. Leaky ReLU und Exponential Linear Unit (ELU) (Abbildung

2.9f) sind Aktivierungsfunktionen, welche versuchen das Problem der verschwindenden Gradienten auch im negativen Eingabebereich zu lösen. Leaky ReLU nutzt dabei im negativen Eingabebereich eine lineare Funktion mit geringer Steigung, ELU eine verschobene Exponentialfunktion.

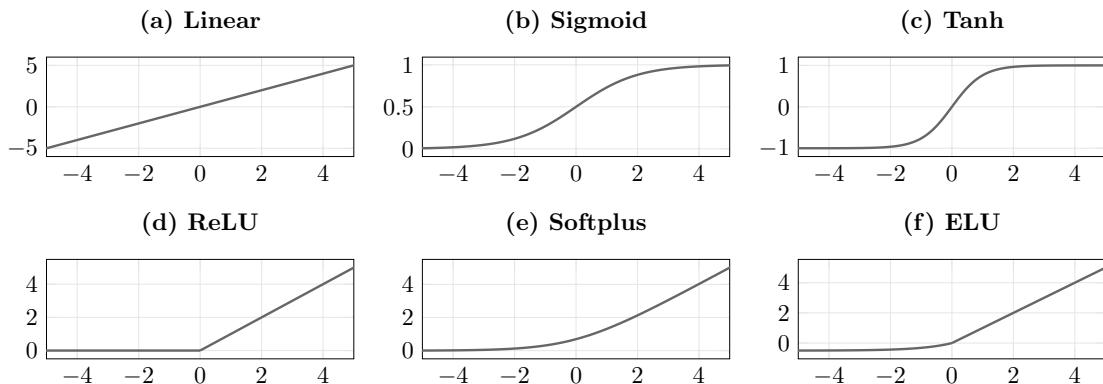


Abbildung 2.9: Aktivierungsfunktionen

Multilayer-Perzeptren (MLP) beschreiben neuronale Netzwerke, die aus mehreren Neuronen in verschiedenen Schichten bestehen. Dabei wird in Eingabe-, Ausgabe- und versteckte Schichten (bzw. input layer, output layer und hidden layer) unterschieden. Abbildung 2.10 zeigt ein KNN, wobei jeder der n Eingänge durch ein Input-Neuron und jeder der m Ausgänge durch ein Output-Neuron repräsentiert wird. Jedes Neuron jeder Schicht ist mit allen Neuronen der nachfolgenden Schicht verbunden. Die dadurch entstehende Architektur wird als fully-connected bezeichnet. Da es sich bei den Verbindungen ausschließlich um vorwärtsgerichtete, von einer Schicht zur Nachfolgeschicht laufende Kanten handelt, wird das Netz als Feed Forward Netz bezeichnet. Neben dieser Netztopologie existieren weitere, wie beispielsweise rekurrente Netze, auf welche in dieser Arbeit nicht genauer eingegangen wird.

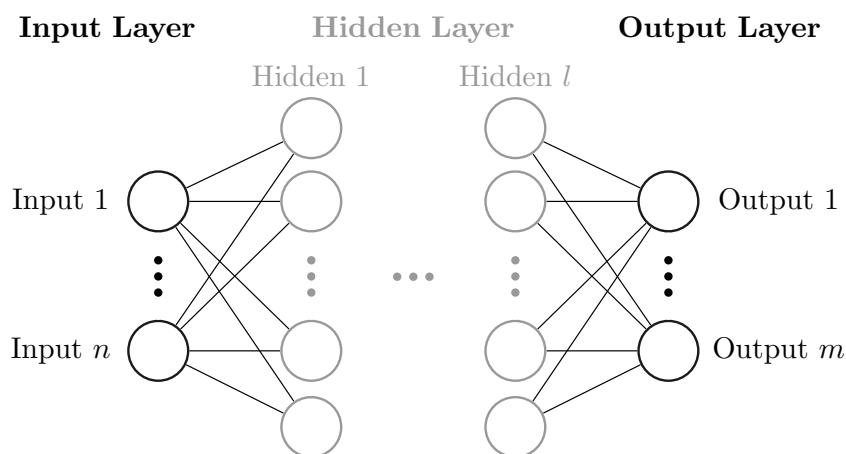


Abbildung 2.10: MLP mit n Eingängen, m Ausgängen und l verdeckten Schichten

Die Eingaben in ein Neuron werden mithilfe von Gewichtungsfaktoren für die jeweiligen

Verbindungen versehen, um den entsprechenden Einfluss auf die Aktivierung des Neurons zu definieren. Die Ausgabe o_j eines Neurons j berechnet sich nach:

$$o_j = f_a(w_{1j}x_1 + w_{2j}x_2 + \dots + w_{nj}x_n + w_{bj}b), \quad (2.22)$$

wobei x die Inputs, b den Bias, f_a die Aktivierungsfunktion und w die Gewichtungen bezeichnen.

Die kontinuierliche Anpassung dieser Gewichte, um mithilfe des Netzwerkes das vorgegebene Problem zu lösen, wird als Lernen bezeichnet.

2.2.2 Lernen in künstlichen neuronalen Netzen

Ziel des Lernens in künstlichen neuronalen Netzen ist vorwiegend das Minimieren einer Kostenfunktion $J(w)$, welche von den Parametern w des KNN abhängig ist. Häufig handelt es sich hierbei um einen Fehler, welcher reduziert werden soll.

Vor dem Training des KNNs sollten die Eingabedaten und die Netzwerke vorbereitet werden, um bessere und schnellere Lernerfolge zu erzielen. Bei bekannter Trainingsmenge kann der Mittelwert des Datensatzes subtrahiert werden, um eine Nullzentrierung der Werte zu erreichen. Mithilfe der Standardabweichung kann zusätzlich eine Normalisierung erfolgen [40]. Bei unbekannten Trainingsmengen können alternativ beispielsweise die physikalischen Grenzen eines bekannten Systems verwendet werden, da diese den maximalen Wertebereich der Eingangsgröße einschränken. Durch die Vorverarbeitung der Daten wird garantiert, dass keine Eingangsgröße aufgrund seiner Skalierung gegenüber den anderen Messgrößen bereits vor dem Training stärker gewichtet wird.

Ebenso relevant wie die Vorbereitung der Eingangsdaten ist die Initialisierung der Gewichte im KNN. Werden alle Netzparameter gleich initialisiert, entstehen Symmetrien, die nicht gewünscht sind. Beispielsweise sind aufgrund der auftretenden Ähnlichkeiten die Änderungen der Gewichte identisch, da jedes Neuron bei der Rückpropagierung des Netzfehlers identisch reagiert. Um diese Symmetrien zu vermeiden, werden vorwiegend kleine zufällige Werte als Startgewichte verwendet, welche für kleine KNN gute Ergebnisse zeigen [40]. In tieferen Netzen ist diese Initialisierung schwierig, da aufgrund der Verkettung der Aktivierungsfunktionen die Neuronausgaben ebenfalls kleine Werte haben werden. In tiefen KNN entstehen dadurch schichtweise immer niedrigere Aktivierungen, wodurch die Gradienten beim Rückpropagieren nahezu verschwinden. Um dies zu vermeiden, existieren fortgeschrittenere Verfahren zur Initialisierung von Netzwerkgewichten, wie beispielsweise die Xavier-Initialisierung [41] oder die He-Initialisierung [42]. Bei letzterer werden die initialen Gewichte aus einer Normalverteilung mit dem Mittelwert 0 und der Standardabweichung

$$\sigma = \sqrt{\frac{2}{n}} \quad (2.23)$$

gesampelt, wobei n der Anzahl der Eingänge in der jeweiligen Schicht entspricht.

In jedem Lernschritt werden drei Phasen durchlaufen: die Berechnung der Netzausgabe, die Ermittlung und anschließende Rückpropagierung des Fehlers durch das Netz und abschließend die Anpassung der Gewichtsmatrizen [43, 44].

Im ersten Schritt, der Feed-Forward-Berechnung, werden die Eingaben \vec{x} in das Netz gegeben und unter Berücksichtigung der Gewichtungen die Netzausgabe sowie die Aktivierung von jedem Neuron berechnet. Die Aktivierungen aller Neuronen einer Schicht werden in einem Vektor zusammengefasst. Mit der Gewichtsmatrix \mathbf{w}_j , welche um den Bias erweitert

wurde, ergibt sich für die Neuronaktivierungen

$$\overrightarrow{o_j} = f_a(\mathbf{w}_j \overrightarrow{o_{j-1}}), \quad (2.24)$$

wobei $\overrightarrow{o_{j-1}} = \overrightarrow{x}$ gilt, wenn es sich um die erste Schicht handelt. Die Ausgaben einer Schicht werden entsprechend als Eingaben in der nächsten Schicht verwendet. Für die spätere Berechnung der einzelnen Gewichtsupdates werden die Ableitungen der Neuronen pro Schicht in einer Diagonalmatrix \mathbf{d} gespeichert. Das am Netzausgang $\overrightarrow{o_{out}}$ erhaltene Ergebnis wird beim Supervised Learning mit dem gewünschten Zielwert \overrightarrow{t} verglichen. Mithilfe einer Fehlerfunktion J , beispielsweise dem mittleren quadratischen Fehler (MSE), wird basierend auf der prädizierten und der Sollausgabe der Fehler berechnet. Dabei gilt

$$J = \frac{1}{2} (\overrightarrow{o_j} - \overrightarrow{t})^2. \quad (2.25)$$

In der zweiten Phase eines Lernschritts wird der Fehler vom Ausgang zum Eingang durch das Netz propagiert, um den Anteil jedes Neurons am entstandenen Fehler zu berechnen. Für jedes Neuron wird unter Zuhilfenahme der gespeicherten Ableitung und des gewichteten Fehlers der nachfolgenden Schicht der neuronabhängige, rückpropagierte Fehler δ ermittelt, für welchen

$$\begin{aligned} \overrightarrow{\delta_{out}} &= \mathbf{d}_{out} (\overrightarrow{o_j} - \overrightarrow{t}) \\ &= \mathbf{d}_{out} \overrightarrow{e} \end{aligned} \quad (2.26)$$

in der letzten und

$$\overrightarrow{\delta_j} = \mathbf{d}_j \mathbf{w}_{j+1} \overrightarrow{\delta_{j+1}} \quad (2.27)$$

in den vorherigen Schichten gilt. Durch Multiplikation mit den Eingaben in die entsprechende Schicht wird der Gradient berechnet. Dieses Vorgehen wird als Backpropagation bezeichnet. Backpropagation ist eine effiziente Technik, um die benötigten Gradienten für die Gewichtsaktualisierungen zu berechnen. Dabei wird die partielle Ableitung der Fehlerfunktion nach dem jeweiligen Gewicht mithilfe der Kettenregel bestimmt. Der dritte Schritt, das Aktualisieren der Parameter, unterscheidet sich je nach verwendetem Lernverfahren. Es existiert eine Vielzahl an Methoden für das Training von neuronalen Netzen. Nachfolgend wird neben dem Standard-Gradientenabstiegsverfahren Stochastic Gradient Descent (SGD) lediglich auf Adam eingegangen, da letzteres im Rahmen dieser Arbeit verwendet wird.

Stochastic Gradient Descent ist ein Optimierungsverfahren erster Ordnung und nutzt entsprechend lediglich den Gradienten g der Kostenfunktion nach den Netzwerkgewichten, welcher mithilfe der Lernrate η skaliert wird. Die Lernrate ist ein frei wählbarer Hyperparameter, der die Schrittweite entlang des Gradienten festlegt. Das neue Gewicht ergibt sich aufgrund des Gradientenabstiegs durch Subtraktion des mittels der Lernrate gewichteten Gradienten von dem vorherigen Gewicht, da die Fehlerfunktion minimiert werden soll:

$$w_t = w_{t-1} - \eta g_t. \quad (2.28)$$

Adam [45] ist ein Optimierungsverfahren mit adaptiver Lernrate, bei welchem basierend auf den ersten partiellen Ableitungen der Kostenfunktion nach den Netzwerkgewichten gewichtsspezifische Lernraten berechnet werden. Die beiden dabei verwendeten Momente

m und v berechnen sich nach folgenden Gleichungen:

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t, \quad (2.29)$$

$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2, \quad (2.30)$$

wobei m und v laufende Mittelwerte des Gradienten g bzw. des quadratischen Gradienten g^2 sind. Entsprechend repräsentiert m den Mittelwert und v die umzentrierte Varianz des Gradienten. β_1 und β_2 sind frei wählbare Hyperparameter, für welche häufig 0.9 bzw. 0.999 genutzt wird [45]. Durch die Initialisierung der beiden Momente mit null entstehen verzerrte Schätzungen, weshalb die Werte korrigiert werden müssen:

$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad (2.31)$$

$$\hat{v}_t = \frac{v_t}{1 - \beta_2^t}. \quad (2.32)$$

Das neue Gewicht ergibt sich anschließend nach:

$$w_t = w_{t-1} - \eta \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \varepsilon}. \quad (2.33)$$

Beim Lernen in künstlichen neuronalen Netzen kann zwischen Online- und Batch-Training unterschieden werden. Beim Online-Lernen werden die Netzparameter nach jedem präsentierten Datenpaar aktualisiert. Beim Batch-Training werden mehrere Datenpaare in einer Gruppe, einem sogenannten Batch, zusammengefasst. Die berechneten Fehler der einzelnen Datenpaare werden aufsummiert, um die Gewichte anzupassen, wodurch der Abstieg in der „echten“ Gradientenrichtung erfolgt. Wird beim Batch-Training der gesamte Trainingsdatensatz verwendet, muss dieser bereits zu Beginn bekannt sein und kann nicht erweitert werden. Da die Eingabedaten in einem Batch voneinander unabhängig sind, kann die Berechnung der Netzausgaben parallelisiert werden, wodurch die Berechnungszeit verkürzt werden kann. In der Praxis werden häufig sogenannte Mini-Batches verwendet, die die Vorteile beider Trainingsarten verbinden. Durch Verwendung mehrerer Datenpaare für einen Updateschritt wird tendenziell eher der „echten“ Gradientenrichtung gefolgt, ohne dass die komplette Datenbasis vor Trainingsbeginn bekannt sein muss. Außerdem können Rechenzeitverkürzungen aufgrund der Parallelisierung ausgenutzt werden.

Zur Verbesserung des Lernerfolgs existieren verschiedene Ansätze, die mit den unterschiedlichen Lernverfahren kombiniert werden können. Nachfolgend wird das Grundprinzip von zwei dieser Verfahren, Batch-Normalization und Dropout, erläutert.

Batch-Normalization [46] ist ein Verfahren, welches verwendet wird, um auch für die Eingaben in die versteckten Schichten eines KNN eine passende Skalierung sicherzustellen. Prinzipiell wird verlangt, dass die Varianz in jedem Layer optimalerweise der Gaußverteilung entspricht. Der Mittelwert und die Standardabweichung eines Datenbatches werden für jedes Feature x berechnet und zur Normalisierung verwendet, wodurch sich der neue Wert \hat{x} ergibt:

$$\hat{x} = \frac{x - \mathbb{E}[x]}{\sqrt{\text{Var}[x]}}. \quad (2.34)$$

Problematischerweise ist nicht für jede Schicht bekannt, ob gaußverteilte Eingangssignale optimal sind. Durch Ergänzung zweier Parameter γ und β erhält das KNN zwei weitere

anpassbare Parameter, welche die Features nach

$$\hat{x} = \gamma \hat{x} + \beta \quad (2.35)$$

beeinflussen können. Durch γ kann folglich die Streuung und durch β der Mittelwert der Gaußverteilung für die Skalierung der Daten erlernt werden. Batch-Normalisierung verbessert den Gradientenfluss durch das Netzwerk, da durch Anpassung der beiden zusätzlichen Hyperparameter verhindert wird, dass alle Neuronen eine Aktivierung von null haben. Folglich werden somit höhere Lernraten und damit verbunden schnelleres Lernen ermöglicht [40].

Dropout ist eine Regulierungsmethode für das Trainieren von KNN, welche Overfitting vermeiden soll. Mit einer bestimmten Wahrscheinlichkeit p wird ein Gewicht auf 0 gesetzt und entsprechend für die Berechnung der Netzausgabe in diesem Lernschritt nicht berücksichtigt, wodurch das Auswendiglernen von Mustern erschwert wird. Beim Testen und Verwenden des trainierten neuronalen Netzes werden alle gelernten Gewichtungen verwendet [47, 48].

Künstliche neuronale Netze können im Reinforcement Learning zur Abbildung der Policy und der Value-Function eingesetzt werden. Policy-based Verfahren sind dem Supervised Learning weniger ähnlich, da kein Fehler zwischen dem Ist- und dem Zielwert gebildet und für die Parameterupdates verwendet wird. Stattdessen wird direkt anhand der Gütekoeffizienten ein Gradient berechnet, dessen Richtung bei den Parameterupdates gefolgt wird. Value-based Reinforcement Learning ist Supervised Learning sehr ähnlich, da basierend auf den Prädiktionen des KNN und der erhaltenen Rewards ein Zielwert ermittelt werden kann. Dieser kann entsprechend zur Bildung des mittleren quadratischen Fehlers oder einer anderen Fehlerfunktion und folglich für das Updaten der Netzwerkparameter verwendet werden.

3 Algorithmen

Dieses Kapitel befasst sich mit den verwendeten Algorithmen, welche zur Gruppe der Actor-Critic-Verfahren gehören.

In jeder der gewählten Methoden wird sowohl eine Policy als auch eine Value-Function erlernt, zu deren Abbildung künstliche neuronale Netze verwendet werden. Diese Funktionsapproximatoren sind besonders für die vorhandenen kontinuierlichen und somit häufig großen Aktions- und Zustandsräume geeignet. Der gewählte Aufbau dieser Netzwerke wird in Kapitel 4.1 genauer erläutert.

Zur Implementierung der Algorithmen wird PyTorch verwendet. PyTorch ist ein Python-Package, welches zum einen zum Ausführen von Tensorberechnungen auf Grafikprozessoren (engl. graphics processing units, GPUs) und zum anderen zum Erstellen und Trainieren von tiefen neuronalen Netzen verwendet werden kann. Im Rahmen dieser Arbeit wird PyTorch für die Implementierung der RL-Agenten und zum Training dieser mittels der verschiedenen Algorithmen genutzt. Im Gegensatz zu anderen Frameworks wie beispielsweise TensorFlow wird der Graph zur Berechnung dynamisch bei der Verwendung der jeweiligen Knotenpunkte erzeugt und nicht im Voraus komplett definiert. PyTorch kann somit als define-by-run Framework bezeichnet werden, bei welchem der aus anderen Frameworks bekannte Kompilierungsschritt entfällt. Infolgedessen stellt es ein benutzerfreundliches Tool dar, welches schnell zum Entwickeln und Testen neuer Ideen im Bereich des Deep Learnings angewendet werden kann [49, 50].

Im weiteren Verlauf des Kapitels wird zuerst die Auswahl der Algorithmen begründet. Anschließend erfolgt eine Erläuterung der mathematischen Grundlagen sowie der Besonderheiten der jeweiligen Algorithmen.

3.1 Auswahl der Algorithmen

Im Rahmen dieser Arbeit sollen Algorithmen zum Erlernen eines zusätzlichen Anteils an einem bestehenden Trajektorienfolgeregelungskonzeptes untersucht werden. Die RL-Methoden müssen entsprechend für kontinuierliche Zustands- und Aktionsräume geeignet sein. Nachfolgend sollen Vor- und Nachteile der Algorithmen vorgestellt werden, welche zur Wahl dieser geführt haben.

Auf Basis der Literaturrecherche wurden sechs Verfahren in die engere Auswahl einbezogen, bei welchen es sich um Asynchronous Advantage Actor-Critic (A3C) [51], Deep Deterministic Policy Gradient (DDPG) [52], Actor-Critic using Kronecker-Factored Trust Region (ACKTR) [53], Policy Gradient and Q-Learning (PGQL) [54], Proximal Policy Optimization (PPO) [55] und Trust Region Policy Optimization (TRPO) [56] handelt.

A3C konnte im Vergleich zu anderen Algorithmen deutliche Vorteile im Bezug auf Geschwindigkeit, Robustheit und Performanz in verschiedenen Experimenten im Bereich von Atari-Spielen sowie im Autorennimulator TORCS erzielen. Da bei diesem Algorithmus jedoch mehrere Agenten verwendet werden, die untereinander erlernte Gewichte austauschen, ist der Algorithmus für die gewünschte Anwendung zur Erweiterung eines Trajektorienfolgeregelungskonzeptes nicht geeignet [51].

DDPG ist ein weiteres Actor-Critic-Verfahren, welches beispielsweise bereits in [57] erfolgreich für eine Anwendung im Fahrzeug verwendet wurde. Aufgrund der zugrundeliegenden Theorie ist es eines der einfacheren Verfahren im Deep Reinforcement Learning, welches für kontinuierliche Aktionsräume geeignet ist [58]. Durch das Wiederverwenden von bereits gesammelten Erfahrungen ist die Methode besonders dateneffizient. Bei diesem Off-Policy Verfahren entsteht durch das Lernen anhand von Daten, welche mit Policies abweichend der

aktuellen gesammelt wurden, eine Verschiebung im Policy Gradienten, wodurch möglicherweise das Konvergenzverhalten beeinträchtigt wird und der Agent entsprechend langsamer seine optimale Handlungsweise erreicht [58]. Zusätzlich bietet DDPG den Vorteil, dass die Exploration des Zustandsraums separat von der Aktionsauswahl betrachtet werden kann. Mit TRPO wurde in verschiedenen Atari-Spielen sowie simulierten Robotik-Aufgaben gezeigt, dass die Policy monoton verbessert werden kann. Bei diesem Algorithmus wird ein Bereich festgelegt, in welchem Parameteranpassungen toleriert werden. Das vorliegende Optimierungsproblem wird mittels eines konjugierten Gradienten-Verfahrens gelöst, welches entsprechend bei großen Netzwerken sehr rechenaufwändig ist [56].

PPO ist eine Verbesserung von TRPO, welche statt der Beschränkung des akzeptierten Parameterupdatebereichs einen Strafterm verwendet. In jedem Updateschritt wird sowohl eine Minimierung der Kostenfunktion als auch eine möglichst geringe Abweichung von der vorherigen Policy angestrebt, wobei jedoch der Rechenaufwand im Vergleich zu TRPO reduziert wird [55].

ACKTR ist ebenfalls ein Trust-Region-Verfahren. Es eignet sich sowohl für Aufgaben mit kontinuierlichen als auch mit diskreten Zustands- und Aktionsraum. Bei diesem Verfahren wird der natürliche Policy Gradient verwendet. Der natürliche Policy Gradient gewichtet das Update für jeden Netzparameter bezüglich des Einflusses auf die Ausgabe, um so die optimale Parameterschrittweite entlang des Gradienten zu finden [59]. Zur Gewichtung wird dabei häufig die Fisher-Information-Matrix (FIM) verwendet, welche beispielsweise mittels kronecker-faktorisierter Approximation (vgl. Kapitel 3.4) angenähert werden kann. Durch dieses Verfahren wird der Rechenaufwand zur Bestimmung der FIM reduziert, wodurch ACKTR ein recheneffizientes RL-Verfahren für beliebig dimensionale Zustandsräume darstellt [53].

Von den drei vorgestellten Trust-Region-Verfahren wird ACKTR gewählt, da es aufgrund der Verwendung des natürlichen Gradienten besonders vielversprechend erscheint. Da es sich um einen On-Policy Algorithmus handelt und entsprechend die Daten von der zu lernenden Policy stammen, konvergiert das Verfahren tendenziell schneller. Ebenfalls bieten Verfahren dieser Gruppe den Vorteil, dass die Exploration der Umgebung durch das Erlernen einer Verteilung zur Auswahl der Aktion, d. h. einer stochastischen Policy, sicher gestellt wird. Nachteilig bleibt jedoch, dass diese Verfahren anfällig für lokale Minima sind. Als dritter RL-Algorithmus wurde PGQL ausgewählt. Hierbei handelt es sich um ein Verfahren, welches On-Policy-Updates für Actor und Critic mit zusätzlichen Off-Policy-Updates für die Critic kombiniert [54]. In einigen in [54] getesteten Umgebungen konnte eine wesentlich höhere Dateneffizienz als beispielsweise Q-Learning gezeigt werden. Außerdem lässt die Kombination von On- und Off-Policy-Updates eine Ausnutzung der Vorteile beider Methoden vermuten.

3.2 Deep Deterministic Policy Gradient

Deep Deterministic Policy Gradient ist ein Algorithmus, den Lillicrap et al. 2019 veröffentlicht haben [52]. Es handelt sich um einen modellfreien Actor-Critic-Algorithmus, der auf dem Deterministic Policy Gradient Algorithmus [60] von Silver et al. basiert. DDPG nutzt Einflüsse von Deep Q-Networks [61] und ist sowohl für niedrigdimensionale, diskrete als auch hochdimensionale und kontinuierliche Aktionsräume geeignet [52]. Neben einer deterministischen Policy wird die State-Action-Value-Function erlernt, wozu insgesamt vier künstliche neuronale Netze mit eigenen Gewichtsmatrizen, ein Replaybuffer sowie ein Rauschprozess verwendet werden.

Bestandteile

Das Actor-KNN $\mu(s|\theta^\mu)$ mit dem Parametersatz θ^μ nutzt der Agent, um auf Basis seines aktuellen Zustands eine Aktion auszuwählen. Zur Beurteilung der Aktion im Zustand und der daraus zu erwartenden Belohnung wird die State-Action-Value-Function mithilfe des Critic-KNNs $Q(s, a|\theta^Q)$ approximiert. Zur Skalierung der Features im Actor-Netzwerk sowie in der ersten Schicht des Critic-Netzwerks wird Batch-Normalisierung (vgl. Kapitel 2.2.2) verwendet.

Der Target-Actor μ' und die Target-Critic Q' sind Kopien der beiden anderen Netzwerke, bei welchen die Parameterupdates nur anteilig übernommen werden. Infolgedessen ändern sich die Parameter pro Zeitschritt geringer, wodurch sich auch die Netzausgaben weniger unterscheiden und somit stabileres Lernen ermöglicht wird.

Da die erlernte deterministische Policy keine Exploration beinhaltet, wird in diesem Algorithmus ein Ornstein-Uhlenbeck (OU)-Prozess [62] verwendet. Hierbei handelt es sich um einen stochastischen Prozess, welcher über drei Parameter μ , θ und σ auf den jeweiligen Aktionsraum angepasst werden kann. In jedem Schritt wird eine Zustandsänderung

$$dX_t = \theta(\mu - X_t)dt + \sigma dW_t \quad (3.1)$$

zum Prozesszustand X_t addiert. Da entsprechend das aktuelle Rauschen mit dem vorherigen Rauschen zusammenhängt, entstehen weniger schnelle Richtungsänderungen im Rauschanteil und dadurch ähnliche Einflüsse auf die Aktion. Bei kontinuierlichen Regelungsaufgaben hängen die Aktionen häufig mit der Ausübung von Kräften, beispielsweise der Beschleunigung eines Fahrzeugs, zusammen. Um den Einfluss der Aktion auf die Position des Fahrzeugs zu ermitteln, ist doppelte Integration notwendig. Bei hochfrequenter Beschleunigung, beispielsweise durch Addieren von Gaußschem Rauschen, ändern sich entsprechend auch Geschwindigkeit und Position wesentlich schneller als bei niedrigfrequenten Aktionsänderungen wie bei dem OU-Rauschen. Durch die Verwendung eines Ornstein-Uhlenbeck-Prozesses wird die Exploration entsprechend unterstützt, da konsequenter einer Richtung im Aktionsraum gefolgt wird.

Der Replaybuffer \mathcal{R} ist ein Speicher mit vordefinierter Größe, in welchem die gesammelten Erfahrungstupel bestehend aus Zustand, Aktion, Nachfolgezustand und Reward gespeichert werden. Durch zufälliges Sampling aus diesem beim Lernen wird die (zeitliche) Abhängigkeit in den Daten vermindert und das Ziel von unabhängigen Datenpaaren realisiert.

Ablauf

In jedem Zeitschritt t wählt der Agent auf Basis seines aktuellen Zustands und mithilfe der im Actor-KNN erlernten Policy eine Aktion. Zu dieser wird der aktuelle interne Zustand des OU-Prozesses, nachfolgend bezeichnet als \mathcal{N}_t , addiert, um eine Exploration des Aktionsraum zu erreichen. Das gesammelte Erfahrungstupel bestehend aus Zustand, Aktion, Reward und Nachfolgezustand wird im Replaybuffer \mathcal{R} gespeichert, aus welchem der Agent zufällige Erfahrungen sampelt. Nach der Bestimmung des State-Value-Zielwertes mithilfe der Target-Netzwerke werden die Parameter der Critic unter Verwendung einer Loss-Funktion aktualisiert. Zum Update des Actors wird der gesampelte Policy-Gradient verwendet. Abschließend werden Anteile der neuen Gewichte für den Target-Actor und die Target-Critic übernommen. Algorithmus 4 zeigt den prinzipiellen Ablauf der Methode.

Algorithmus 4 DDPG

```

1: Critic-Network  $Q(s, a|\theta^Q)$  zufällig mit Gewichten  $\theta^Q$  initialisieren
2: Actor-Network  $\mu(s|\theta^\mu)$  zufällig mit Gewichten  $\theta^\mu$  initialisieren
3: Target-Network  $Q'$  mit Gewichten  $\theta^{Q'} \leftarrow \theta^Q$  initialisieren
4: Target-Network  $\mu'$  mit Gewichten  $\theta^{\mu'} \leftarrow \theta^\mu$  initialisieren
5: Replay-Buffer  $\mathcal{R}$  initialisieren
6: for episode = 1, M do
7:   Noise-Process  $\mathcal{N}$  initialisieren
8:   Initialen Beobachtungszustand  $s_1$  erhalten
9:   for t = 1, T do
10:    Aktion  $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$  auswählen
11:    Aktion  $a_t$  ausführen und Reward  $r_t$  und neuen Zustand  $s_{t+1}$  beobachten
12:    Erfahrungstupel  $(s_t, a_t, r_t, s_{t+1})$  in  $\mathcal{R}$  speichern
13:    Zufälligen Minibatch von  $N$  Erfahrungstupeln  $(s_t, a_t, r_t, s_{t+1})$  aus  $\mathcal{R}$  samplen
14:    Target  $y_i$  berechnen
15:    Critic durch Minimieren des Loss'  $L$  updaten
16:    Actor mithilfe des gesampelten Policy-Gradienten  $\nabla_{\theta} J$  updaten
17:    Target-Networks updaten
18:   end for
19: end for

```

Updates der KNN-Parameter

Für das Update der Critic wird eine Umgebung geschaffen, die Supervised Learning entspricht. Das Target-Actor-Netzwerk wird verwendet, um basierend auf den gesampelten Nachfolgezuständen die entsprechend nachfolgenden Aktionen zu prädizieren. Das Tupel bestehend aus Nachfolgezustand und Nachfolgeaktion wird anschließend vom Critic-Target-Network verwendet, um einen Q-Wert zu approximieren. Durch Gewichtung dieses Values und Addition mit dem gesampelten Reward ergibt sich der Zielwert

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'}). \quad (3.2)$$

Anschließend wird der mittlere quadratische Fehler zwischen diesem Zielwert und dem durch die Critic berechneten Q-Wert verwendet, um das Critic-KNN zu aktualisieren. Aufsummiert für alle Erfahrungstupel eines Batches ergibt sich das Loss L entsprechend zu

$$L = \frac{1}{N} \sum_i (y_i - Q(s_i, a_i|\theta^Q))^2. \quad (3.3)$$

Um anschließend das Actor-KNN zu aktualisieren, wird der zu erwartende Return ausgehend von einem Startzustand s_t als Gütemaß J verwendet. Unter Verwendung des gleichbedeutenden Q-Wertes folgt:

$$\nabla_{\theta^\mu} J = \mathbb{E} [\nabla_{\theta^\mu} Q(s, a|\theta^Q)|_{s=s_t, a=\mu(s_t|\theta^\mu)}]. \quad (3.4)$$

Durch Sampling und Anwendung der Kettenregel kann der Policy Gradient, welcher für das Update der Actor-Parameter verwendet wird, durch

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i} \quad (3.5)$$

approximiert werden.

Die Updates der Target-Actor-KNN-Parameter $\theta^{\mu'}$ bestimmen sich nach

$$\theta_{t+1}^{\mu'} = \tau \theta_{t+1}^{\mu} + (1 - \tau) \theta_t^{\mu'} \quad (3.6)$$

und die des Target-Critic-KNNs analog nach

$$\theta_{t+1}^{Q'} = \tau \theta_{t+1}^Q + (1 - \tau) \theta_t^{Q'}, \quad (3.7)$$

wobei τ der Update-Parameter ist, welcher häufig auf 0.001 gesetzt wird [52].

3.3 Policy Gradient and Q-Learning

Policy Gradient and Q-Learning ist ein Algorithmus, welcher 2017 veröffentlicht wurde [54]. Bei diesem Verfahren wird ein Policy Gradient-Ansatz, welcher On-Policy seine Policy verbessert, um eine Off-Policy-Komponente in Form von Q-Learning (vgl. Kapitel 2.1.3) erweitert. Im Gegensatz zu DDPG erlernt der Agent in diesem Algorithmus eine stochastische Policy, weshalb keine zusätzliche Exploration notwendig ist. Als Critic-Anteil wird die State-Value-Function erlernt. Im Algorithmus werden zwei künstliche neuronale Netze und ein Replaybuffer verwendet.

Bestandteile

Das Actor-KNN $\pi(s|\theta^\pi)$ mit dem Parametersatz θ^π wird zur Berechnung des Mittelwertes und der Standardabweichung einer Gaußverteilung verwendet, da diese in der Literatur häufig zur Abbildung einer stochastischen Policy genutzt wird. Aus dieser wird anschließend die gewählte Aktion im aktuellen Zustand gesampelt. Das Critic-Netzwerk $V(s|\theta^V)$ mit den Gewichten θ^V dient zur Approximation einer State-Value-Function.

Der Replaybuffer \mathcal{R} ist wie bei DDPG ein Speicher vordefinierter Größe, in welchem die gesammelten Erfahrungstupel des Agenten gespeichert werden. Dieser wird für das Q-Learning-Update verwendet, wobei ebenfalls der Vorteil von unabhängigen Datenpunkten ausgenutzt wird.

Bei Online-PG-Updates wird häufig ein Entropie-Regularisierer hinzugefügt, der verhindern soll, dass die Policy deterministisch wird. Für die Entropie H^π einer Policy π in einem Zustand s gilt

$$H^\pi(s) = - \sum_a \pi(s, a) \log \pi(s, a). \quad (3.8)$$

Durch die zusätzliche Addition eines geringen Anteils der Entropie beim Update der Actor-Netzwerkparameter wird entsprechend sichergestellt, dass der RL-Agent seine Umwelt kontinuierlich weiter erkundet.

Ablauf

Die theoretischen Grundlagen unter der Annahme, dass eine perfekte Critic Q^π als Referenz bekannt ist, sind [54] zu entnehmen. Abweichend von der Theorie ist in der Praxis die optimale Critic jedoch nicht bekannt, weshalb das Verfahren eine Schätzung der Critic verwendet.

In jedem Zeitschritt t wird das Actor-Netzwerk genutzt, um den Mittelwert μ und die Standardabweichung σ einer Gaußverteilung \mathcal{N} zu bestimmen, von welcher anschließend eine Aktion a gesampelt wird. Das Erfahrungstupel $(s_t, \log \pi(s_t, a_t), H^\pi(s_t), r_t, s_{t+1})$ bestehend aus dem Zustand s_t , dem Logarithmus der Policy evaluiert an der Stelle der

Action $\log \pi(s_t, a_t)$, der Entropie der Policy $H^\pi(s_t)$, dem Reward r_t und dem Nachfolgezustand s_{t+1} wird im Replaybuffer \mathcal{R} gespeichert. In jedem Schritt werden das Actor- und das Critic-KNN mithilfe eines TD-Actor-Critic-Verfahrens [38] angepasst. In beliebigen Abständen wird ein Batch aus dem Replaybuffer gesampelt, welcher für ein zusätzliches Q-Learning-Update des Critic-KNNs verwendet wird. Der prinzipielle Ablauf dieses Verfahren ist Algorithmus 5 zu entnehmen.

Algorithmus 5 PGQL

```

1: Critic-Netzwerk  $V(s|\theta^V)$  zufällig mit Gewichten  $\theta^V$  initialisieren
2: Actor-Netzwerk  $\pi(s|\theta^\pi)$  zufällig mit Gewichten  $\theta^\pi$  initialisieren
3: Replay-Buffer  $\mathcal{R}$  initialisieren
4: for episode = 1, M do
5:   for t = 1, T do
6:     Aktion  $a_t \sim \pi(a|\mu, \sigma) = \mathcal{N}(a|\mu, \sigma)$ 
7:     Aktion  $a_t$  ausführen und Reward  $r_t$  und neuen Zustand  $s_{t+1}$  beobachten
8:     Erfahrungstupel  $(s_t, \log \pi(s_t, a_t), H^\pi(s_t), r_t, s_{t+1})$  in  $\mathcal{R}$  speichern
9:     Critic durch Minimieren des Loss'  $L$ 
10:    Actor mithilfe des Policy Gradienten updaten
11:    if zusätzliches Q-Learning-Update then
12:      Zufälligen Minibatch von  $N$  Erfahrungstupeln
         $(s_t, \log \pi(s_t, a_t), H^\pi(s_t), r_t, s_{t+1})$  aus  $\mathcal{R}$  samplen
13:      Q-Wert für aktuellen State und Aktion schätzen
14:      Q-Wert für nächsten State und nächste Aktion schätzen
15:      Critic updaten
16:    end if
17:  end for
18: end for

```

Updates der KNN-Parameter

Mithilfe des Critic-Netzwerks wird der Value für den aktuellen und für den nächsten Zustand geschätzt, um basierend auf diesen den TD-Error δ zu bestimmen:

$$\delta_t = (r + \gamma V(s_{t+1})) - V(s_t). \quad (3.9)$$

Dieser Fehlerterm wird quadratisch als Loss-Funktion L für die Critic verwendet. Der Term in den Klammern wird auch als TD-Target bezeichnet und entsprechend einem „Label“ im Supervised Learning verwendet. Für das Update des Actor-Netzwerks wird der Gradient der Zielfunktion J bezogen auf die Parameter θ der Policy benötigt. Hier gilt entsprechend des Policy Gradient Theorems

$$\nabla_\theta J(\theta) = \mathbb{E} [\nabla_\theta (\log \pi_\theta(s, a)) Q_\pi(s, a)]. \quad (3.10)$$

Analog zur Baseline bei REINFORCE (vgl. Kapitel 2.1.4) kann der State-Value V verwendet werden, um den tatsächlichen „Vorteil“ der Wahl einer Aktion gegenüber allen anderen in einem Zustand zu bewerten. Zur Berechnung wird die sogenannte Advantage-Funktion

$$A(s, a) = Q(s, a) - V(s) \quad (3.11)$$

verwendet. Da der Q-Wert durch den mit Ausführung der entsprechenden Aktion erhaltenen Reward und den State-Value des Nachfolgezustands approximiert werden kann, gilt:

$$A(s, a) = r + \gamma V(s_{t+1}|s, a) - V(s) = \delta_\pi. \quad (3.12)$$

Aufgrund dessen, dass der Value eines States nicht von der Action abhängt, kann der Q-Value in Gleichung 3.10 durch die Advantage-Function bzw. den TD-Error ersetzt werden, ohne den Policy Gradienten zu beeinflussen. Entsprechend folgt für den Policy Gradienten, welcher für das Parameterupdate des Actor-KNN genutzt wird:

$$\nabla_\theta J(\theta) = \mathbb{E} [\nabla_\theta (\log \pi_\theta(s, a)) \delta_\pi]. \quad (3.13)$$

Die durch Ausführung der Aktionen in den verschiedenen Zuständen gesammelten Erfahrungstupel werden in einem Replaybuffer gespeichert. In beliebigen Abständen sampelt ein unabhängiger Prozess Daten aus diesem Speicher und performt basierend auf ihnen ein Q-Learning-Update, welches die Critic verbessert. Der Q-Wert eines State-Action-Paares kann mithilfe des Logarithmus der Policy, der Entropie der Policy und dem berechneten State-Value der Critic approximiert werden:

$$\tilde{Q}^\pi(s, a) = \alpha(\log \pi(s, a) + H^\pi(s)) + V(s). \quad (3.14)$$

Der Hyperparameter α gewichtet dabei den Einfluss der Policy auf die Schätzung des Q-Wertes. Entsprechend dieser Gleichung werden mit den gesampelten Erfahrungstupeln aus dem Replaybuffer die Q-Values für das aktuelle und das nachfolgende Zustand-Action-Paar approximiert. Anschließend wird ein Q-Learning-Update nach Gleichung 2.16 durchgeführt.

3.4 Actor-Critic using Kronecker-Factored Trust Region

Actor-Critic using Kronecker-Factored Trust Region ist ein On-Policy Actor-Critic-Verfahren, welches 2017 von Wu et al. veröffentlicht wurde [53]. Hierbei wird der natürliche Policy Gradient (vgl. Kapitel 3.1) mittels Kronecker-factored approximate curvature (K-FAC) mit Trust Region approximiert, um Actor und Critic zu optimieren. Die Methode ist sowohl für kontinuierliche als auch für diskrete Aktions- und Zustandsräume geeignet. Bei ACKTR wird mithilfe von zwei künstlichen neuronalen Netzen eine stochastische Policy und die State-Value-Function erlernt.

Bestandteile

Das Actor-Netz $\pi(s|\theta^\pi)$ mit den Gewichtungen θ^π lernt wie bei PGQL den Mittelwert und die Standardabweichung einer Gaußverteilung, welche für das Sampling der Aktion verwendet wird. Die State-Value-Function wird mittels des Critic-KNNs $V(s|\theta^V)$ mit dem Parametersatz θ^V nachgebildet.

Ablauf

In jedem Zeitschritt t werden mittels des Actor-KNN der Mittelwert und die Standardabweichung einer Gaußverteilung \mathcal{N} prädiziert, von welcher eine Aktion gesampelt wird. Dabei werden die Aktivierungen der einzelnen Neuronen gespeichert, um mit deren Hilfe am Ende der Episode die Fisher-Information-Matrix zur Bestimmung des natürlichen Gradienten zu approximieren. Die gewählte Aktion wird anschließend ausgeführt und der erhaltene Reward und der Nachfolgezustand beobachtet. Der Logarithmus der Policy evaluiert an der Stelle der Action $\log \pi(s_t, a_t)$ und der Value v werden bis zum Episodenende

gespeichert, da erst nach Erreichen eines Endzustandes die Parameterupdates berechnet werden. Der prinzipielle Ablauf von ACKTR ist in Algorithmus 6 zu sehen.

Algorithmus 6 ACKTR

```

1: Critic-Network  $V(s|\theta^V)$  zufällig mit Gewichten  $\theta^V$  initialisieren
2: Actor-Network  $\pi(s|\theta^\pi)$  zufällig mit Gewichten  $\theta^\pi$  initialisieren
3: for episode = 1, M do
4:   Initialen Beobachtungszustand  $s_1$  erhalten
5:   for t = 1, T do
6:     Aktion  $a_t \sim \pi(a|\mu, \sigma) = \mathcal{N}(a|\mu, \sigma)$ 
7:     Aktion  $a_t$  ausführen und Reward  $r_t$  und neuen Zustand  $s_{t+1}$  beobachten
8:     Value  $v$  des aktuellen Zustands vorhersagen
9:     Erfahrungstupel  $(\log \pi(s_t, a_t), v)$  bis Episodenende speichern
10:    end for
11:    Critic durch Minimieren des Loss'  $L$  updaten
12:    Actor mithilfe des Policy Gradienten updaten
13: end for

```

Updates der KNN-Parameter

ACKTR nutzt den natürlichen Gradienten für die Updates der Netzwerkgewichtungen. Der natürliche Gradient liefert die Richtung, in welche die Parameter verändert werden müssen, um die aktuell größte, direkte Verbesserung der Zielfunktion zu erreichen. Bei Verfahren, die den natürlichen Policy Gradienten verwenden, wird die Zielfunktion, beispielsweise in Form des zu erwartenden Returns, unter der Bedingung maximiert, dass die Norm $\|\Delta\theta\|_F$ einen vordefinierten Wert δ nicht überschreitet. Der Hyperparameter δ legt dabei eine sogenannte Trust Region fest, in welcher die Parameter θ verändert werden dürfen. Für die Norm gilt:

$$\|\Delta\theta\|_F = (\Delta\theta^T F \Delta\theta)^{\frac{1}{2}}, \quad (3.15)$$

wobei $\Delta\theta$ die Differenz zwischen den vorherigen und den aktuellen Parametern bezeichnet. F bezeichnet dabei die Fisher-Information-Matrix, welche eine Ableitung zweiter Ordnung der Kullback-Leibler (KL) -Divergenz ist und als Distanzfunktion verwendet werden kann. Die KL-Divergenz beschreibt den Unterschied zwischen zwei Verteilungen P und Q , wobei

$$D_{KL}(P||Q) = \sum_{x=1}^N P(x) \log \frac{P(x)}{Q(x)} \quad (3.16)$$

gilt. Je verschiedener die beiden Verteilungen, desto größer die KL-Divergenz [63]. Unter Einbezug der approximierten Fisher-Matrix und des Policy Gradienten ergibt sich das natürliche Policy Gradienten Update für die Parameter θ zu

$$\theta \leftarrow \theta - \eta F^{-1} g, \quad (3.17)$$

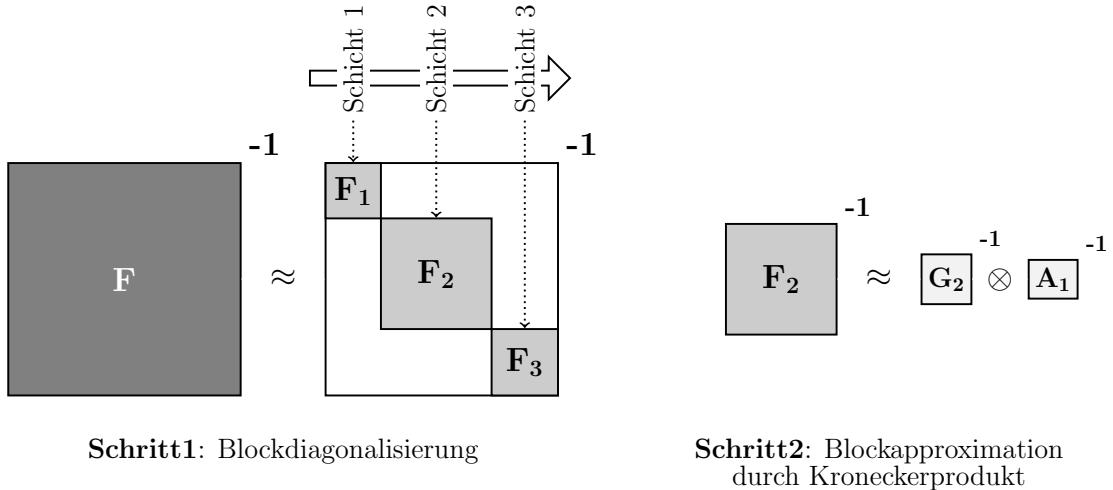
welches dem SGD-Update mit zusätzlicher Gewichtung durch die inverse Fisher-Matrix entspricht. Da laut [56] im Zusammenhang mit tiefen KNN dabei häufig sehr große Parameterupdates und damit einhergehend eine nahezu deterministische Policy entstehen, wird

die tatsächliche Lernrate η durch

$$\min \left(\eta_{max}, \sqrt{\frac{2\delta}{\Delta\theta^T \hat{F} \Delta\theta}} \right) \quad (3.18)$$

ersetzt, wobei die maximale Lernrate η_{max} und der Trust Region Radius δ Hyperparameter sind.

Natural Policy Gradient ist ein Optimierungsverfahren zweiter Ordnung, weshalb es komplex und rechenintensiv ist. Da die Berechnung der Inversen der FIM für die vielen Parameter eines KNN sehr rechenaufwändig ist, existieren verschiedene Approximierungsmethoden für den natürlichen Gradienten. K-FAC [64,65] gehört zu der Gruppe der Verfahren, welche die Fisher-Information-Matrix mittels eines Vorgehens approximieren, welches eine einfache Berechnung der Inversen ermöglicht. Dazu werden mehrere Kronecker-Produkte kleinerer inverser Matrizen verwendet, die anschließend zusammengefügt werden. Abbildung 3.1 visualisiert das Vorgehen von K-FAC zur Approximation der inversen Fisher-Information-Matrix am Beispiel eines KNN mit drei Schichten.



Schritt1: Blockdiagonalisierung

Schritt2: Blockapproximation durch Kroneckerprodukt

Abbildung 3.1: Approximation der inversen FIM mittels K-FAC für ein KNN mit drei Schichten nach [7]

Im ersten Schritt wird die inverse FIM als Blockdiagonalmatrix dargestellt, wobei die jeweiligen Blöcke die Parameter der einzelnen Layer enthalten. Jeder dieser Blöcke wird im nächsten Schritt durch das Kronecker-Produkt der Ableitungsmatrix der aktuellen Schicht und der Neuronaktivierungsmatrix der vorherigen Schicht approximiert. Dabei nutzt K-FAC die Eigenschaft des Kronecker-Produkts, die besagt, dass

$$(A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \quad (3.19)$$

gilt. Der Diagonalblock F_i der i-ten Schicht kann durch die Gleichung

$$F_i = \mathbb{E} [\nabla_i \log p(y|x, \theta) \nabla_i \log p(y|x, \theta)^T] \quad (3.20)$$

beschrieben werden, wobei p eine beliebige Verteilung ist. Mittels Backpropagation kann der Gradient von $\log p(y|x, \theta)$ als Kronecker-Produkt des Gradienten des Outputs der i-

ten Schicht g_i und der Aktivierung der Vorgängerschicht a_{i-1} ausgedrückt werden. Daraus folgt:

$$\begin{aligned} F_i &= \mathbb{E} [g_i g_i^T \otimes a_{i-1} a_{i-1}^T] \\ &\approx \mathbb{E} [g_i g_i^T] \otimes \mathbb{E} [a_{i-1} a_{i-1}^T] \\ &= G_i \otimes A_{i-t}. \end{aligned} \quad (3.21)$$

Entsprechend gilt für die Approximierung der inversen FIM:

$$F_i^{-1} = G_i^{-1} \otimes A_{i-t}^{-1}. \quad (3.22)$$

Die so approximierte inverse FIM kann anschließend zur Gewichtung des Policy Gradienten verwendet werden, um die Parameterupdates schichtweise zu berechnen, wobei

$$\Delta\theta_i = \min \left(\eta_{max}, \sqrt{\frac{2\delta}{\Delta\theta_i^T F_i \Delta\theta_i}} \right) F_i^{-1} g_i \quad (3.23)$$

gilt.

4 Validierung der Algorithmik

Bevor die Algorithmen in der Simulationsumgebung zur automatisierten Fahrzeugführung verwendet werden, soll deren Implementierung anhand eines Minimalbeispiels validiert werden. Auf Basis dieses Beispiels sollen erste Erkenntnisse bezüglich der Anwendbarkeit und mögliche Besonderheiten der Algorithmen gesammelt werden. Das Continuous Mountain Car (CMC) wird verwendet, da es sich hierbei um eine Umgebung handelt, welche einen kontinuierlichen Zustands- sowie Aktionsraum verwendet. Zudem ist aufgrund der geringen Komplexität eine schnelle Evaluierung möglich.

Das Continuous Mountain Car ist eine der Umgebungen, welche OpenAI in ihrer Bibliothek *gym* zur Verfügung stellt. Diese Sammlung beinhaltet mehrere sogenannte Environments, mit denen ein RL-Agent interagieren und somit trainiert werden kann [66].

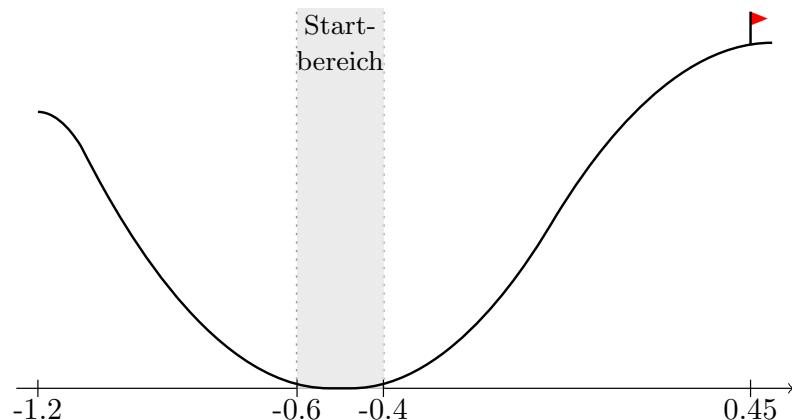


Abbildung 4.1: Umgebung des Continuous Mountain Cars

Beim Mountain Car handelt es sich um ein Auto, dass sich auf einer eindimensionalen Spur zwischen zwei Bergen befindet und das Ziel auf der rechten Seite erreichen soll (siehe Abbildung 4.1). Da der Motor des Fahrzeugs nicht stark genug ist, muss mithilfe beider Berge ein Momentum aufgebaut und ausgenutzt werden, um die Aufgabe zu erfüllen. In der verwendeten kontinuierlichen Version wird auf Basis des Zustandes, welcher aus Fahrzeugposition und -geschwindigkeit besteht, eine Aktion bestimmt, die der angewandten Motorkraft entspricht. Die Wertebereiche für Zustand und Aktion sind Tabelle 4.1 zu entnehmen.

Tabelle 4.1: Bestandteile und Wertebereiche für Zustand und Aktion beim Continuous Mountain Car

	Bezeichnung	Minimum	Maximum
Zustand	Fahrzeugposition	-1.2	1.2
	Fahrzeuggeschwindigkeit	-0.07	0.07
Aktion	angewandte Motorkraft	-1	1

Zum Start jeder neuen Episode wird der Agent im Startbereich, welcher in Abbildung 4.1 grau markiert ist, mit einer Geschwindigkeit von 0 platziert. Auf Basis dieses Zustandes wird mithilfe des verwendeten RL-Algorithmus eine Aktion gewählt. Nach der Ausführung dieser Aktion a liefert die Umgebung einen nächsten Zustand und einen Reward r zurück,

welcher nach folgender Gleichung berechnet wird:

$$r = \begin{cases} 100 & \text{Ziel erreicht} \\ -0.1 * a^2 & \text{sonst.} \end{cases} \quad (4.1)$$

Entsprechend wird der Agent für höheren Kraftaufwand stärker bestraft. Der Agent erhält lediglich einen positiven Einzelreward, wenn dieser die Zielmarkierung, welche einer Position von 0.45 entspricht, überschreitet.

4.1 Netzwerkarchitekturen und Hyperparameter

Mit jedem der Algorithmen wurden unter Variation der jeweiligen Hyperparameter verschiedene RL-Agenten in der Continuous Mountain Car Umgebung trainiert. Zur Abbildung der Policy und der Value-Function wird jeweils ein künstliches neuronales Netz verwendet. Der prinzipielle Aufbau für die verwendeten KNN wurde auf fully-connected MLPs mit zwei versteckten Schichten festgelegt, wobei in den versteckten Schichten ReLU als Aktivierungsfunktion verwendet wird. Abbildung 4.2 zeigt die Netzwerkarchitekturen für die Actor-KNN.

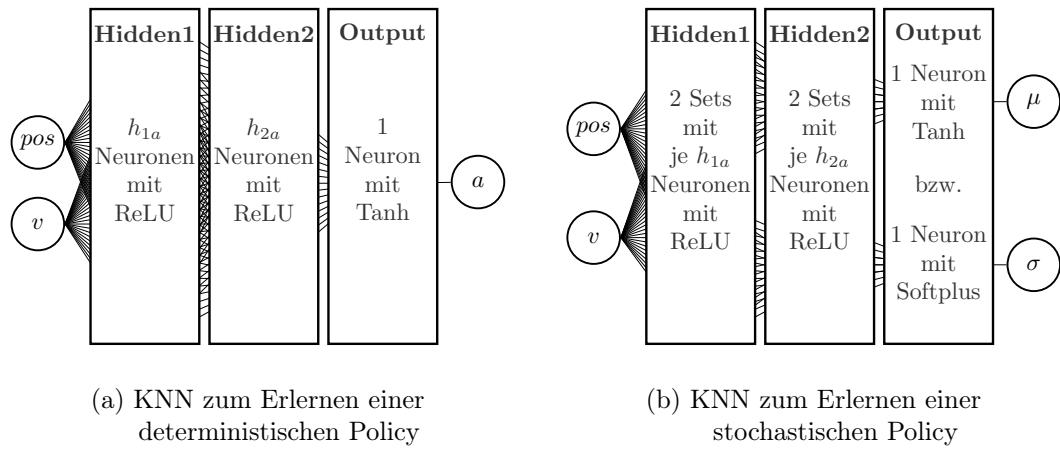


Abbildung 4.2: Netzarchitekturen für Actor-KNN

Da bei DDPG eine deterministische Policy erlernt wird, wird das fully-connected KNN aus Abbildung 4.2a verwendet, bei welchem basierend auf der Fahrzeugposition pos und der Fahrzeuggeschwindigkeit v direkt eine explizite Aktion a bestimmt wird. ACKTR und PGQL verwenden die Architektur aus Abbildung 4.2b, um den Mittelwert μ und die Standardabweichung σ einer Gaußfunktion zu erlernen, von welcher anschließend eine Aktion gesampelt wird. Die Anzahl der Neuronen h_{1a} in der ersten und h_{2a} in der zweiten versteckten Schicht der KNN sind neben der Lernrate ein beliebig wählbarer Hyperparameter.

Als Critic wird je nach Algorithmus entweder eine State-Action- oder eine State-Value-Function mittels eines KNN abgebildet, wie Abbildung 4.3 zeigt. Das in Abbildung 4.3a dargestellte KNN wird von DDPG-Agenten verwendet, da diese basierend auf dem aktuellen Zustand und der in diesem gewählten Aktion eine State-Action-Value-Function Q als Critic erlernen. Bei ACKTR und PGQL wird hingegen eine State-Value-Function V erlernt, weshalb die Handlung des Agenten als Eingang in das Critic-KNN nicht notwendig ist.

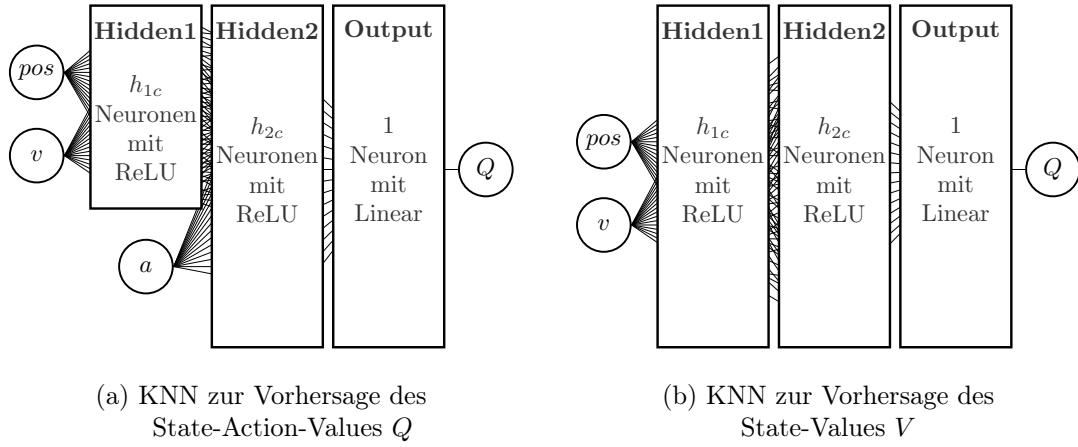


Abbildung 4.3: Netzarchitekturen für Critic-KNN

Die Anzahl der Trainingsdurchläufe sowie der darin maximal getätigten Aktionen des Agenten sind ebenfalls vor Trainingsbeginn festzulegen. Bei allen Methoden wird der Hyperparameter γ verwendet, welcher die Gewichtung der zukünftigen Rewards bei der Berechnung des Returns festlegt. Tabelle 4.2 zeigt die zusätzlichen algorithmenspezifischen Hyperparameter, die anschließend näher erläutert werden.

Tabelle 4.2: Algorithmenspezifische Hyperparameter

Algorithmus	$ep_{expl,max}$	$ep_{expl,only}$	Noise μ, σ, θ	Batchsize	$qsteps$	δ
DDPG	x	x	x	x		
PGQL				x	x	
ACKTR						x

Bei DDPG sind zusätzlich die Anzahl der Episoden zu definieren, in welchen der Aktion Rauschen hinzugefügt wird, um die Exploration des Agenten zu garantieren. Der OU-Rauschprozess wird dabei über die Parameter μ, σ, θ definiert. Der Hyperparameter $ep_{expl,only}$ legt die Anzahl der Episoden fest, in welchen der Agent Erfahrungen sammelt sowie im Replaybuffer speichert, dabei jedoch noch nicht lernt. $ep_{expl,max}$ gibt die maximale Episodenanzahl an, in welchen externe Exploration zugefügt wird. Das OU-Rauschen wird mittels des Gewichtungsfaktors n , berechnet nach

$$n = \begin{cases} 1 & \text{Episode } \leq ep_{expl,only} \\ 1 - \text{episode}/ep_{expl,max} & ep_{expl,only} < \text{Episode } \leq ep_{expl,max} \\ 0 & \text{sonst,} \end{cases} \quad (4.2)$$

gewichtet. Während der explore-only Episoden, in welchen der Agent seine Umwelt erkundet und die gesammelten Erfahrungen in den Replaybuffer schreibt ohne zu lernen, wird das gesamte Rauschen hinzugefügt. Während der weiteren Runden bis zum Erreichen der Maximalanzahl an Explorationsepisoden wird das Rauschen linear reduziert. Im Anschluss wird der prädizierten Aktion kein Rauschen mehr hinzugefügt. Der entsprechende Verlauf des Gewichtungsfaktors n ist Abbildung 4.4 zu entnehmen.

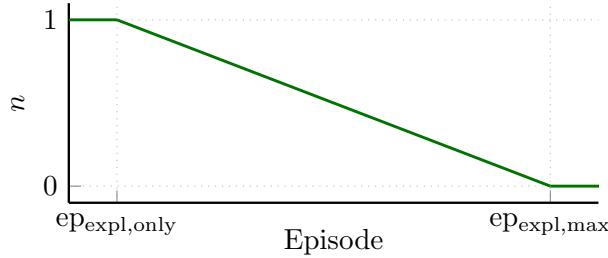


Abbildung 4.4: Verlauf des Gewichtungsfaktors n für das externe Rauschen bei DDPG

Mithilfe der Batchsize wird die Anzahl an Erfahrungstupeln festgelegt, welche für das Lernen aus dem Replaybuffer gesampelt werden. Bei PGQL wird zusätzlich mittels qsteps festgelegt, in welchen Abständen das zusätzliche Q-Learning Update für das Critic-KNN ausgeführt wird. Die Trust Region, welcher die Änderung der Parameter beschränkt, wird bei ACKTR durch den Hyperparameter δ bestimmt.

4.2 Simulationsergebnisse

Die Funktionsfähigkeit der implementierten Algorithmen soll durch Trainieren verschiedener Agenten in der Continuous Mountain Car Umgebung geprüft werden. Als Maß der Güte wird hierbei jeweils der über eine Episode aufsummierte Reward betrachtet.

OpenAI Gym definiert die CMC-Challenge als erfolgreich gelöst, wenn ein Episodenreward über 90 erreicht wird [67]. Für die nachfolgenden Untersuchungen wurde jeder Agent 200 Episoden trainiert. Eine Episode endet, wenn der Agent das Ziel auf dem rechten Hügel erreicht oder maximal 1000 Aktionen ausgeführt wurden.

Training mit DDPG

Die verwendeten KNN entsprechen dem Aufbau in Abbildung 4.2a bzw. Abbildung 4.3a, wobei die Neuronenanzahl in den versteckten Schichten auf $h_{1a} = h_{1c} = 20$ und $h_{2a} = h_{2c} = 10$ gesetzt wurde. Die Lernrate für den Actor beträgt 10^{-4} , die für die Critic 10^{-3} . Ausgenommen der letzten Schicht werden alle Gewichte mittels der PyTorch-Standardinitialisierung gesetzt. PyTorch initialisiert die Gewichte eines linearen Layers standardmäßig mittels einer He-Initialisierung und den Bias mittels einer Gleichverteilung zwischen $-\frac{1}{\sqrt{n}}$ und $\frac{1}{\sqrt{n}}$, wobei n die Anzahl der Eingaben in die entsprechende Schicht angibt. In der letzten Schicht werden die Anfangsgewichtungen aus einer Normalverteilung mit einem Mittelwert von 0 und einer Standardabweichung von 0.0001 gesampelt. Der verwendete Replaybuffer kann maximal 10^6 Erfahrungstupel beinhalten, bevor die ältesten gelöscht und durch die neusten ersetzt werden. Von den 200 maximalen Trainingsepisoden erkundet der Agent in den ersten zehn ausschließlich seine Umgebung, wobei der zu jeder Aktion addierte Rauschanteil maximal ist und speichert die Erfahrungen im Replaybuffer. Bis einschließlich Episode 100 wird der Faktor zur Gewichtung des zur Aktion addierten Rauschens nach Gleichung 4.2 berechnet und entsprechend mit fortschreitender Episodenzahl immer geringer. Die Parameter für den Ornstein-Uhlenbeck-Rauschprozess werden auf $\mu = 0$, $\sigma = 0.2$ und $\theta = 0.15$ gesetzt. Zur Berechnung von jedem Parameterupdate werden aus dem Replaybuffer zufällig 64 Erfahrungstupel gesampelt. Der Hyperparameter γ zur Gewichtung der zukünftigen Rewards wird auf 0.99 gesetzt. Die Target-Netzwerke kopieren zu Beginn die initialen Gewichte von Actor und Critic. Anschließend werden in jedem Lernschritt für die Parameterupdates der Target-KNN die zuvor berechneten Parameterupdates mit

einem Faktor von 0.001 gewichtet übernommen.

Abbildung 4.5 zeigt den Verlauf des über eine Episode aufsummierten Rewards für einen mit DDPG trainierten RL-Agenten. Vor Episode 100 wurde jede fünfte zur Evaluation genutzt, d. h. die durch das Actor-Netzwerk vorhergesagten Aktionen wurden ohne zusätzliches Rauschen vom Agenten ausgeführt. Da nach Runde 100 die tatsächlich prädizierten Aktionen ohne zusätzliches Rauschen ausgeführt werden, kann jede Runde zur Kontrolle der aktuellen Agenten-Performanz genutzt werden.

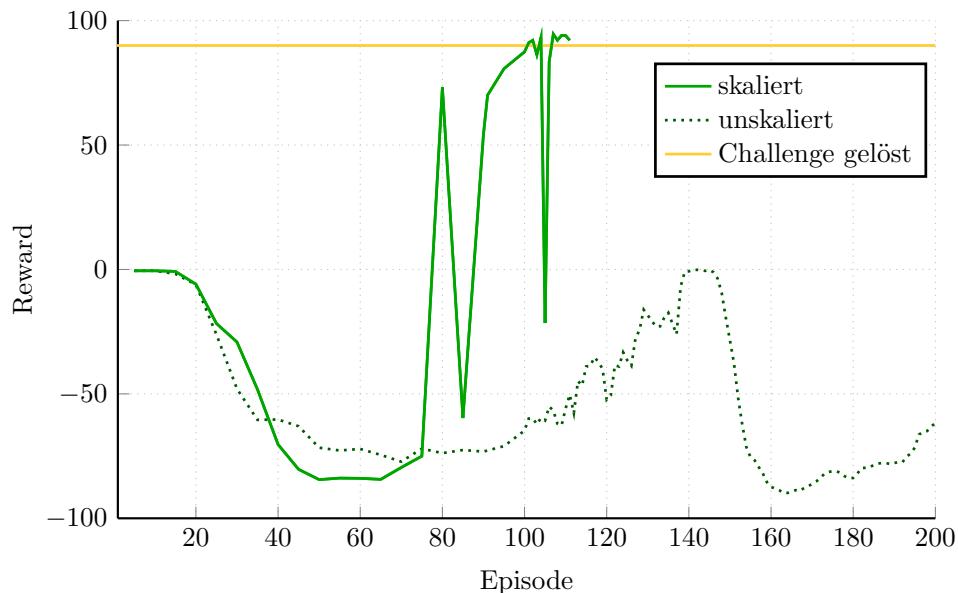


Abbildung 4.5: Vergleich von zwei DDPG-Agenten mit verschiedenen Eingangsdaten

Die grüne Kurve zeigt den Verlauf bei vorverarbeiteten Eingangsgrößen. Entsprechend wurden sowohl die Fahrzeugposition als auch die Fahrzeuggeschwindigkeit auf einen Bereich von -1 bis 1 skaliert. Die gepunktete Kurve zeigt den Verlauf ohne Vorverarbeitung der Netzeingänge. Die orangefarbene Linie kennzeichnet den Challenge-Erfolg mit einem Episodenreward von 90.

Die Übereinstimmung der Episodenrewards beider Agenten in Runde 5 und Runde 10 beruhen auf der Tatsache, dass in den ersten zehn Runden ausschließlich Erfahrungen gesammelt und die KNN noch nicht trainiert werden.

Der durchgehend negative Episodenreward des Agenten, welcher auf Basis der nicht vorverarbeiteten Zustände trainiert wurde, beweist, dass dieser innerhalb der betrachteten Episoden das Ziel nie erreicht.

Bei Verwendung der vorverarbeiteten KNN-Eingangsdaten ist das Training erfolgreicher. Der Agent erreicht während der Evaluation in Runde 80 basierend auf den gewählten Aktionen mittels des Actor-KNNs erstmals das Ziel. Ab Episode 107 erreicht der Agent fünfmal in Folge einen Reward über 90, weshalb das Training vorzeitig beendet wird. Im Trainingsverlauf zeigen sich zwei Einbrüche des Rewards, den der Agenten mit vorverarbeiteten Eingangsdaten erhält. Eine mögliche Ursache hierfür ist die Position, von welcher der Agent startet. Wurde die zugehörige oder ähnliche Startpositionen während des Trainings seltener oder nicht besucht, können während des Trainings schlechtere Ergebnisse auftreten. Wie nachfolgend Abschnitt 4.3 zeigen wird, erlernt der Agent bis zum Trainingsende gewinnbringende Verhaltensweisen ausgehend von einer Vielzahl an Startpositionen.

Da der Agent ohne vorverarbeitete Eingangsgrößen während der Phase mit externer Exploration, welche in Runde 100 endet, das Ziel nie erreicht und entsprechend keine Erfahrungstupel mit positiven Reward im Replaybuffer gespeichert hat, kann dieser die Challenge nicht lösen. Der nicht kontinuierliche Verlauf ergibt sich aufgrund der verschiedenen Startpositionen in Kombination mit dem fortlaufenden Training. Da sich bei DDPG die Vorverarbeitung der Eingangsdaten als nützlich erwiesen hat, werden die nachfolgenden Trainingsdurchläufe ausschließlich mit zuvor auf einen Bereich von -1 bis 1 skalierten Netzeingaben durchgeführt.

Training mit PGQL

Da der PGQL-Agent eine stochastische Policy und eine State-Value-Function erlernt, werden die Netzwerkarchitekturen nach Abbildung 4.2b und Abbildung 4.3b verwendet. Die Neuronenanzahl in den versteckten Schichten wird wie bei DDPG auf $h_{1a} = h_{1c} = 20$ und $h_{2a} = h_{2c} = 10$ gesetzt. Ebenso analog betragen die Lernrate für das Actor-KNN 10^{-4} und für das Critic-KNN 10^{-3} . Für das zusätzliche Q-Learning-Update, welches in jedem zehnten Agentenschritt berechnet wird, wird ein Batch bestehend aus 64 Erfahrungstupeln aus einem Replaybuffer mit maximal 10^6 möglichen Einträgen gesampelt. Der Hyperparameter γ zur Gewichtung der zukünftigen Rewards wird auf 0.99 gesetzt, der Hyperparameter α , welcher nach Gleichung 3.14 den Einfluss der Policy-Gaußverteilung auf die Schätzung des Q-Wertes angibt, auf 0.001.

Analog zum vorherigen Experiment wurde der PGQL-Agent zuerst ausgehend von der PyTorch-Standardinitialisierung in den vorderen und der Initialisierung mittels einer Normalverteilung $\mathcal{N}(\mu = 0, \sigma = 0.0001)$ in der letzten Schicht, trainiert. Das Ergebnis dieses Trainingsdurchlaufs zeigt die gepunktete Kurve in Abbildung 4.6.

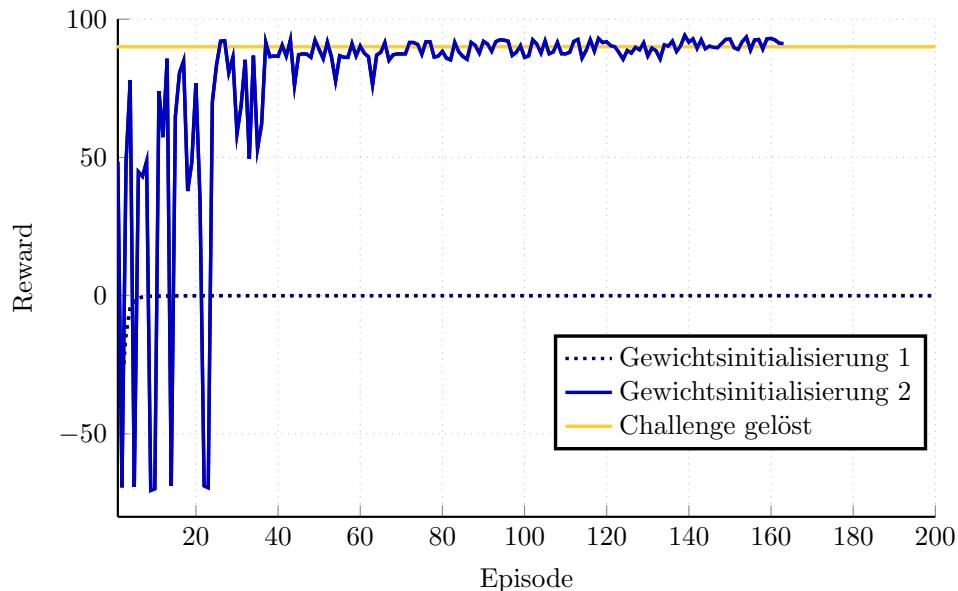


Abbildung 4.6: Vergleich von zwei PGQL-Agenten mit verschiedener Gewichtsinitialisierung

Dabei zeigte sich eine Annäherung des aufsummierten Rewards einer Runde gegen 0. Aus Gleichung 4.1 wird gefolgert, dass die getätigten Aktionen sehr gering sein müssen, d. h., dass sich das Fahrzeug nahezu nicht bewegt. Zur Analyse dieser Thematik wurden die Ausgaben des Actor-KNN zu verschiedenen Trainingszeitpunkten betrachtet. Mithilfe der

in den jeweiligen Trainingsepisoden gespeicherten Actor-Netzen wurden die Netzausgaben für Mittelwert und Standardabweichung bestimmt. Es wurden 1000 mögliche Positionen zwischen -1.2 und 0.5 mit zufälligen Geschwindigkeiten zwischen -0.07 und 0.07 auf einen Bereich von -1 bis 1 skaliert und anschließend als Netzeingaben verwendet. Die erhaltenen Netzausgaben wurden je Episode gemittelt und in Abbildung 4.7 dargestellt.

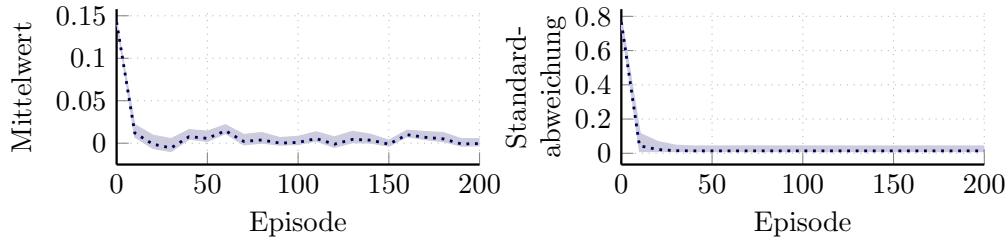


Abbildung 4.7: Ausgaben des Actor-KNN bei Gewichtsinitialisierung 1

Die Linien zeigen die gemittelten Werte über alle 1000 Positionen in der Episode, die eingefärbten Bereiche die vollständigen Wertebereiche. Die Vermutung, dass der Agent sich nahezu nicht bewegt, wird durch diese Abbildung bestätigt, da sich sowohl der Mittelwert als auch die Standardabweichung der Gaußfunktion zur Ermittlung der nächsten Aktion an 0 annähern. In der ersten Episode haben der prädizierte Mittelwert und die prädizierte Standardabweichung einen nahezu konstanten Wert von 0.1426 bzw. 0.7663. Da die Startpositionen nicht in der Mitte der möglichen Positionen liegen und entsprechend auch nach der Skalierung einen kleinen negativen Wert haben, entstehen durch die Initialisierung der Gewichte nahe 0 und der Verwendung von ReLU als Aktivierungsfunktion geringe Neuronenaktivierungen. Entsprechend hat die Startposition des Fahrzeugs in der ersten Episode aufgrund der Gewichtsinitialisierung kaum einen Einfluss auf die Netzausgaben. Die zugehörige Gaußverteilung für das Sampling der Actions zeigt die gepunktete Kurve in Abbildung 4.8.

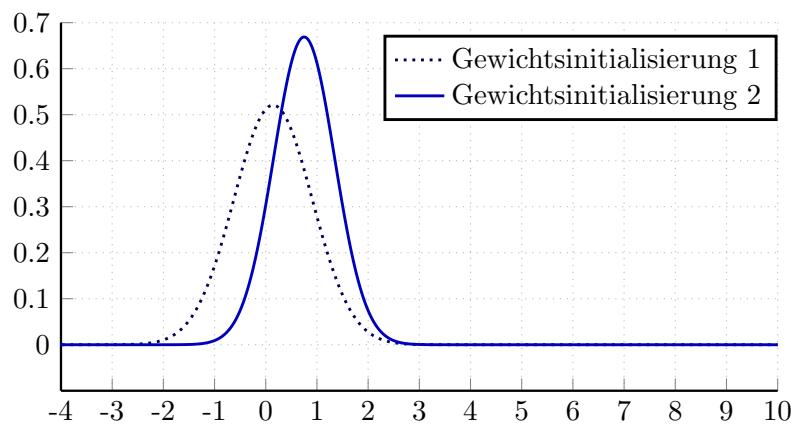


Abbildung 4.8: Gaußverteilung für das Sampling der Actions bei verschiedenen Gewichtsinitialisierungen

Aufgrund der Gewichtsinitialisierung hat die Position in der ersten Episode keinen Einfluss auf die Aktionswahl. Die durch zufälliges Sampling von der Verteilung erhaltenen Aktionen werden mit annähernd gleicher Wahrscheinlichkeit positiv oder negativ sein, weshalb sich das Fahrzeug nicht zwingend in aufeinanderfolgenden Schritten in dieselbe Richtung

bewegt. Infolgedessen ist es möglich, dass der Agent niemals das Ziel erreicht und niemals einen positiven Reward erhält.

Durch Verwendung einer anderen Gewichtsinitialisierung sollen zu Trainingsbeginn Aktionen in Richtung des Ziels, d. h. Aktionen mit positivem Vorzeichen, präferiert werden. In jeder Schicht werden die Gewichte zufällig aus einer Normalverteilung mit Mittelwert 1 und Standardabweichung 0.0001 gewählt, weshalb durch die Initialisierung eine Aktion von durchschnittlich 0.74 bevorzugt wird. Die durchgezogenen Linie in Abbildung 4.8 zeigt die zur ersten Episode mit Gewichtsinitialisierung 2 gehörende, durchschnittliche Gaußglocke. Der Mittelwert und die Standardabweichung ergeben sich als Mittelwert für 1000 getestete Positionen mit zufälligen Geschwindigkeiten im gesamten Zustandsraum. Analog zu Abbildung 4.7 visualisiert Abbildung 4.9 die Wertebereiche der Mittelwerte und Standardabweichungen, welche mit den in der jeweiligen Episode gespeicherten KNN für 1000 getestete Zustände prädiziert wurden. Die Initialisierung in Episode 0 erfolgt mit Gewichtsinitialisierung 2.

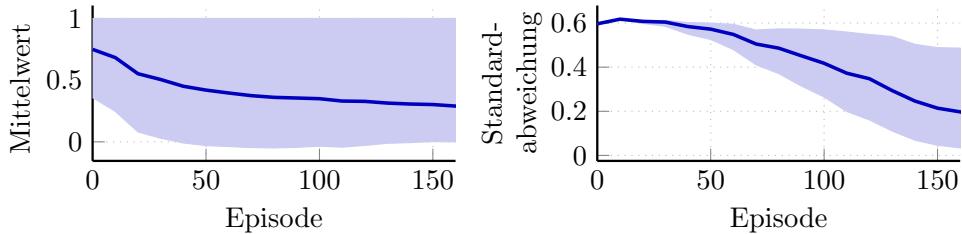


Abbildung 4.9: Ausgaben des Actor-KNN bei Gewichtsinitialisierung 2

Die Linien in Abbildung 4.9 kennzeichnen erneut den Mittelwert der identischen, getesteten Zustände in der jeweiligen Episode, die eingefärbten Bereiche die vollständigen Wertebereiche. Der linke Teil der Abbildung zeigt anhand des eingefärbten Bereiches deutlich, dass im Gegensatz zu Gewichtsinitialisierung 1 eine Abhängigkeit des prädizierten Mittelwertes von der jeweiligen Position besteht. Die rechte Kurve verdeutlicht, dass im Verlauf des Trainings das Wählen der Entscheidungen sicherer wird, da der Mittelwert der Standardabweichungen über die Episodenanzahl abnimmt.

Die Auswirkungen auf den Episodenreward verdeutlicht die blaue Kurve in Abbildung 4.6. Der Agent, welcher mit Gewichtsinitialisierung 2 startet, erreicht bereits ab Trainingsbeginn regelmäßig das Ziel, wie der positive Episodenreward zeigt. In erfolglosen Episoden sind die Rewards sehr niedrig, weshalb starke Sprünge im Rewardverlauf entstehen. Mit steigender Trainingsepisodenzahl werden die Unterschiede aufeinanderfolgender Rewards jedoch geringer. Grund für den nicht kontinuierlichen Verlauf des Rewards ist die verwendete stochastische Policy. Wie der rechte Teil von Abbildung 4.9 zeigt, ist die Standardabweichungen für einige Zustände sehr hoch, weshalb mit hoher Wahrscheinlichkeit eine von der besten Aktion abweichende gewählt wird.

Das Training mit Gewichtsinitialisierung 2 wurde nach 163 Episoden vorzeitig beendet, da in fünf aufeinanderfolgenden Episoden ein Reward über 90 erreicht wurde.

Training mit ACKTR

Wie bei den beiden vorherigen Algorithmen wird auch dieser Agent mit vorverarbeiteten Zuständen trainiert.

Bei ACKTR werden wie bei PGQL eine stochastische Policy und eine State-Value-Function erlernt, weshalb erneut die Netzwerkarchitekturen nach Abbildung 4.2b und Abbildung

4.3b verwendet werden. Die Neuronen in den versteckten Schichten werden ebenfalls auf $h_{1a} = h_{1c} = 20$ und $h_{2a} = h_{2c} = 10$ gesetzt. Die Lernrate des Actor-KNNs beträgt 10^{-4} und die des Critic-KNNs 10^{-3} . Der Hyperparameter γ zur Gewichtung der zukünftigen Rewards wird auf 0.99 und der Trust-Region-Parameter δ zur Beschränkung der Parameteränderungen auf 0.01 gesetzt. Die KNN-Parameter werden jeweils am Ende einer Episode geupdated.

Der ACKTR-Agent wurde zuerst ausgehend von der PyTorch-Standardinitialisierung in den vorderen und der Initialisierung mittels einer Normalverteilung $\mathcal{N}(\mu = 0, \sigma = 0.0001)$ in der letzten Schicht, trainiert. Basierend auf dieser Gewichtsinitialisierung findet bei diesem Algorithmus eine Veränderung der Netzparameter statt, sodass das Ziel erreicht wird. Infolgedessen ist keine abweichende Initialisierung notwendig, um Aktionen in Richtung des Ziels zu bevorzugen.

Abbildung 4.10 vergleicht exemplarisch die bis zur zehnten Runde erlernten Policies der beiden Agenten. Dazu werden zehn Zustände getestet, welche Tabelle 4.3 entnommen werden können. Neben einer Position auf dem linken Hügel wird eine Position im Startbereich, eine Position nah vor dem Ziel sowie zwei Positionen auf dem rechten Hügel untersucht. Dazu werden jeweils die maximale positive und die maximale negative Geschwindigkeit kombiniert. Der linke Teil von Abbildung 4.10 zeigt die prädizierten Mittelwerte und Standardabweichungen bezogen auf den jeweiligen Zustand, der rechte Teil zeigt die aus der Verteilung gesampelte Aktion.

Tabelle 4.3: Verglichene Zustände von ACKTR- und PGQL-Agent mit ReLU-Aktivierung

Zustand	1	2	3	4	5	6	7	8	9	10
Position	-1	-1	-0.6	-0.6	0	0	0.2	0.2	0.4	0.4
Geschwindigkeit	-0.07	0.07	-0.07	0.07	-0.07	0.07	-0.07	0.07	-0.07	0.07

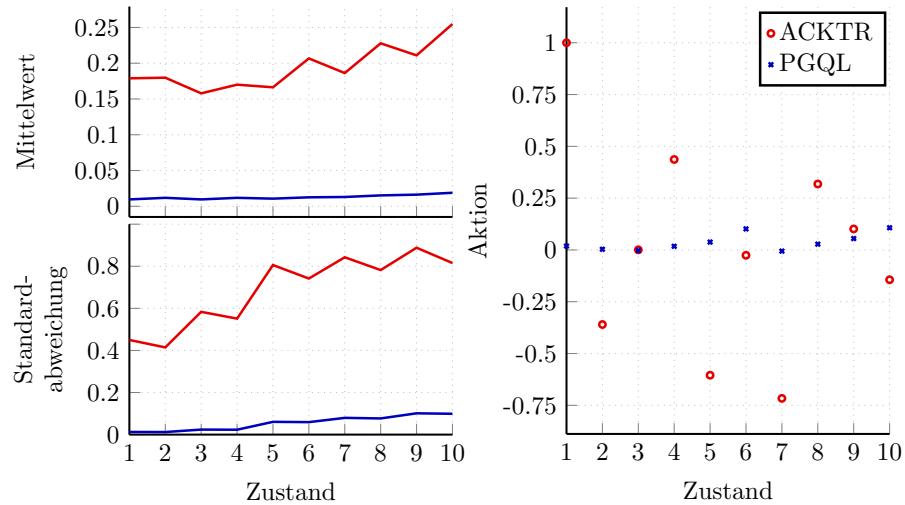


Abbildung 4.10: Vergleich von ACKTR- und PGQL-Agent mit ReLU-Aktivierung exemplarisch für 10 Zustände

Mit einem Mittelwert von durchschnittlich 0.013 und einer Standardabweichung von durchschnittlich 0.055 ergibt sich eine schmale Gaußglocke um 0 für die Aktionsauswahl des PGQL-Fahrzeugs, weshalb ausschließlich kleine Aktionen um 0 gewählt werden. Für den

ACKTR-Agenten zeigt sich eine deutlichere Varianz in den Mittelwerten und deutlich höhere Standardabweichungen, weshalb Aktionen aus nahezu dem gesamten Aktionsraum gewählt werden. Die durchschnittliche Standardabweichung von 0.687 zeigt, dass der Agent seine Umgebung noch umfangreich exploriert.

Bei Verwendung von ReLU (vgl. Abbildung 2.9d) als Aktivierungsfunktion wird aufgrund des Funktionsverlaufs potenziell die Hälfte der Gradienten sowie die Hälfte aller Neuronaktivierungen null. Da die Gewichte mit kleinen Werten um null initialisiert werden, sind ebenfalls die Neuronaktivierungen im positiven Bereich sehr gering. Da dies besonders bei der Berechnung der Fisher-Matrix ein Nachteil sein kann, wird in [53] stattdessen die Verwendung von ELU (vgl. Abbildung 2.9f) vorgeschlagen. Abweichend von Abbildung 4.2b und Abbildung 4.3b wird in dem nächsten Versuch ELU statt ReLU verwendet. Das Training wurde vorzeitig nach 63 Episoden beendet, da zuvor fünfmal in Folge ein Reward über 90 erzielt wurde. Abbildung 4.11 stellt die Ergebnisse mit den unterschiedlichen Aktivierungsfunktionen gegenüber.

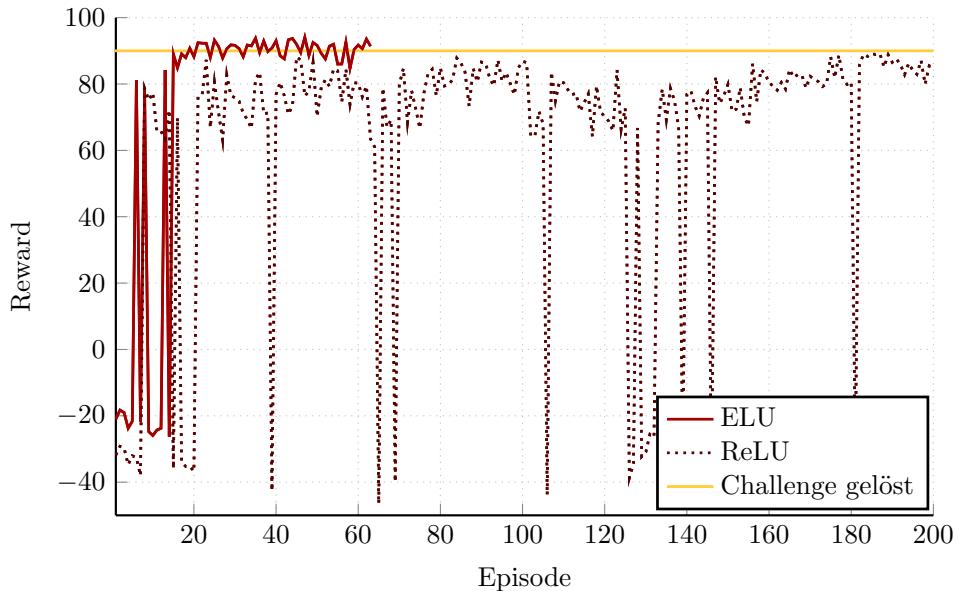


Abbildung 4.11: Vergleich von zwei ACKTR-Agenten mit verschiedenen Aktivierungsfunktionen

Die gepunktete Kurve zeigt den Verlauf des rundenweise aufsummierten Rewards für den ACKTR-Agenten, welcher ReLU als Aktivierungsfunktion in den versteckten Schichten verwendet. Bereits zum Beginn des Trainings erreicht der Agent regelmäßig einen positiven Reward und dementsprechend das Ziel. Während der aufsummierte Episodenreward mit fortschreitender Episodenanzahl tendenziell steigt, zeigen sich immer wieder starke Einbrüche. Der Agent, welcher ELU anstelle von ReLU verwendet, erhält in den ersten Episoden seltener einen positiven Reward. Hier sind ebenfalls starke Sprünge zu erkennen, welche jedoch ab Episode 15 verschwinden. Die Einbrüche des Rewards sind in beiden Fällen durch die Variation der Startpositionen sowie das Erlernen einer stochastischen Policy erklärbar. Gerade in frühen Trainingsphasen wird die mittels des Actor-KNN prädizierte Standardabweichung aufgrund der Initialisierung der Gewichte um 0 und der Verwendung von Softplus in der letzten Schicht mit einem Wert um 0.7 zu entsprechend zufällig gewählten Aktionen führen. Allgemein zeigt sich, dass der Agent unter Verwendung von

ELU deutlich erfolgreicher agiert und die CMC-Challenge bereits erstmals in Episode 21 lösen kann, wohingegen der ACKTR-Agent, welcher ReLU verwendet, im Trainingsverlauf häufiger das Ziel, jedoch nie mit einem Reward über 90 erreicht.

4.3 Vergleich der Ergebnisse und Fazit

Zum Abschluss des Kapitels sollen die drei Algorithmen miteinander verglichen werden. Dazu werden die jeweils aus den vergangenen Versuchen besten Agenten gegenübergestellt. Der DDPG-Agent wurde 111, der ACKTR-Agent 63 und der PGQL-Agent 164 Runden trainiert bevor das Training beendet wurde. Alle Agenten hatten im Training die gleichen Startpositionen, entsprechend der Trainingsdauer haben DDPG und PGQL jedoch potenziell mehr Startzustände als ACKTR gesehen.

Für den Vergleich wurden 100 Positionen im Startbereich in Kombination mit der Geschwindigkeit 0 als Anfangszustände gewählt. Diese Eingangsgrößen wurden auf einen Bereich zwischen -1 und 1 skaliert. Der von dem jeweiligen Agenten erzielte Episodenreward ist nachfolgend in Abbildung 4.12 über die dazugehörige Anfangsposition dargestellt.

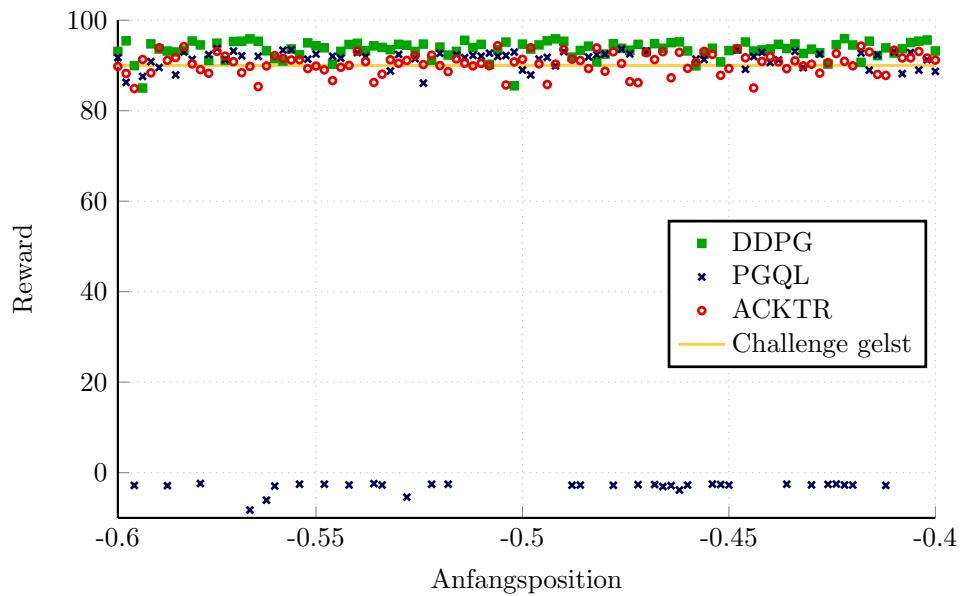


Abbildung 4.12: Vergleich der finalen Agenten für verschiedene Anfangszustände des CMC

Das Fahrzeug, welches mit PGQL trainiert wurde, erreicht nicht von allen Startpositionen ausgehend das Ziel, wobei sich jedoch keine Regelmäßigkeit zeigt. Eine Unterrepräsentation des Startbereichs kann ausgeschlossen werden, da sich keine Überschneidung mit den Ergebnissen der anderen beiden Agenten zeigt. Sowohl der DDPG-Agent als auch der ACKTR-Agent erreichen unabhängig vom Startzustand das Ziel. Tabelle 4.4 verdeutlicht, dass das mit DDPG trainierte Fahrzeug dabei minimal effizienter ist, da durchschnittlich ein höherer Reward erzielt wird.

Das mit PGQL trainierte Mountain Car erreicht einen deutlich geringeren durchschnittlichen Reward, da es nur in 67% der Versuche das Ziel erreicht. Wird ausschließlich der durchschnittliche Reward der erfolgreichen Versuche des PGQL-Agenten betrachtet, erzielt dieser einen Reward von 91.51 und ist somit ausgehend von diesen Startzuständen effizienter als der ACKTR-Agent.

Tabelle 4.4: Ergebnisse des Agenten-Tests beim Continuous Mountain Car

Algorithmus	Ziel erreicht	Reward > 90	durchschnittlicher Reward
DDPG	100/100	96/100	93.58
ACKTR	100/100	62/100	90.48
PGQL	67/100	52/67	60.24

In den vorherigen Versuchen mit dem Continuous Mountain Car zeigt sich, dass die Vorverarbeitung der Eingangsgrößen hilfreich ist. Agenten, welche mit DDPG oder ACKTR trainiert werden, scheinen weniger von der Gewichtsinitialisierung beeinträchtigt und entsprechend erfolgversprechender für kompliziertere Problemstellungen. Weiterhin sollte das Problem von seltenen Rewards genannt werden, welches beim CMC vorliegt, da der Agent nur beim Erreichen des Ziels einen positiven Reward bekommt. Das Erlernen einer guten Policy ist hierbei für On-Policy-Verfahren wie ACKTR und PGQL erschwert, da diese nicht wie Off-Policy-Verfahren erfolgreiche Erfahrungen wiederverwenden können. Prinzipiell ist zu vermuten, dass eine passende Hyperparameterwahl sowie infolgedessen ein erfolgreiches Training mittels PGQL schwieriger sein wird, da sich bereits im vorherigen Minimalbeispiel mit kleinen Zustands- und Aktionsraum Performanzdefizite zeigten.

5 Verwendung der Algorithmen zur Erweiterung eines Trajektorienfolgeregelungskonzeptes

Nachdem anhand des Minimalbeispiels erste Erkenntnisse über die ausgewählten Actor-Critic-Algorithmen gesammelt wurden, sollen diese im nachfolgenden Kapitel zur Erweiterung eines bestehenden Trajektorienfolgeregelungskonzeptes eingesetzt werden. Dazu wird zuerst der Aufbau der Simulationsumgebung sowie der Reinforcement Learning Agenten erläutert, bevor Simulationsergebnisse dargestellt und ausgewertet werden.

5.1 Aufbau

Nachfolgend soll die Fusion der Fahrzeugsimulation mit den RL-Agenten und der entsprechende Aufbau beschrieben werden. Prinzipiell existieren dabei zwei Hauptbestandteile, die Simulationsumgebung zur Fahrzeognachbildung und der Reinforcement Learning Agent. Im Rahmen dieser Arbeit wird ein Simulationsaufbau genutzt, welcher aus mehreren Komponenten besteht, die miteinander kommunizieren. Dazu zählt einerseits ein performanter Computer mit vier Nvidia GeForce GTX TITAN X Grafikkarten, auf welchem die Algorithmen trainiert und die Daten weiterverarbeitet werden können. Andererseits ist eine Micro-Autobox der Firma dSpace vorhanden, welche eine echtzeitfähige Recheneinheit darstellt, die zur prototypischen Funktionsentwicklung genutzt werden kann und den verwendeten Fahrzeugemulator abbildet. Durch die Verwendung dieses Echtzeitsteuergerätes wird ein Aufbau nah am tatsächlichen Fahrzeug gewährleistet. Die dadurch entstehenden Restriktionen, wie beispielsweise längere Simulationszeiten, werden akzeptiert, da die Algorithmen bezüglich ihrer Eignung für die Erweiterung des Trajektorienfolgeregelungskonzeptes eines realen Fahrzeugs evaluiert werden sollen.

5.1.1 Aufbau der Simulationsumgebung

Der Fahrzeugemulator wurde im Rahmen dieser Arbeit als „Blackbox“ angesehen. Für das bessere Verständnis wird nachfolgend jedoch kurz auf Grundlagen der Fahrzeugtechnik und auf die Bestandteile der Fahrzeugsimulationsumgebung eingegangen.

Grundlagen der Fahrdynamik

Die ausführlichen Grundlagen der Fahrzeugtechnik und Fahrdynamik sind umfangreich in der Literatur, wie beispielsweise in [8, 68], beschrieben. Nachfolgend wird daher lediglich auf Ausschnitte eingegangen, die das Verständnis der Arbeit erleichtern sollen.

Zur Betrachtung der Dynamik eines Fahrzeugs ist ein Koordinatensystem notwendig, in welchem wirkende Kräfte und Bewegungen betrachtet werden können. Abbildung 5.1 zeigt das Fahrzeugkoordinatensystem, bei welchem die positive x-Achse in Fahrtrichtung, die y-Achse in Fahrtrichtung nach links und die z-Achse nach oben zeigen. Während die Bewegungen entlang der Achsen auftreten, treten Wanken, Nicken und Gieren als Momente mit entsprechenden Winkeln um die x-, y- bzw. z-Achse des Fahrzeugs auf. Die Dynamik eines Fahrzeugs ist folglich durch drei translatorische und drei rotatorische Freiheitsgrade beschreibbar [68].

Der Schwimmwinkel β bezeichnet den Winkel zwischen dem Geschwindigkeitsvektor des Fahrzeugs und der Fahrzeulgängsachse. Bei geringen wirkenden Querbeschleunigungen ist dieser sehr gering. In fahrdynamischen Grenzbereichen können jedoch hohe Querbeschleunigungen und entsprechend größere Schwimmwinkel auftreten.

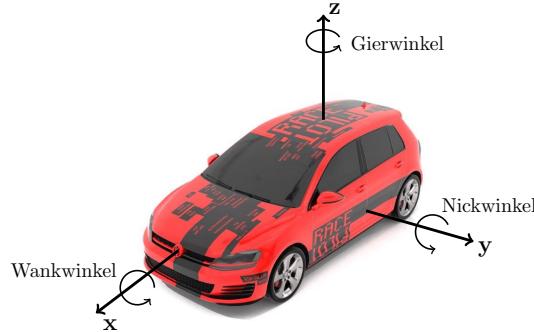


Abbildung 5.1: Fahrzeugkoordinatensystem angelehnt an [8, 71]

Ein Gierwinkel ψ entsteht durch Drehung des Fahrzeugs um seine z-Achse und bezeichnet entsprechend den Winkel zwischen der x-Achse des Fahrzeugs und der x-Achse des ortsfesten Koordinatensystems. Bei einer Kreisfahrt um den Ursprung dieses Koordinaten- systems ändert sich der Gierwinkel fortlaufend. Deswegen wird vorwiegend die Änderung des Winkels, d. h. die Gierwinkelgeschwindigkeit $\dot{\psi}$ sowie deren Änderung, die Gierwinkelbeschleunigung $\ddot{\psi}$, zur Beschreibung des Fahrzustands verwendet.

Um Fahrzeugsysteme auslegen zu können, ist eine mathematische Beschreibung des Fahrzeugs notwendig. Das lineare Einspurmodell von Rieckert und Schunck [69] ist das vermutlich bekannteste, vereinfachte Modell, welches bereits 1940 veröffentlicht wurde. Beim linearen Einspurmodell wird der Fahrzeugschwerpunkt auf Fahrbahnhöhe verlagert, weshalb Wank- und Hubbewegungen nicht berücksichtigt werden müssen. Infolgedessen können Geschwindigkeiten und Beschleunigungen nur in Längs- und in Querrichtung auftreten. Das entsprechend vereinfachte Fahrzeugmodell umfasst folglich nur noch zwei translatorische und einen rotatorischen Freiheitsgrad. Außerdem wird vorausgesetzt, dass das System linear ist [8]. Aufgrund der getroffenen Vereinfachungen ist das Modell lediglich in Normalsituationen zur Beschreibung des Fahrzeugs geeignet. Das nichtlineare Einspurmodell ist eine Erweiterung des linearen Einspurmodells, welches im Gegensatz zu diesem auch in Grenzbereichen, d. h. bei höheren auftretenden Querbeschleunigungen, verwendet werden kann.

Zur Beschreibung einer Strecke kann beispielsweise die Kurvenkrümmung κ verwendet werden. Diese entspricht dem Kehrwert des Krümmungsradius der Kurve. Je enger die Kurve, desto kleiner der Kurvenradius und entsprechend desto größer die Kurvenkrümmung. Da eine gerade Strecke als Kurve mit unendlich großem Radius angesehen werden kann, ist die Kurvenkrümmung folglich 0.

Simulationsumgebung

Die verwendete Simulationsumgebung beinhaltet neben der Bahnplanung einen Fahrzustandsbeobachter, einen Fahrdynamikregler und ein Fahrzeugmodell. Der Aufbau des Systems ist in Abbildung 5.2 zu sehen.

Mithilfe der gegebenen Karte ermittelt der Bahnplaner eine Trajektorie, welche anschließend als Sollkurs für das Fahrzeug auf der gegebenen Strecke dient. Basierend auf dem Sollkurs, dem Istkurs und den aktuellen Zustandsgrößen bereitet der Fahrdynamikbeobachter die Zustandsgrößen auf. Mithilfe von Zustandsbeobachtern können auf Basis von Messgrößen und einem Modell nicht-messbare Größen geschätzt werden [68]. Die durch

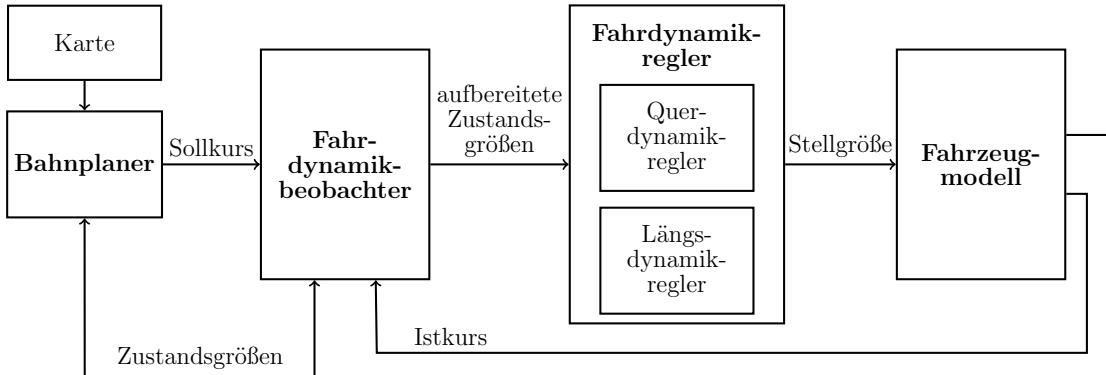


Abbildung 5.2: Simulationsumgebung zur automatisierten Fahrzeugführung nach [9]

den Beobachter aufbereiteten Zustandsgrößen des Fahrzeugs dienen als Eingangsgrößen für die Regelung. Der Fahrdynamikregler ist in einen längs- und einen querregelnden Anteil separierbar. Da der Fokus dieser Arbeit auf die Querdynamikregelung gelegt wird, sind die zugehörigen Komponenten in Abbildung 5.3 zu sehen. Als Vorsteuerung wird die invertierte Regelstrecke, d. h. ein invertiertes Einspurmodell, verwendet. Neben der Kurvenkrümmung der Trajektorie κ^* und der Ist-Geschwindigkeit des Fahrzeugs v_x dienen die Fahrzeugparameter für das Einspurmodell Θ hierbei als Eingangsgrößen. Um Ungenauigkeiten dieses Modells und äußere Störgrößen zu kompensieren, wird zusätzlich ein PID-Regler verwendet, welcher einen Lenkwinkel basierend auf der Abweichung zwischen Soll- und Isttrajektorie sowie dem Gierwinkelfehler ermittelt.

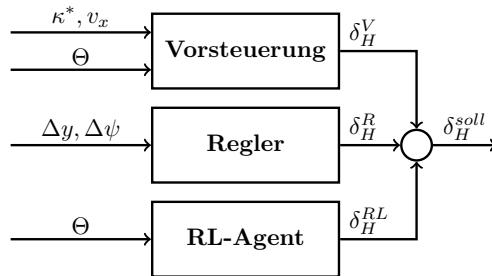


Abbildung 5.3: Komponenten des Quer-dynamikregelungskonzeptes nach [9]

Vermehrt finden in der Regelungstechnik auch adaptive Systeme Anwendung, welche basierend auf Messdaten Zusammenhänge abbilden können, die in den Fahrzeugmodellen schwierig darstellbar sind [12, 70]. Zusätzlich können diese datengetriebenen Methoden eingesetzt werden, um Veränderungen der Modellparameter und damit einhergehend die sich verschlechternde Güte der Regelung bestehend aus Vorsteuerung und Regler auszugleichen. In der aktuellen Anwendung soll hierfür ein Reinforcement Learning Agent trainiert werden, welcher anhand von verschiedenen Fahrzustands- und Streckeninformationen einen additiven Lenkwinkel stellt.

Als Fahrzeugmodell wird ein nichtlineares Einspurmodell verwendet, anhand welchem der nächste Fahrzeugzustand berechnet wird.

5.1.2 Aufbau des RL-Agenten

Nachfolgend soll der Aufbau der Implementierung des Reinforcement Learning Agenten erläutert werden. Um die Echtzeitfähigkeit des Systems zu wahren, müssen der Prädiktions- und der Lernteil in verschiedene Python-Prozesse aufgeteilt werden. Infolgedessen wird gewährleistet, dass der RL-Agent innerhalb von 5 ms einen Zustand erhält, basierend auf diesem eine Aktion prädiziert und diese an den Fahrzeugemulator zur Ausführung übermittelt. Der entsprechende prinzipielle Aufbau ist Abbildung 5.4 zu entnehmen.

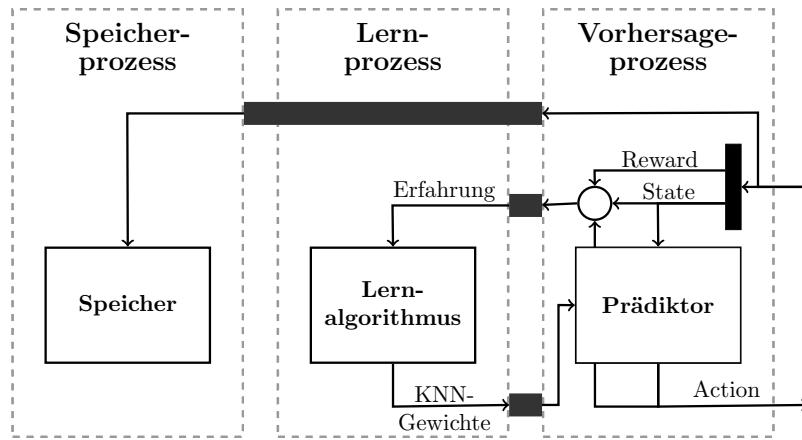


Abbildung 5.4: Vereinfachte Darstellung der Prozesse des RL-Agenten

Neben den notwendigen Prozessen für die Prädiktion der nächsten Aktion basierend auf dem aktuellen Zustand und dem Lernprozess zum Anpassen der Parameter der künstlichen neuronalen Netze existiert ein dritter Prozess, in welchem aktuelle Informationen bezüglich des Fahrzeugzustands für die nachträgliche Analyse des Trainings gespeichert werden. Der RL-Agent erhält vom Echtzeitsteuergerät eine Nachricht, welche neben dem aktuellen Zustand den Reward sowie weitere fahrzeugspezifische Informationen enthält. Basierend auf dem gesendeten Datenpaket extrahiert der erste Prozess den Agenten-State, um anhand dessen die nächste Aktion zu prädizieren, welche an das Echtzeitsteuergerät zurückgesendet wird.

Während mit Übertragung des Zustands, Prädiktion der Aktion sowie dem Zurücksenden dieser an den Fahrzeugemulator die Anforderung an eine Dauer unter 5 ms erfüllt werden kann, benötigt der zweite Prozess, welcher von Prozess 1 den Zustand, den Reward sowie die Aktion erhält, deutlich mehr Zeit. In diesem sogenannten Lernprozess werden die zusammengehörigen Zustände, Aktionen sowie Rewards mit zusätzlichen algorithmenspezifischen Daten zu den entsprechenden Erfahrungstupeln kombiniert. Mit dem Lernalgorithmus werden basierend auf den Erfahrungen neue KNN-Gewichte bestimmt, welche vom zweiten Prozess an den ersten übermittelt werden. Je nach Lernalgorithmus sind für dieses Vorgehen zwischen durchschnittlich 0.02 s bei DDPG und 1.9 s bei ACKTR notwendig. Der deutliche Unterschied ist zum einen durch die Anzahl der gerechneten Erfahrungen begründbar. Während bei DDPG ein Batch von 64 Erfahrungen verwendet wird, nutzt ACKTR die Erfahrungen einer gesamten Runde, welche mit ungefähr 10000 Samples rund 150mal höher ist. Zum anderen ist der Rechenaufwand bei ACKTR aufgrund der Berechnung der Fisher-Matrix deutlich komplexer.

Der dritte Prozess erhält die gesamte Nachricht des Echtzeitsteuergerätes und speichert die enthaltenen Informationen, um die anschließende Analyse des Trainings zu ermöglichen.

5.2 Einbindung des RL-Agenten in die Simulationsumgebung

In diesem Abschnitt soll zuerst auf den Zustands- und den Aktionsraum eingegangen werden, welche dem Agenten zur Verfügung stehen, bevor im Anschluss die verwendete Rewardmetrik erläutert wird.

Zustandsraum

Zur Berechnung der additiven Regelanteile des RL-Agenten stehen verschiedene kontinuierliche Fahrzeugzustandsgrößen zur Verfügung, welche anschließend erläutert werden sollen. Die Symbole inklusive einer kurzen Beschreibung und der für die Datenvorverarbeitung verwendeten Wertebereiche sind Tabelle 5.1 zu entnehmen. Bei v_x und v_y handelt es sich um Zustandsgrößen, welche die aktuelle Geschwindigkeit des Fahrzeugs im Schwerpunkt in Längs- bzw. Querrichtung beschreiben. Die ebenfalls in Längs- und Queranteil separierte Beschleunigung des Fahrzeugs stammt aus dem Beobachter, in welchem das Signal gefiltert wurde. Der Schwimmwinkel beschreibt den Winkel zwischen dem Fahrzeuggeschwindigkeitsvektor und der -längsachse. Der Gierwinkel beschreibt die rotatorische Bewegung des Fahrzeugs um die Hochachse des Fahrzeugkoordinatensystems. Da zur Abbildung der Dynamik die zeitliche Ableitung der Winkelgröße von größerem Interesse ist, werden stattdessen die Gierwinkelgeschwindigkeit und die Gierwinkelbeschleunigung zur Beschreibung des Fahrzustands verwendet. Die Trajektorie des Bahnplanungsalgorithmus beinhaltet die gewünschten Zustandsgrößen in Form der Geschwindigkeiten und Beschleunigungen in Fahrzeuglängsrichtung sowie die Kurvenkrümmung für verschiedene Referenzzeitpunkte entlang der Strecke. Als weitere Eingangsgröße kann der aktuelle Lenkwinkel verwendet werden. Weiterhin stehen verschiedene Fehlermaße zur Verfügung, welche zum Training der RL-Agenten genutzt werden können. Die Querablage e_y beschreibt die Abweichung von Soll- zu Isttrajektorie. Durch Verwendung des Geschwindigkeits- und des Beschleunigungsfehlers können Abweichungen von den Ist- zu den gewünschten Sollwerten einbezogen werden. Der Gierwinkelfehler beschreibt die Abweichung in der Ausrichtung des Fahrzeugs.

Aktionsraum

Die Kommunikation zwischen Echtzeitsteuergerät und RL-Agent ist neben der Querregelung ebenfalls auf die Längsregelung ausgelegt. Entsprechend stellt Tabelle 5.2 neben dem Wertebereich für den Stellwinkel ebenfalls die Wertebereiche für die Brems- und Gaspedalstellung dar.

Zur Erweiterung der Querdynamikregelung erlernt der Agent einen kontinuierlichen Stellwinkel, welcher einen beliebigen Wert zwischen $\pm 4.5 \text{ rad} \approx \pm 258^\circ$ annehmen kann. Der Gesamtstellwinkel bestehend aus den Anteilen der Vorsteuerung, der Regelung und des RL-Agenten wird anschließend gemäß der Übersetzung in den entsprechenden Lenkwinkel umgerechnet. Da das tatsächliche Stellsignal für das Brems- bzw. Gaspedal durch Addition mit dem Anteil des Reglers und der Vorsteuerung gebildet wird, ist es möglich "negatives Bremspedal" bzw. "negatives Gaspedal" zu stellen, um die anderen Regelanteile auszugleichen.

Rewardmetrik

Der laterale Reward, den der RL-Agent zum Erlernen eines Stellwinkels verwendet, setzt sich aus zwei Anteilen zusammen. Da der RL-Agent Ungenauigkeiten im Fahrzeugmodell ausgleichen soll, welche sonst durch den Regler ausgeglichen werden, wird unter anderem das Stellsignal des Lenkwinkelreglers in die Berechnung des lateralen Rewards einbezogen.

Tabelle 5.1: Verfügbare Fahrzeuggrößen zum Erlernen des Lenkwinkels, der Gaspedal- so- wie der Bremspedalstellung in der Fahrzeugsimulation

Symbol	Beschreibung	Minimum	Maximum
v_x	Geschwindigkeit in x-Richtung	0 m/s	40 m/s
v_y	Geschwindigkeit in y-Richtung	-3 m/s	3 m/s
β	Schwimmwinkel	-0.1745 rad	0.1745 rad
$\dot{\psi}$	Gierwinkelgeschwindigkeit	-0.7 rad/s	0.7 rad/s
$\ddot{\psi}$	Gierwinkelbeschleunigung	-1.3 rad/s ²	1.3rad/s ²
$a_{y,beo}$	Beschleunigung in y-Richtung	-10 m/s ²	10m/s ²
$a_{x,beo}$	Beschleunigung in x-Richtung	-11 m/s ²	11 m/s ²
κ_0	Kurvenkrümmung zum Referenzzeitpunkt 0	-0.1818 m ⁻¹	0.1818 m ⁻¹
κ_{50}	Kurvenkrümmung zum Referenzzeitpunkt 50	-0.1818 m ⁻¹	0.1818 m ⁻¹
κ_{105}	Kurvenkrümmung zum Referenzzeitpunkt 105	-0.1818 m ⁻¹	0.1818 m ⁻¹
κ_{120}	Kurvenkrümmung zum Referenzzeitpunkt 120	-0.1818 m ⁻¹	0.1818 m ⁻¹
κ_{200}	Kurvenkrümmung zum Referenzzeitpunkt 200	-0.1818 m ⁻¹	0.1818 m ⁻¹
v_0	Geschwindigkeit zum Referenzzeitpunkt 0	0 m/s	40 m/s
v_{50}	Geschwindigkeit zum Referenzzeitpunkt 50	0 m/s	40 m/s
v_{105}	Geschwindigkeit zum Referenzzeitpunkt 105	0 m/s	40 m/s
v_{120}	Geschwindigkeit zum Referenzzeitpunkt 120	0 m/s	40 m/s
v_{200}	Geschwindigkeit zum Referenzzeitpunkt 200	0 m/s	40 m/s
a_0	Beschleunigung zum Referenzzeitpunkt 0	-11 m/s ²	11 m/s ²
a_{50}	Beschleunigung zum Referenzzeitpunkt 50	-11 m/s ²	11 m/s ²
a_{105}	Beschleunigung zum Referenzzeitpunkt 105	-11 m/s ²	11 m/s ²
a_{120}	Beschleunigung zum Referenzzeitpunkt 120	-11 m/s ²	11 m/s ²
a_{200}	Beschleunigung zum Referenzzeitpunkt 200	-11 m/s ²	11 m/s ²
δ_H^{soll}	Lenkwinkel	-200°	200°
e_y	Querablage	-	-
e_ψ	Gierwinkelfehler	-	-
e_{vx}	Geschwindigkeitsfehler	-	-
e_{ax}	Beschleunigungsfehler	-	-

Tabelle 5.2: Erlernbare Actions der RL-Agenten

Symbol	Beschreibung	Minimum	Maximum
δ_H^{RL}	Stellwinkel	-4.5 rad	4.5 rad
$pedal_b$	Bremspedalstellung	-1	1
$pedal_g$	Gaspedalstellung	-100	100

Die zweite eingehende Komponente ist die Abweichung zwischen Soll- und Isttrajektorie des Fahrzeugs, da dieses möglichst wenig von der geplanten Linie des Bahnplaners abweichen soll.

chen soll.

Abbildung 5.5 zeigt die um -1 verschobene Gaußverteilung mit Mittelwert 0 und Standardabweichung 0.2, welche zur Bestimmung des Rewardanteils der Querablage verwendet wird.

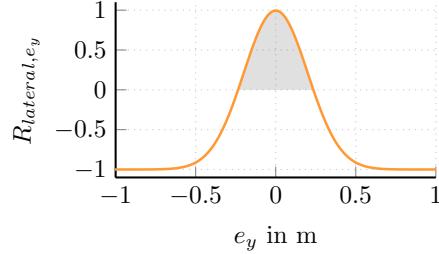


Abbildung 5.5: Gaußkurve zur Bestimmung des Rewardanteils der Querablage

Der grau eingefärbte Bereich in Abbildung 5.5 kennzeichnet die tolerierte Abweichung von der Solltrajektorie, bei welcher der Agent entsprechend eine positive Rückmeldung erhalten wird. Als weiterer Anteil in der Rewardmetrik wird der betragsmäßige Regelanteil in Grad berücksichtigt, der mit einem Faktor von 0.01 als Bestrafung eingeht. Die sich daraus ergebende Gleichung für den lateralen Reward lautet:

$$\begin{aligned} R_{lateral} &= R_{lateral,e_y} + R_{lateral,Regler} \\ &= \frac{1}{\sqrt{2\pi\sigma^2}} \exp\left(-\frac{(e_y - \mu)^2}{2\sigma^2}\right) - 1 + (-0.01) \left| \frac{180}{\pi} \delta_H^R \right| \end{aligned} \quad (5.1)$$

mit $\mu = 0$ und $\sigma = 0.2$.

Hyperparameter

Wie bereits beim CMC wurden zwei getrennte fully-connected KNN mit zwei versteckten Schichten zur Repräsentation des Actors und der Critic verwendet. Die Netzwerke zum Abbilden der Policies wurden entsprechend Abbildung 4.2 und die Netzwerke zum Abbilden der Value-Functions nach Abbildung 4.3 aufgebaut. Die Neuronenzahlungen wurden für beide KNN identisch gewählt. Die in den nachfolgenden Versuchen verwendeten Neuronenzusammenstellungen mit den entsprechenden Bezeichnungen sind Tabelle 5.3 zu entnehmen.

Tabelle 5.3: Verwendete Neuronenzahlungen in der Simulation

Bezeichnung	h_{1a}	h_{2a}	h_{1c}	h_{2c}
Architektur 1	200	100	200	100
Architektur 2	100	50	100	50
Architektur 3	30	20	30	20

DDPG verwendet die Netzwerke wie in Abbildung 4.2a und Abbildung 4.3a dargestellt und entsprechend ReLU als Aktivierungsfunktion in den versteckten Schichten. In der letzten Schicht des Actors wird der Tangenshyperbolicus und in der Critic die Identitätsfunktion verwendet. Um die Stabilität und Sicherheit des Fahrzeugs zu gewährleisten, sollen anfangs Aktionen möglichst um null gewählt werden. Da DDPG den Tangenshyperbolicus in der letzten Schicht verwendet, wird die Netzausgabe mit 4.5 multipliziert, um den gesamten

möglichen Bereich des Stellwinkels abzubilden. Durch eine zufällige Initialisierung der Gewichte aus einer Gaußverteilung mit Mittelwert 0 und Standardabweichung 0.001 soll zu Beginn ein niedriger Winkel gestellt werden. Die Exploration des Agenten wird gewährleistet, da der externe OU-Rauschprozess eine Variation der Stellwinkel abweichend von null garantiert. Die anderen Gewichte werden entsprechend der PyTorch-Standardinitialisierung gesetzt.

Die Initialisierung der Agenten, welche eine stochastische Policy erlernen, ist komplizierter, da durch die Initialisierung sowohl die Sicherheit des Fahrzeugsystems als auch die Exploration garantiert werden muss. PGQL nutzt die Netzwerke nach Abbildung 4.2b und Abbildung 4.3b, ACKTR verwendet die abgewandelten Netzwerkarchitekturen nach Abbildung 4.2b und Abbildung 4.3b mit ELU statt ReLU. Beide Actor-Netzwerke für stochastische Policies werden in den vorderen Schichten zur Berechnung des Mittelwertes sowie in allen Schichten zur Berechnung der Standardabweichung mittels der PyTorch-Standardinitialisierung gesetzt. Die letzte Schicht zur Berechnung des Mittelwertes wird abweichend mit Gewichten aus einer Gaußverteilung $\mathcal{N}(\mu = 0, \sigma = 0.001)$ initialisiert. Der Anfangswert (random seed) des Zufallsgenerators wird in jedem Versuch identisch gewählt, um die Vergleichbarkeit weitestgehend zu gewährleisten.

Jeder Agent wird 120 Runden trainiert, da diese Anzahl als guter Kompromiss zwischen benötigter Simulationsdauer und Erkennbarkeit der Performanz-Tendenzen angesehen wird.

5.3 Referenzfahrt

Mithilfe der trainierten RL-Agenten soll ein vorhandenes Trajektorienfolgeregelungskonzept bestehend aus einer Vorsteuerung und einem Regler erweitert werden. Nachfolgend soll dieses Regelungssystem ohne RL-Agenten betrachtet werden, um die ermittelten Messgrößen als Referenz für die anschließenden Versuche zu verwenden. Alle Fahrversuche in der Simulation wurden mit einem Dynamikfaktor von 0.8 durchgeführt. Bei dem Dynamikfaktor handelt es sich um eine Möglichkeit, mit welcher das planungsseitig vorgegebene Geschwindigkeits- sowie Beschleunigungsprofil verändert werden kann, um infolgedessen auch Fahrten außerhalb des fahrdynamischen Grenzbereichs umzusetzen. Dabei wird das vorgegebene Geschwindigkeitsprofil linear und das Beschleunigungsprofil quadratisch mittels des Dynamikfaktors skaliert. Abbildung 5.6 zeigt die insgesamt rund 1070 Meter lange Strecke, welche das Fahrzeug in allen Experimenten befährt. Diese bietet ein abwechslungsreiches Streckenprofil, da neben Geraden verschiedene Links- und Rechtskurven mit unterschiedlichen Krümmungen existieren. Der rote Punkt kennzeichnet den Rundenstartpunkt, welcher folglich dem Streckenmeter 0 entspricht.

Die Farbgebung der Strecke in der Abbildung kennzeichnet die durchschnittliche gefahrene Geschwindigkeit in diesem Teilstück. Je dunkler der Streckenabschnitt, desto höher die Fahrzeuggeschwindigkeit. Das Fahrzeug erreicht seine Höchstgeschwindigkeit von rund $113 \frac{\text{km}}{\text{h}}$ nach dem geraden Streckenabschnitt bei Streckenmeter 320.

Zur Beurteilung der nachfolgend trainierten RL-Agenten werden neben der Querablage auch die Regleraktivität des Lenkwinkelreglers und der laterale Reward im Allgemeinen betrachtet. Zur Ermittlung der Referenzwerte fährt das Fahrzeug zehn Runden, welche anschließend gemittelt werden. Abbildung 5.7a zeigt den mittleren Fehler zwischen Soll- und Isttrajektorie, welcher im Mittel rund 17 cm beträgt und somit nur gering von der gewünschten Solltrajektorie abweicht.

Wird neben der gemittelten Querablage die absolute Abweichung von der Referenzlinie exemplarisch für Runde 7 betrachtet, treten maximale Abweichungen von bis zu 44.8 cm auf. Überwiegend zeigt sich jedoch eine Abweichung zwischen Soll- und Isttrajektorie, die

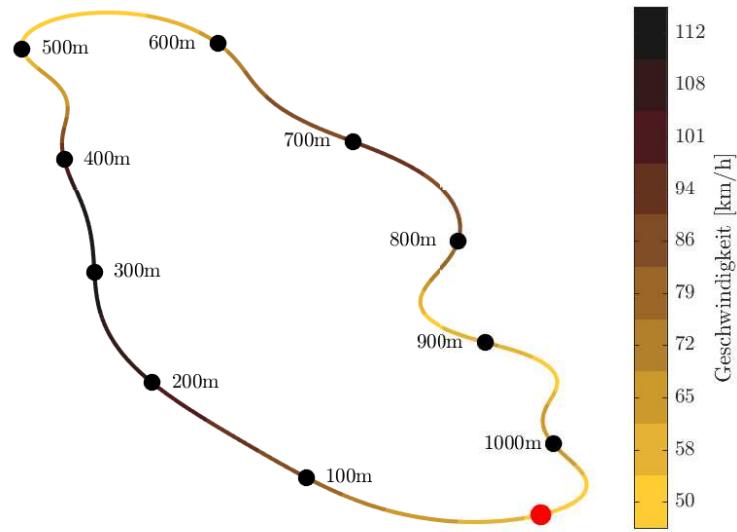


Abbildung 5.6: Gefahrene Strecke in der Fahrzeugsimulation

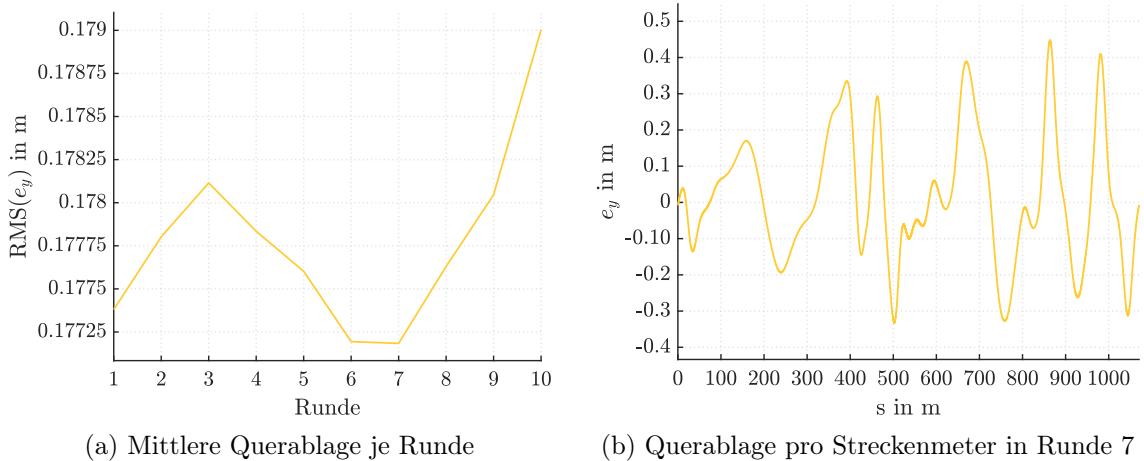


Abbildung 5.7: Querablage bei der Referenzfahrt

betragsmäßig unter der tolerierten Querablage von 0.233 m liegt.

Als weiterer Einfluss auf den lateralen Reward ist der Stellwinkel zu betrachten. Abbildung 5.8 zeigt den Verlauf des Stellwinkelsignals des Reglers δ_H^R , der Vorsteuerung δ_H^V sowie des Gesamtkonzeptes δ_H^{ges} über die Streckenmeter exemplarisch für Runde 7.

Dabei wird deutlich, dass das gegebene invertierte nicht-lineare Einspurmodell als Vorsteuerung das Fahrzeugmodell der Simulationsumgebung gut abbildet und entsprechend wenig Aktivität des Reglers benötigt wird. Dieser stellt bei Streckenmeter 860 den maximalen positiven Winkel von 27.41° und den maximalen negativen Winkel von -29.19° bei Streckenmeter 1042, welcher in der letzten Kurve des Kurses liegt. Insgesamt ergibt sich in dieser Runde ein durchschnittlicher Stellwinkel von rund $11.11^\circ \approx 0.194$ rad. Der RMS des Reglerstellwinkels, welcher als Referenz für die nachfolgenden Versuche verwendet wird, beträgt 0.1947 rad.

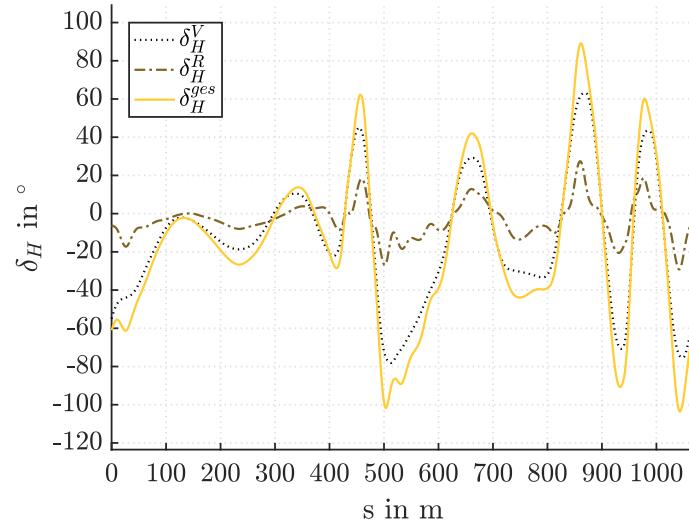


Abbildung 5.8: Stellwinkel in Runde 7 der Referenzfahrt

Durch die Varianz im quadratischen Mittel des Stellwinkels des Reglers und die Variation in der Querablage ergeben sich entsprechend auch unterschiedliche laterale Rewards, die in Abbildung 5.9 zu sehen sind. Der mittlere Reward über die zehn gefahrenen Runden, welcher als Referenz in den nachfolgenden Versuchen verwendet wird, beträgt $R_{lateral} = 4357.1$.

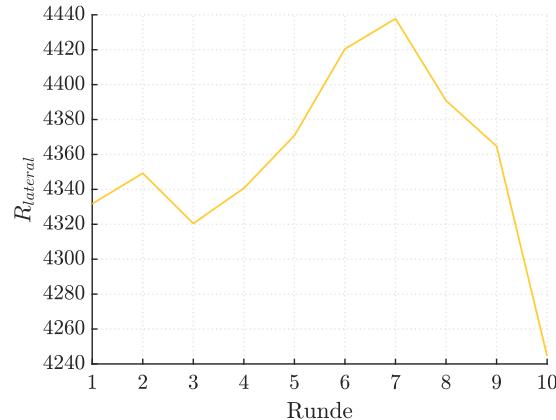


Abbildung 5.9: Lateraler Reward bei der Referenzfahrt

5.4 Simulationsergebnisse

Basierend auf den nachfolgenden Untersuchungen soll ein Überblick über die Eignung der drei Actor-Critic-Algorithmen für das Erlernen eines RL-Agenten zur Erweiterung eines bestehenden Trajektorienfolgeregelungskonzeptes gegeben werden. Mithilfe des zusätzlich gestellten Lenkwinkels soll die Querführung eines Fahrzeugs verbessert werden. Die in Kapitel 5.3 ermittelten Messgrößen dienen als Referenz für die durchgeführten Versuche sowie als Grundlage für die Bewertung der Eignung der jeweiligen Agenten und entsprechenden Actor-Critic-Verfahren.

Training mit DDPG

Im ersten simulierten Fahrversuch wurde der DDPG-Agent mit allen verfügbaren Fahrzustands- und Fehlergrößen sowie den Streckeninformationen aus Tabelle 5.1 trainiert. Zur Abbildung der Policy und der State-Action-Value-Function wurden die drei, in Abschnitt 5.2 erläuterten, Netzarchitekturen verwendet. Das für die Exploration des Zustandsraum bei DDPG notwendige externe Rauschen wird in der ersten Runde komplett zu der vom Actor-KNN prädizierten Aktion addiert. Anschließend wird dieses nach Gleichung 4.2 bis einschließlich Runde 80 reduziert. Zur Evaluierung des Lernfortschritts wird in jeder fünften Runde kein Rauschen hinzugefügt, um damit eine Vergleichbarkeit der Ergebnisse ohne einen zu hohen Zusatz an zusätzlicher Simulationszeit zu generieren. Entsprechend ist in den nachfolgenden Abbildungen vor Runde 80 der zugehörige Wert jeder fünften und anschließend jeder Runde dargestellt.

Abbildung 5.10 zeigt den über die jeweilige Runde aufsummierten Reward für die drei verschiedenen Netzwerkkonfigurationen.

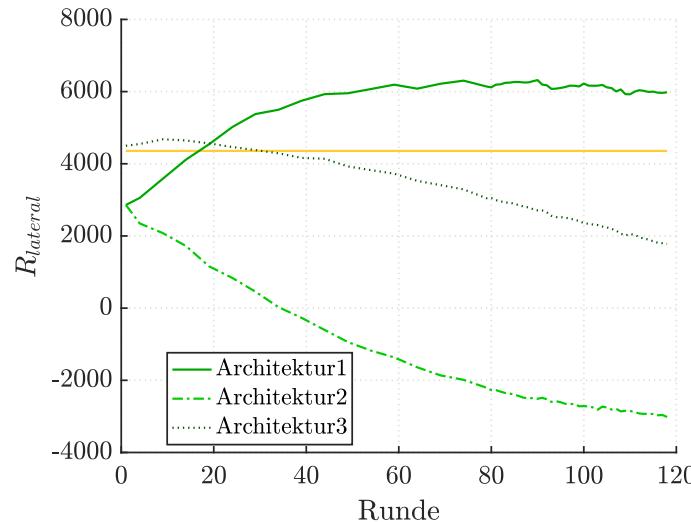


Abbildung 5.10: Lateraler Reward von drei DDPG-Agenten mit allen nach Tabelle 5.1 verfügbaren Eingangsgrößen

Hierbei zeigt sich, dass ausschließlich die Regelung inklusive des RL-Agenten mit 200 Neuronen in der ersten und 100 Neuronen in der zweiten versteckten Schicht einen höheren Reward als die Referenzfahrzeugregelung erzielt. Bei dem Trainingsende in Runde 120 erreicht diese mit einem Wert von 5980.7 eine prozentuale Verbesserung des lateralen Rewards von 37.26%. Bei den RL-Agenten mit kleineren neuronalen Netzen ist eine kontinuierliche Verschlechterung der Performanz deutlich zu erkennen. Der in der letzten Trainingsepisode

erhaltene Reward ist für die beiden Architekturen um annähernd 170% bzw. 59% schlechter als der Referenzwert.

Da die Kombination aus Querablage und Aktivität des Reglers ausschlaggebend für den letztendlich erhaltenen Reward des Agenten sind, sollen in Abbildung 5.11 diese Anteile getrennt voneinander betrachtet werden.

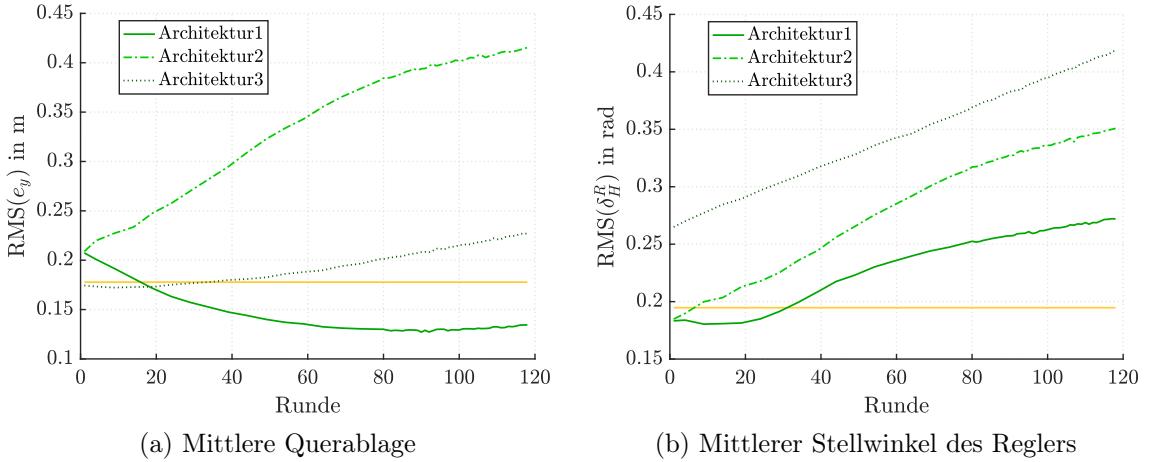


Abbildung 5.11: Rewardanteile von drei DDPG-Agenten mit allen nach Tabelle 5.1 verfügbaren Eingangsgrößen

Abbildung 5.11a vergleicht die gemittelte Abweichung zwischen Soll- und Isttrajektorie der drei DDPG-Agenten mit verschiedenen Netzarchitekturen. Hier ist deutlich zu sehen, dass mit den größtem KNN mit 200 Neuronen in der ersten und 100 Neuronen in der zweiten versteckten Schicht die Querablage signifikant minimiert werden konnte. Die Fahrzeuregellungen, die mittels des RL-Agenten mit 100 bzw. 30 Neuronen in der ersten und 50 bzw. 20 Neuronen in der zweiten versteckten Schicht erweitert wurden, entfernen sich im Trainingsverlauf von der Solltrajektorie.

Die Agenten mit Architektur 1 und Architektur 2 starten trotz zufälliger Initialisierung in Runde 1 mit einem ähnlichen RMS der Querablage von rund 21 cm. Während diese mit Architektur 1 bis Runde 200 auf 13.44 cm reduziert werden kann, verdoppelt sich diese bei Architektur 2 mit 41.6 cm nahezu. Das Fahrzeug mit dem RL-Agenten, welches seine Policy und Value-Function mit Netzwerken mit je 30 Neuronen in der ersten und 20 Neuronen in der zweiten versteckten Schicht bestimmt, trifft in der ersten Runde im Mittel die Solltrajektorie besser als das Referenzfahrzeug, wie die mittlere Querablage unter dem Referenzwert verdeutlicht. Mit zunehmender Rundenzahl steigt der RMS der Querablage jedoch auf mehr als die doppelte Abweichung.

Bei keiner der drei getesteten Netzwerkarchitekturen kann der Agent den mittleren Stellwinkel des Reglers dauerhaft unter den Referenzwert senken, wie Abbildung 5.11b verdeutlicht. Bei Verwendung des RL-Agenten mit dem größten KNN stellt der Regler in der letzten Runde mit einem RMS des Stellwinkels von 0.2720 rad knapp 40% mehr als bei der Referenzfahrt. Beim Einsatz der RL-Agenten mit kleineren Netzwerken zum Erlernen von Policy und Value-Function ist eine noch höhere Aktivität des Reglers erforderlich.

Da bei einer betragsmäßig kleineren Querablage als 0.233 m (vgl. Kapitel 5.2) der Rewardanteil $R_{lateral,e_y}$ positiv ist, ist der mit Architektur 1 erzielte Anteil der Querablage am lateralen Reward in jeder der Evaluationsrunden immer positiv. Folglich erhält der Agent bei steigender Regleraktivität aufgrund der stärkeren Gewichtung des lateralen Versatzes

dennoch eine positive Rückmeldung aus seiner Umgebung.

Bei Verwendung aller zur Verfügung stehenden Eingangsgrößen ist der RL-Agent mit Netzwerkarchitektur 1 am erfolgreichsten. Auf Basis der Versuchsergebnisse kann jedoch keine Kausalität zwischen steigender Neuronenanzahl sowie damit verbesserter Güte des RL-Agenten festgestellt werden, da der Agent mit den wenigsten Neuronen einen höheren Episodenreward als der Agent mit mittlerer Neuronenzahl erreicht.

Die verfügbaren Eingangsdaten aus Tabelle 5.1 sind sowohl auf das Erlernen eines Lenkwinkels als auch auf das Erlernen der Gas- sowie der Bremspedalstellung ausgelegt. Aufgrund der theoretischen Möglichkeit, dass während des Lernprozesses irrelevante Eingangsgrößen identifiziert und minimiert werden können, wurden anfänglich alle verfügbaren Fahrzustands- und Streckeninformationen verwendet. Da im Rahmen dieser Arbeit jedoch der Fokus auf die Querdynamikregelung gelegt wird, werden nachfolgend die Eingänge reduziert, um die Komplexität des Zustandsraums zu verringern. Neben dem Geschwindigkeits- wird der Beschleunigungsfehler nicht weiter berücksichtigt, da das Stellen eines Lenkwinkels durch den Regler hierrauf keinen direkten Einfluss hat. Um die Dimensionalität der Eingangsdaten weiter zu verringern, wird die Vorausschau auf einen zukünftigen Zeitpunkt beschränkt. Entsprechend werden ausschließlich die 15 Eingangsgrößen $v_x, v_y, \beta, \dot{\psi}, \ddot{\psi}, a_{y,beo}, a_{x,beo}, \kappa_0, \kappa_{50}, v_0, v_{50}, a_0, a_{50}, e_y$ und e_ψ verwendet.

Nachfolgend wird der vorherige Versuch mit der reduzierten Eingangszahl wiederholt. Abbildung 5.12 zeigt den lateralen Gesamtreward je Runde von den drei DDPG-Agenten.

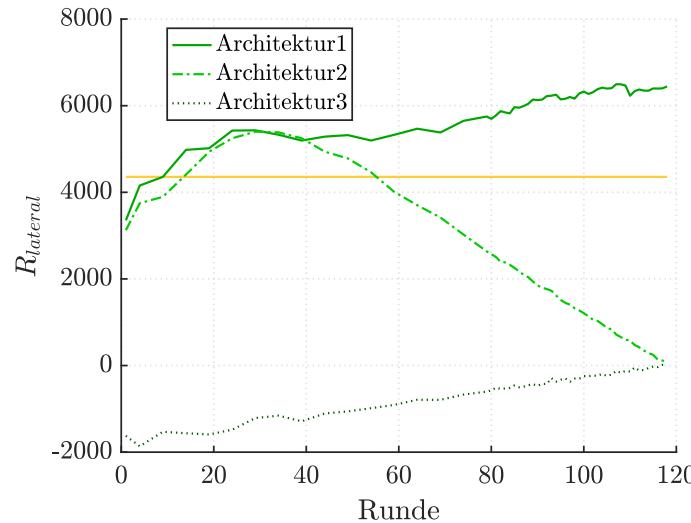


Abbildung 5.12: Lateraler Reward von drei DDPG-Agenten mit 15 der verfügbaren Eingangsgrößen

Wie zuvor starten die beiden größeren Netze mit Architektur 1 und 2 mit einem ähnlichen aufsummierten Reward von 3353.3 bzw. 3125.5. Im Verlauf des Trainings kann der DDPG-Agent mit Architektur 1 seinen Reward auf einen Wert von 6444.7 steigern und liefert somit ein rund 48% besseres Ergebnis als ohne den RL-Agenten. Nach anfänglicher Maximierung sinkt der vom Agenten mit Architektur 2 erhaltene Reward kontinuierlich ab. Der Agent mit Architektur 3 startet mit einem deutlich geringeren Reward in der ersten Episode. Im Verlauf des Trainings kann dieser verbessert werden, bleibt jedoch weit unter dem Referenzwert. Um den deutlich negativen Anfangsreward zu untersuchen, werden nachfolgend die Stellwinkel des RL-Agenten, des Reglers und der Vorsteuerung in Runde 1 sowie die Querablage über die Streckenmeter in Abbildung 5.13 dargestellt.

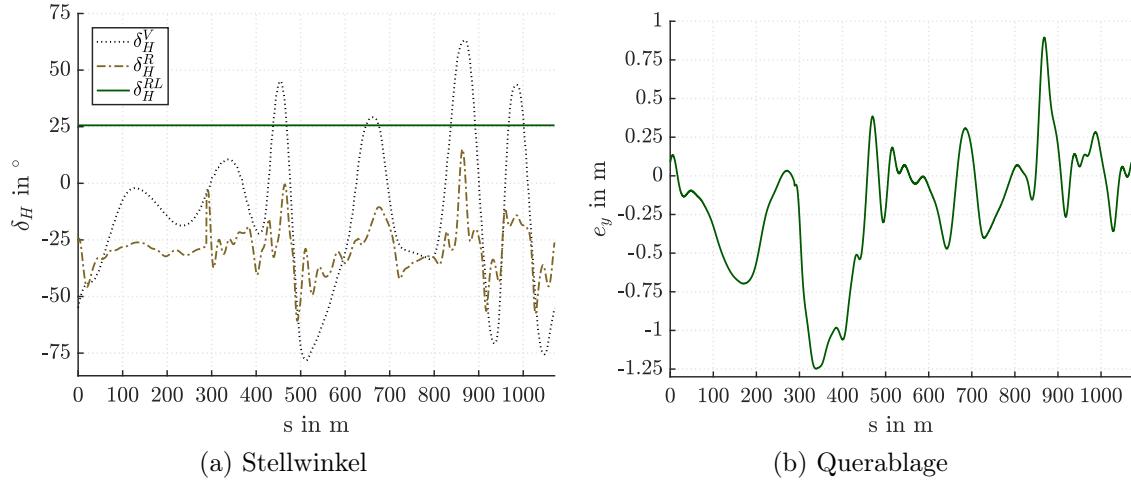


Abbildung 5.13: Stellwinkel und Querablage des DDPG-Agenten mit Architektur 3 und 15 der verfügbaren Eingangsgrößen in Runde 1

Bei Betrachtung von Abbildung 5.13a wird deutlich, dass aufgrund der in Kapitel 5.2 vorgestellten Initialisierung in der ersten Runde ein Winkel von konstant 0.43 rad $\approx 25^\circ$ durch den RL-Agenten gestellt wird. Der deutlich von 0 abweichende Winkel ergibt sich zum einen durch die Skalierung mit 4.5, um den gesamten Wertebereich des Winkels abzudecken, und zum anderen aufgrund der Umrechnung von Radiant in Grad. Da der Regler versucht diesen konstanten Offset auszugleichen, ist der gestellte Winkel dem Referenzverlauf aus Abbildung 5.8 ähnlich, jedoch zeigt sich deutlich die Verschiebung um rund -25° . Zur genaueren Analyse des Rewards für alle drei DDPG-Agenten mit der reduzierten Eingangszahl werden in Abbildung 5.14 die für die Berechnung des lateralen Rewards relevanten Messgrößen dargestellt.

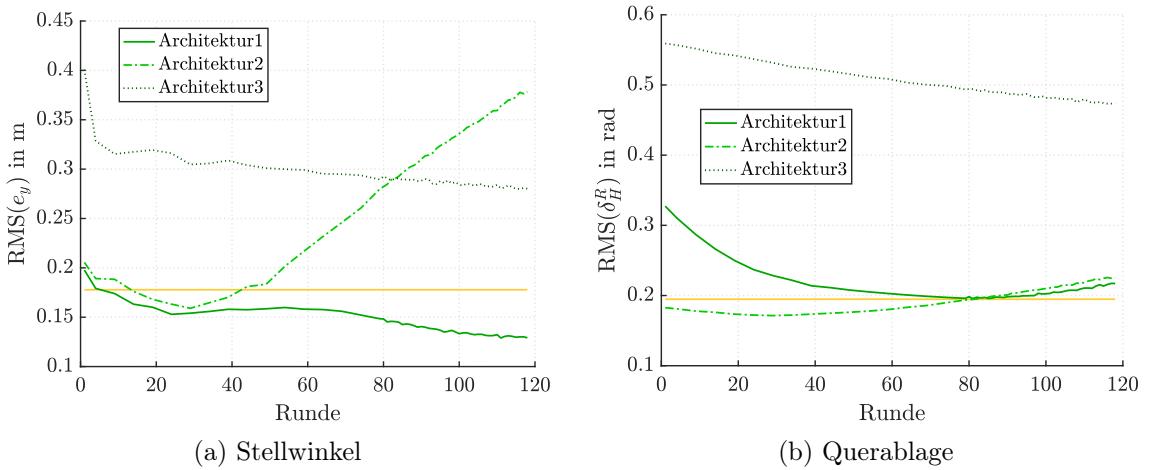


Abbildung 5.14: Rewardanteile von drei DDPG-Agenten mit 15 der verfügbaren Eingangsgrößen

Der RMS der Querablage in Abbildung 5.14a zeigt, dass der Agent mit Architektur 2 zu Beginn einen passenden additiven Winkel stellt und entsprechend die Abweichung zwischen Soll- und Isttrajektorie minimiert. Ab Runde 30 verschlechtert sich die Performanz kontinuierlich, welches auf einen deutlichen Anstieg der Querablage zurückzuführen ist.

Entsprechend steigt auch der zugehörige RMS des Stellwinkels vom Regler, da dieser versucht die Abweichung auszugleichen.

Mithilfe des größten KNN zur Abbildung von Policy und State-Action-Value-Function kann die Querablage kontinuierlich auf abschließend rund 13 cm reduziert werden. Das quadratische Mittel des Regler-Stellwinkels konnte im Verlauf des Trainings um rund ein Drittel verringert werden. Trotzdem wird der Referenzwert nicht unterschritten.

Wie der Reward bereits vermuten ließ, performt der RL-Agent mit Netzarchitektur 3 schlechter. Mit einer anfänglichen Querablage von 40 cm im Mittel, welche durch den deutlich zu hohen gestellten Winkel des RL-Agenten verursacht wird, ist es dem Agenten nicht möglich, die Abweichung zur Solltrajektorie unter den Referenzwert zu minimieren. Obwohl der RL-Agent die Aktivität des Lenkwinkelreglers von Trainingsanfang bis Trainingsende minimieren kann, stellt dieser deutlich mehr als bei den anderen beiden Agenten. Um den deutlich unterschiedlichen Verlauf der Performanz der Agenten mit Architektur 1 und Architektur 2 zu analysieren, obwohl diese mit einem ähnlichen Reward beginnen, wird in Abbildung 5.15 die Veränderung des Agenten-Stellwinkels sowie der Querablage in den Evaluationsrunden bis Runde 80 visualisiert. Die um Streckenmeter 290 auftretenden Sprünge in den Signalen sind Messfehlern geschuldet.

In Abbildung 5.15c und Abbildung 5.15d zeigt sich, dass trotzdem ähnlichem Anfangsreward beide Agenten in der ersten Runde stark verschiedene Winkel stellen. Der Agent mit Architektur 1 stellt in der ersten Runde nahezu konstant $0.1977 \text{ rad} \approx 11.33^\circ$, wohingegen der Agent mit Architektur 2 im Mittel $-0.0674 \text{ rad} \approx -3.86^\circ$ stellt. Die Abweichungen sind durch die Initialisierung der Netzwerkgewichte erklärbar, da die PyTorch-Standardinitialisierung in den vorderen Schichten die He-Initialisierung (vgl. Kapitel 2.2.2) verwendet, welche die Anzahl der Neuronen berücksichtigt. Entsprechend entstehen unterschiedliche hohe Eingaben in die letzte Schicht und mit der dortigen Initialisierung um 0 abweichende Aktionsprädiktionen.

Aufgrund der unterschiedlichen Stellwinkel der RL-Agenten ergeben sich ebenfalls verschiedene Verläufe für die Stellwinkel der zugehörigen Regler, welche in Abbildung 5.15a und Abbildung 5.15b zu sehen sind. Im Mittel stellt der Regler bei Architektur 2 geringere Winkel, wie der Startwert der zugehörigen Kurve in Abbildung 5.14b bestätigt. Während sich der gestellte Winkel des Reglers bei Architektur 2 über die Runden jedoch tendenziell betragsmäßig erhöht, wird der Winkel beim Regler zugehörig zum RL-Agenten mit Architektur 1 über weite Teile der Strecke über die dargestellten 80 Runden minimiert.

Aus Abbildung 5.15c wird deutlich, dass der Agent mit zunehmender Rundenzahl den Winkel Richtung null verschiebt und entsprechend der Strecke sowohl positive als auch negative Winkel stellt. Insgesamt sind die gestellten Winkel des RL-Agenten mit maximal 6.73° bzw. -8.93° in Runde 80 gering. Der Agent mit Architektur 2, zu sehen in Abbildung 5.15d verändert den Stellwinkel ebenfalls im Verlauf des Trainings abhängig der Strecke, wobei mit 20.59° bzw. -11.67° deutlich größere Winkel gestellt werden. Bei Vergleich der beiden Abbildungen wird deutlich, dass die gestellten Winkel beider Agenten stark voneinander abweichen und entsprechend die unterschiedliche Performanz verursachen. Abbildung 5.15f zeigt die Querablage des Agenten mit Architektur 2. Hier zeigt sich, dass auf den verhältnismäßig hohen Lenkwinkel um Streckenmeter 100 eine entsprechend hohe Querablage des Fahrzeugs folgt. Mit ähnlichem Stellwinkel nach Streckenmeter 600 wird eine deutlich geringere Querablage erreicht. Bei Betrachtung der Kurve zwischen Streckenmeter 300 und 400 bzw. 600 und 700 in Abbildung 5.6 wird jedoch deutlich, dass beide unterschiedliche Krümmungen aufweisen und entsprechend andere Lenkwinkel erfordern. Der RL-Agent mit Architektur 2 kann auf Streckenabschnitten mit anfangs höherer Abwei-

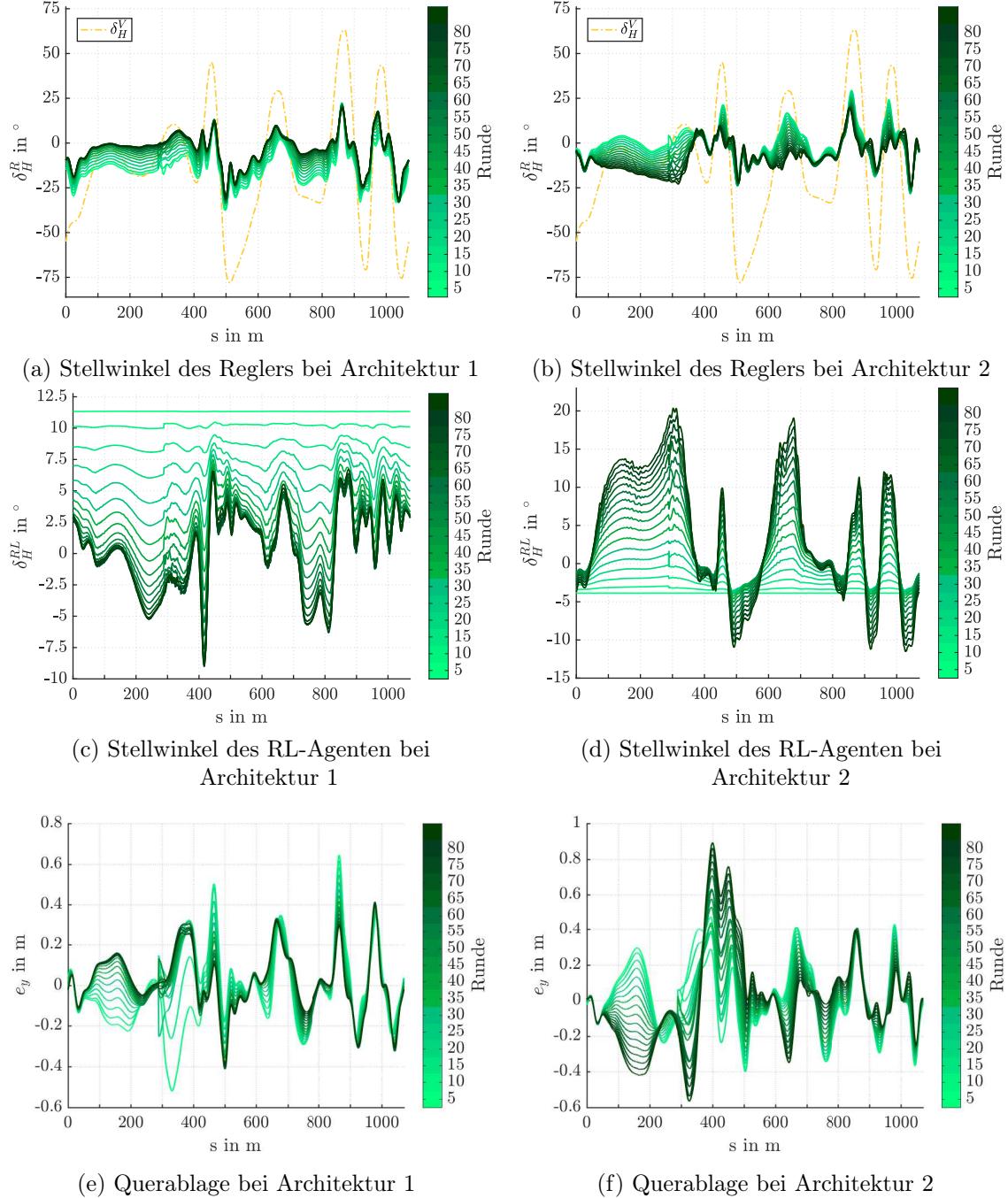


Abbildung 5.15: Vergleich der Stellwinkel und der Querablage von zwei verschiedenen DDPG-Agenten mit 15 der verfügbaren Eingangsgrößen

chung zur Solltrajektorie ein verbessertes Verhalten erlernen, wie Abbildung 5.15e zeigt. Aus der Betrachtung des Rewards (vgl. Abbildung 5.12) sowie dessen einzelner Bestandteile (vgl. Abbildung 5.14) folgt, dass der RL-Agent mit den neuronalen Netzen mit je 200 Neuronen in der ersten und 100 Neuronen in der zweiten versteckten Schicht auch bei weniger Eingangsdaten das beste Ergebnis erzielt. Trotz der geringeren Anzahl an Eingangsparametern performen die kleineren Netzarchitekturen weiterhin wenig zufriedenstellend. In den vorherigen Versuchen zeigt sich, dass beide RL-Agenten, welche Netze mit 200 Neu-

ronen in der ersten und 100 Neuronen in der zweiten versteckten Schicht enthalten, mit DDPG die besten Trainingsergebnisse liefern. Beide erreichen dabei merklich höhere Rewards als das Referenzfahrzeug, bei welchem die Trajektorienfolgeregelung ausschließlich aus dem klassischen Ansatz aus Vorsteuerung und Regler besteht. Die Reduzierung der Eingangsgrößen vermindert die Lernbarkeit des additiven Stellwinkels nicht, im direkten Vergleich ist der final erhaltene Rundenreward sogar etwas höher.

Da mit niedrigerer Eingangsgrößenzahl die Komplexität reduziert und wie gezeigt das Ergebnis nicht verschlechtert wird, werden nachfolgend ausschließlich die 15 Fahrzustandsgrößen und Streckeninformationen verwendet.

Training mit PGQL

Nach Betrachtung des Trainings mittels Deep Deterministic Policy Gradient soll PGQL zum Erlernen eines additiven Stellwinkels verwendet werden. Basierend auf den 15 Eingangsgrößen $v_x, v_y, \beta, \dot{\psi}, \ddot{\psi}, a_{y,beto}, a_{x,beto}, \kappa_0, \kappa_{50}, v_0, v_{50}, a_0, a_{50}, e_y$ und e_ψ wird mithilfe der KNN nach Abbildung 4.2b bzw. Abbildung 4.3b mit den Neuronenanzahlen aus Tabelle 5.3 eine stochastische Policy sowie eine State-Value-Function erlernt. Da entsprechend kein zusätzliches Rauschen hinzugefügt wird, kann jede gefahrene Runde zur Beurteilung der aktuellen Agentenperformance verwendet werden.

In Abbildung 5.16 wird der über die jeweilige Runde aufsummierte laterale Reward für die drei getesteten Architekturen dargestellt.

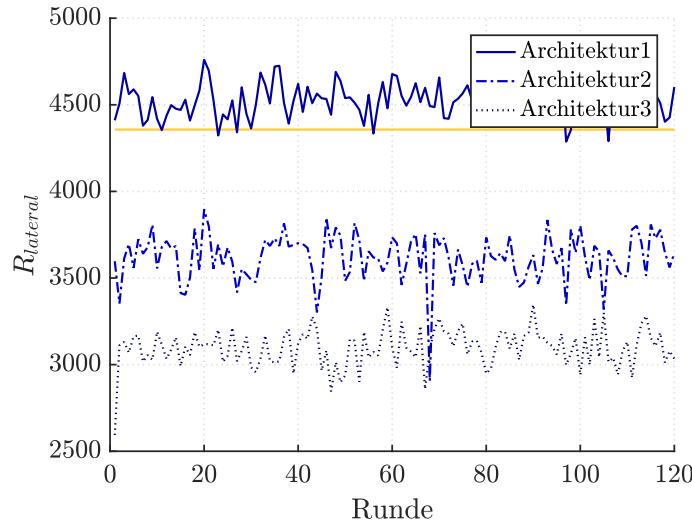


Abbildung 5.16: Lateraler Reward von drei PGQL-Agenten mit 15 der verfügbaren Eingangsgrößen

Der PGQL-Agent mit den größten KNN erzielt in nahezu allen Runden einen Reward über dem Referenzwert und liefert damit die besten Ergebnisse. Bei Betrachtung der Kurvenverläufe für alle drei Netzarchitekturen zeigt sich jedoch, dass keine kontinuierliche Verbesserung zu erkennen ist. Der Reward des Agenten mit Architektur 1 schwankt um 4500, der des Agenten mit Architektur 2 um 3600 und der des Agenten mit Architektur 3 um 3100.

Die Verläufe der einzelnen Anteile des lateralen Rewards in Abbildung 5.17 zeigen ein ähnliches Muster und entsprechend wenig Änderungen der einzelnen Einflüsse über die Rundenanzahl.

Aufgrund dessen, dass bei PGQL die Gewichte nach jeder einzelnen Aktion geupdated wer-

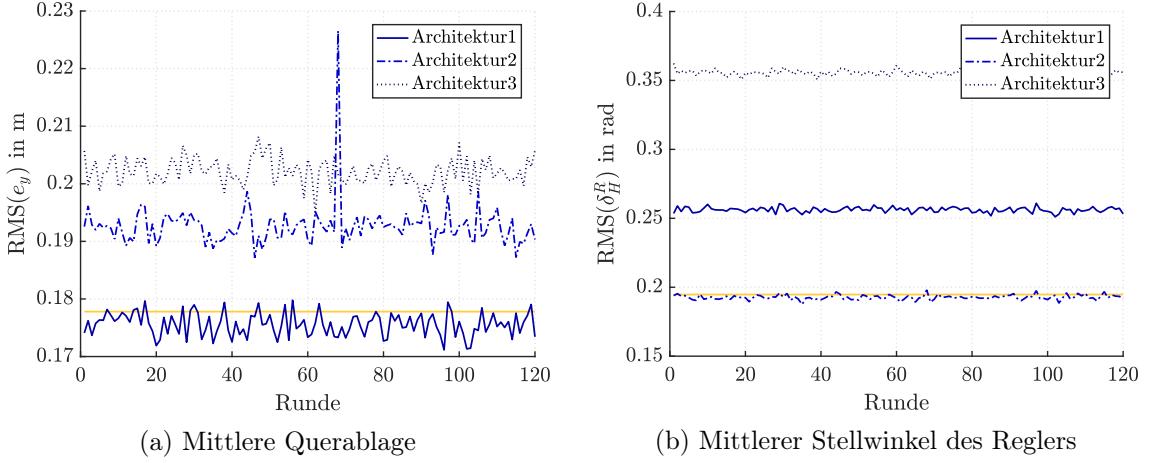


Abbildung 5.17: Rewardanteile von drei PGQL-Agenten mit 15 der verfügbaren Eingangsgrößen

den, soll nachfolgend der Zusammenhang zwischen dem Stellwinkel des RL-Agenten und dem erhaltenen Reward genauer beleuchtet werden. Zur Vereinfachung wird lediglich der Agent mit Netzarchitektur 1 betrachtet, da dieser den höchsten Reward erzielt. Abbildung 5.18b zeigt beispielhaft den Stellwinkel des RL-Agenten, den Stellwinkel des Reglers sowie den erhaltenen lateralen Reward für die ersten zehn Streckenmeter in Runde 10.

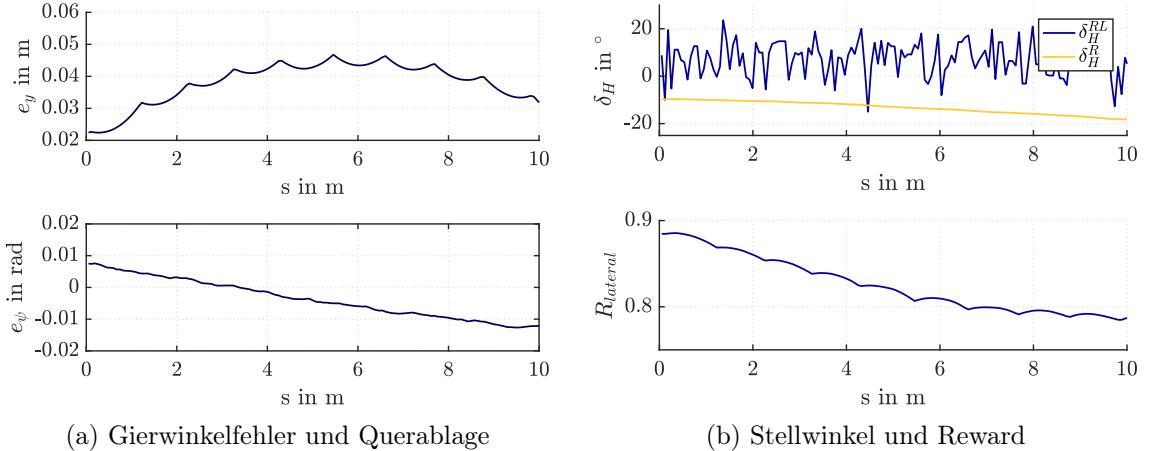


Abbildung 5.18: Zusammenhang zwischen Stellwinkel und Reward des PGQL-Agenten mit Architektur 1 auf den ersten zehn Metern von Runde 10

Abbildung 5.18a zeigt oben die Abweichung zwischen Soll- und Isttrajektorie auf den ersten zehn Streckenmetern. Hierbei wird deutlich, dass der Agent nur wenige Zentimeter von der geplanten Trajektorie abweicht, weshalb der Anteil der Querablage am lateralen Reward (vgl. Abbildung 5.5) entsprechend hoch sein wird. Die Längsregelung wird durch den RL-Agenten nicht direkt beeinflusst, weshalb angenommen werden kann, dass das Fahrzeug mit RL-Agenten eine ähnliche Geschwindigkeit wie das Referenzfahrzeug fährt. Da sich der Streckenabschnitt in den betrachteten 10 m wenig ändert (vgl. Abbildung 5.6), sind die Zustände und entsprechend die Eingangsgrößen für das Actor-Netzwerk des PGQL-Agenten nahezu konstant. Trotzdem zeigt sich in Abbildung 5.18b, dass der Agent die Aktionen eher zufällig wählt, da diese stark zwischen positiven und negativen Stellwinkel

springen. Der Regler hingegen ermittelt sein Stellsignal basierend auf dem Gierwinkelfehler und der Querablage, welche wie Abbildung 5.18a zeigt, geringe Änderungen aufweisen. Entsprechend schwankt das Stellsignal des Reglers nicht, es wird lediglich von knapp -10° auf ungefähr -18° verändert. Durch Gewichtung nach Gleichung 5.1 folgt ein Anteil am lateralen Reward zwischen -0.1 und -0.18 . Da die Abweichung zwischen Soll- und Isttrajektorie immer kleiner als 0.233 m ist, ist der Gesamtreward trotzdem dauerhaft positiv. Durch die Ähnlichkeit der Zustände in aufeinanderfolgenden Zeitschritten und aufgrund der Initialisierung der Critic, welche zu initialen Ausgaben nahe 0 führt, wird die Anpassung der Critic hauptsächlich auf Basis des Rewards geschehen (vgl. Gleichung 3.9). Da dieser positiv ist, wird der Agent den entsprechenden Zustand als gut bewerten und infolgedessen den Actor anpassen. Da die Netzausgaben des Actors die gezeigten starken Schwankungen aufweisen, wird vermutet, dass aufgrund der Online-Updates häufig gegenläufige Änderungen entstehen. Entsprechend ist es für den Agenten schwierig die Gewichte so anzupassen, dass der Reward selbstbestimmt maximiert wird.

Da PGQL bereits im Rahmen der Validierung der Implementierung am Beispiel des Continuous Mountain Car keine durchgehend erfolgreiche Performance und Schwierigkeiten aufgrund der Abhängigkeiten von der gewählten Gewichtsinitialisierung zeigte, wird nicht weiter nach einer erfolgreicher Hyperparametervariation gesucht. Nachfolgend wird ACKTR als erfolgversprechenderes Verfahren, welches eine stochastische Policy erlernt, evaluiert.

Training mit ACKTR

ACKTR ist der zweite evaluierte Actor-Critic-Algorithmus, der eine stochastische Policy erlernt. Dabei werden der Aufbau der Netzwerke sowie die Anzahl der Neuronen aus Abschnitt 5.2 verwendet. Zur Minderung der Komplexität des Problems ohne Einschränkung der Lernbarkeit werden erneut die 15 Eingangsgrößen aus den beiden vorherig genutzten Algorithmen als Netzeingaben zum Erlernen der Policy und der State-Value-Function verwendet.

Aufgrund der hohen Varianz in der Aktionswahl bei ähnlichen Zuständen, die sich bei PGQL zeigte, sollen nachfolgend die Gewichte zur Berechnung abweichend der vorherigen Initialisierung gewählt werden. Bei Verwendung der in Abschnitt 5.2 beschriebenen Initialisierung der Gewichte liefert das Actor-KNN zu Beginn aufgrund der Verkettung der Aktivierungsfunktion einen Mittelwert von nahezu null. Aufgrund der Initialisierung und der Verwendung von ReLU in den vorderen Schichten wird die Eingabe in die letzte Schicht zur Berechnung der Standardabweichung um null liegen. Durch die Verwendung von Softplus in der letzten Schicht, wird die Netzausgabe folglich ungefähr 0.69 betragen. Durch ein von der Standardinitialisierung abweichendes erstes Setzen des Bias soll eine Verschiebung der Schichteingabe erreicht werden. In Kombination mit der Softplus-Aktivierungsfunktion wird entsprechend eine andere initiale Standardabweichung für die Gaußverteilung zum Aktionssampling erzielt. Nachfolgend wird der Bias der letzten Schicht zur Prädiktion der Standardabweichung mit -2 initialisiert. Aufgrund der Initialisierung der früheren Schichten und entsprechend einer Eingabe um null in die letzte Schicht wird die Neuronaktivierung nahezu ausschließlich vom Bias abhängen. Folglich wird durch die Verwendung von Softplus (vgl. Abbildung 2.9) und dem Bias von -2 die Netzausgabe für die Standardabweichung um 0.127 liegen.

Im anschließenden Versuch werden drei Agenten mit ACKTR trainiert, die die Neuronenkonstellationen aus Tabelle 5.3 zur Abbildung einer stochastischen Policy und einer State-Value-Function nutzen. Der erzielte aufsummierte Reward je Runde ist vergleichend für die Agenten in Abbildung 5.19 dargestellt.

Der Agent mit den künstlichen neuronalen Netzen der Architektur 1 liefert einen Reward,

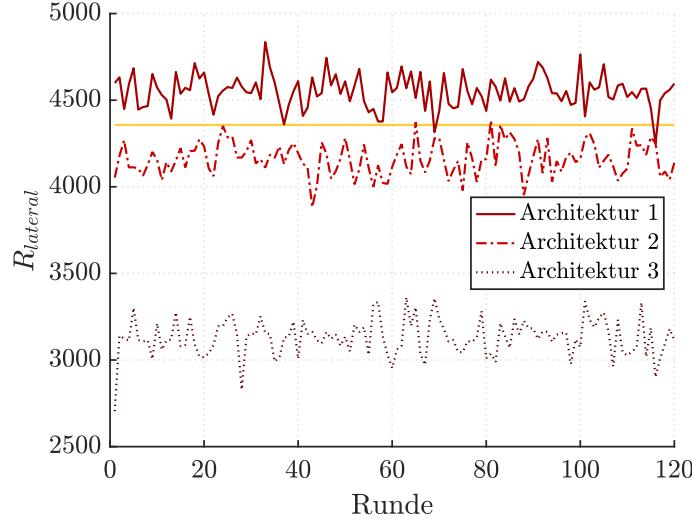


Abbildung 5.19: Lateraler Reward von drei ACKTR-Agenten mit verschiedenen Netzarchitekturen

welcher nahezu durchgehend über dem Referenzwert liegt, wohingegen der mit Architektur 2 erzielte Wert knapp unter dem Referenzwert bleibt. Der mithilfe von Architektur 3 erzielte Reward ist mit einem Wert um 3130 deutlich geringer.

Prinzipiell zeigt sich ein ähnlicher Verlauf wie beim Training mit PGQL (vgl. Abbildung 5.16). Die einzelnen Kurven zeigen keine klaren Richtungstendenzen der Agenten, sondern schwanken um einen konstanten Wert. Abbildung 5.20b zeigt den mittleren Stellwinkel des Reglers, der für jeden Agenten nahezu konstant ist. Die Schwankungen im Reward basieren entsprechend auf Schwankungen in der mittleren Querablage, welche Abbildung 5.20a darstellt.

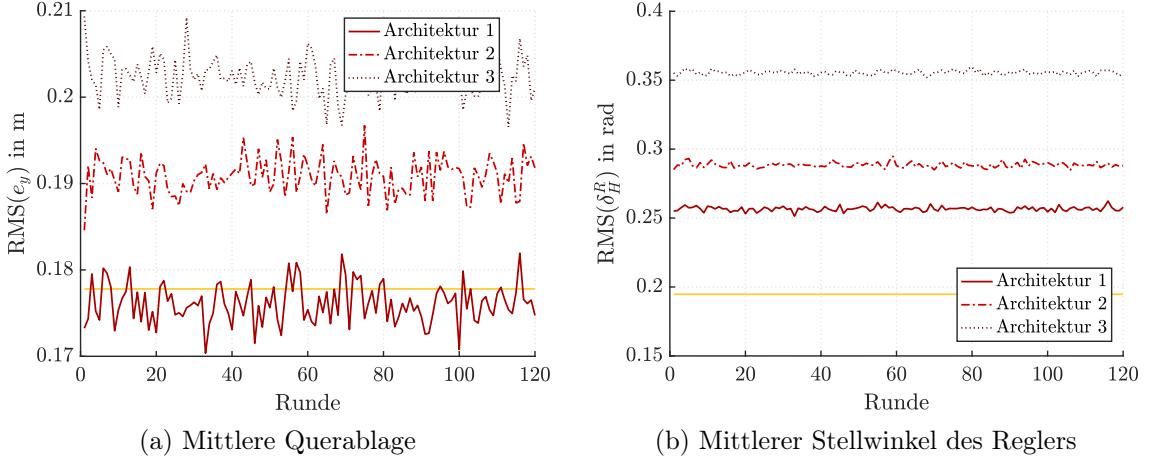


Abbildung 5.20: Rewardanteile von drei ACKTR-Agenten mit verschiedenen Netzarchitekturen

Da die beiden Agenten mit den größeren Netzwerken im Mittel nur im Bereich der akzeptierten Toleranz von der Solltrajektorie abweichen, ist der nach Gleichung 5.1 berechnete Rewardanteil überwiegend positiv. Aufgrund der geringen Gewichtung des Anteils durch den Stellwinkelregler in Kombination mit dem im Mittel gestellten $0.26 \text{ rad} \approx 14.9^\circ$ bzw.

$0.28 \text{ rad} \approx 16.04^\circ$ ist der laterale Reward ebenfalls insgesamt überwiegend positiv. Die mittlere Abweichung zwischen Soll- und Isttrajektorie des Agenten mit Architektur 3 ist höher. Bei der in Abbildung 5.20a gepunktet dargestellten Kurve ist davon auszugehen, dass der Agent in mehreren Fällen den tolerierten Bereich von $\pm 23.3 \text{ cm}$ verlässt und entsprechend neben der Aktivität des Stellwinkelreglers ebenfalls aufgrund der Querablage einen negativen Reward erhält.

Aufgrund dessen, dass sich im Training keine kontinuierliche Entwicklung der Agenten zeigt, wird nachfolgend in Abbildung 5.21 exemplarisch die mit Architektur 1 prädizierte Standardabweichung gemittelt über die jeweilige Trainingsepisode betrachtet.

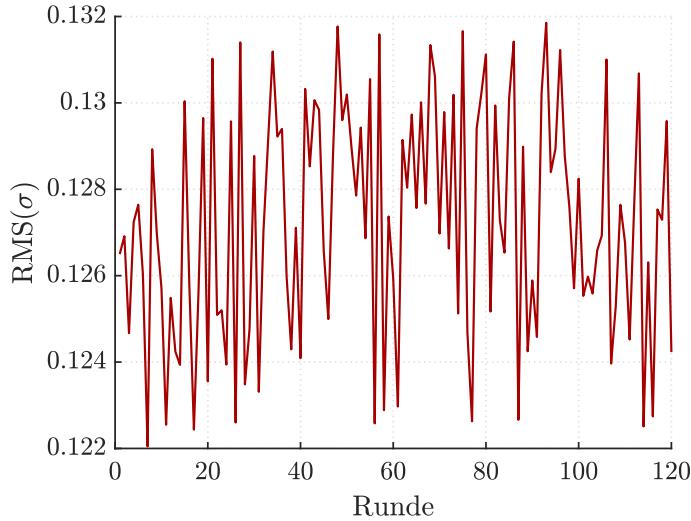


Abbildung 5.21: Prädizierte Standardabweichung des ACKTR-Agenten mit Architektur 1

Die Annahme, dass die prädizierte Standardabweichung zu Beginn stark durch die Initialisierung des Bias verändert werden kann, wird durch die mittlere prädizierte Standardabweichung in Runde null bestätigt. Diese entspricht nahezu dem erwartenden Wert von 0.127, welcher zu Beginn des Abschnitts begründet wurde. Allgemein schwankt die im Mittel prädizierte Standardabweichung um diesen erwarteten Wert. Entsprechend wird die Standardabweichung nicht wie gewünscht im Trainingsverlauf verringert, wodurch der Agent über den gesamten Trainingszeitraum vorwiegend zufällige Aktionen wählt.

Abbildung 5.22 zeigt exemplarisch für Runde 1 und Runde 120 den Stellwinkelverlauf des RL-Agenten mit Architektur 1 sowie des Reglers, um die Auswirkungen der hohen prädizierten Standardabweichung zu verdeutlichen.

Dabei stellt Abbildung 5.22a exemplarisch den Stellwinkel dar. Prinzipiell sind beide Verläufe ähnlich, da das Stellsignal in beiden Runden um einen Mittelwert von 9.17° schwankt, welcher durch die Gewichtsinitialisierung begründet werden kann. Insgesamt zeigt sich ein sehr sprunghafter Verlauf zwischen negativen Werten bis -22° und positiven Werten bis 39° . Da zwischen zwei Aktionen aufgrund der Echtzeitanforderung lediglich 5 ms vergehen und zum Aufbau sowie zur Umsetzung der Kraft eine gewisse Zeit erforderlich ist, wird der tatsächlich in der Fahrzeugsimulation gestellte Winkelanteil durch den RL-Agenten annähernd dem Mittelwert von 9.17° entsprechend. Abbildung 5.22b zeigt den Verlauf des Stellwinkelsignals der Vorsteuerung sowie des Reglers exemplarisch für Runde 1 und Runde 120. Der ähnliche Verlauf im Stellwinkel des ACKTR-Agenten spiegelt sich im Verlauf des Stellwinkels durch den Regler wider, der entsprechend ebenfalls in beiden Runden ähnlich ist. Die starken Schwankungen im Signal des RL-Agenten führen zu Schwankungen im

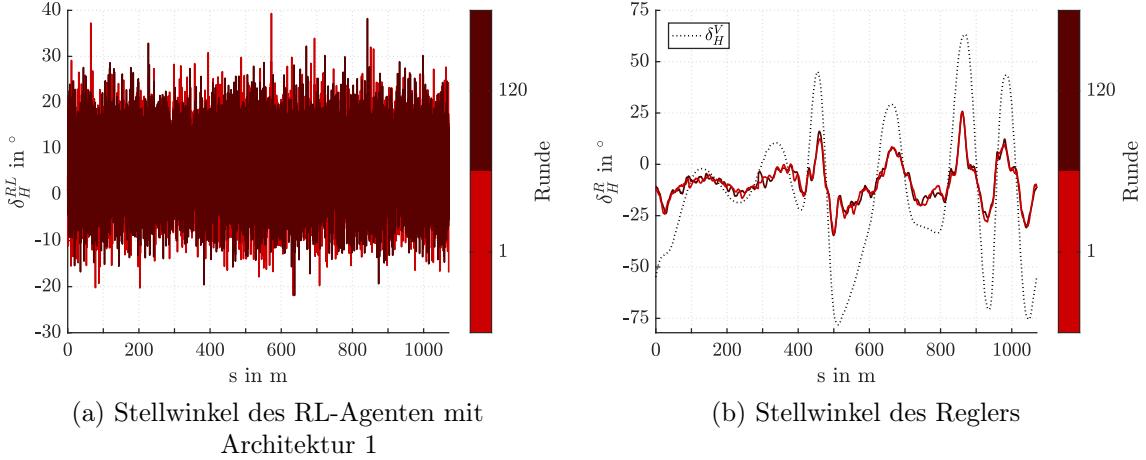


Abbildung 5.22: Stellwinkel des ACKTR-Agenten mit Architektur 1 in Runde 1 und 120

Signal des Reglers, wie sich im Vergleich zwischen Abbildung 5.22b und dem Referenzfahrzeug in Abbildung 5.8 zeigt. Daraus folgt ebenfalls der im Mittel höhere RMS des Reglerstellwinkels, welcher in Abbildung 5.20b dargestellt wurde.

Aufgrund der Schwierigkeiten beim Erlernen der Standardabweichung wird diese im nächsten Versuch nicht erlernt. Abweichend zum vorherigen Versuch wird nachfolgend ein Actor-KNN mit prinzipiellem Aufbau nach Abbildung 4.2a verwendet, welches jedoch ELU statt ReLU verwendet und keine explizite Aktion sondern den Mittelwert einer Gaußverteilung prädiziert. Die Neuronen werden entsprechend Architektur 1 gewählt. Die Standardabweichung der Gaußverteilung wird ausgehend vom Startwert 0.25 linear über 80 Runden reduziert, bis ein Minimalwert von 0.01 erreicht ist. Wie bei DDPG wird vor Runde 80 jede fünfte als Evaluationsrunde verwendet, d. h., dass die Standardabweichung auf 0.01 gesetzt wird, um die Performanz des Netzes zu beurteilen. Der Verlauf der Standardabweichung ist in Abbildung 5.23 zu sehen.

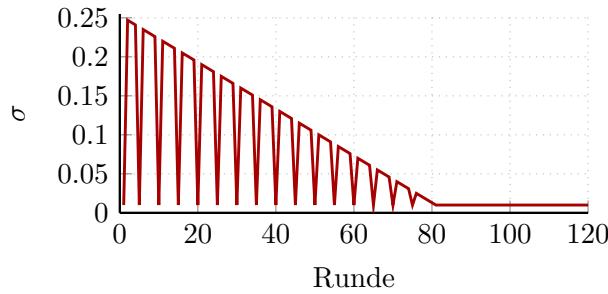


Abbildung 5.23: Rundenbasierte Standardabweichung für das Training eines ACKTR-Agenten

Nachfolgend werden zwei Agenten, welche Netzwerke mit 200 Neuronen in der ersten und 100 Neuronen in der zweiten versteckten Schicht verwenden, verglichen. Dabei handelt es sich zum einen um den vorherigen Agenten, welcher Mittelwert und Standardabweichung erlernt und zum anderen um einen Agenten, welcher den Mittelwert erlernt und eine rundenreduzierte Standardabweichung verwendet. Abbildung 5.24 zeigt den lateralen Reward. Obwohl der Agent, bei welchem die Standardabweichung über die Rundenzahl minimiert wird, am Trainingsende einen höheren Reward als zu Trainingsbeginn erzielt, zeigt sich kein

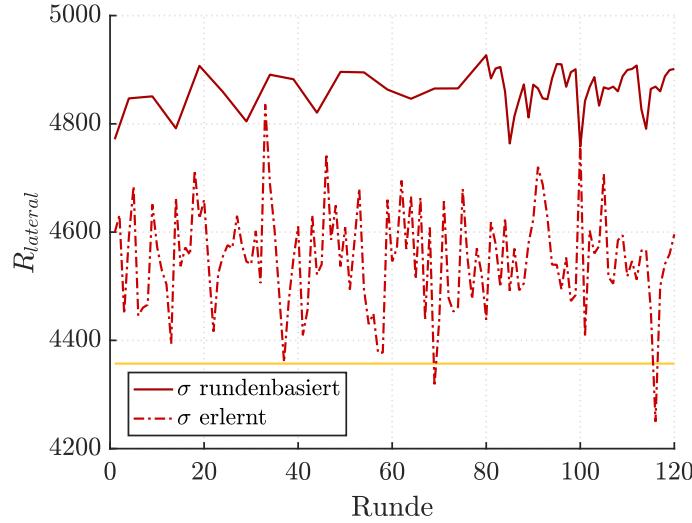


Abbildung 5.24: Lateraler Reward von einem ACKTR-Agenten mit Architektur 1

kontinuierlicher Verlauf. In der Abbildung können ab Runde 80 die Verläufe direkt verglichen werden. Hierbei zeigt sich, dass der Agent mit rundenbasierter Standardabweichung geringere Schwankungen im Reward aufweist als der Agent mit erlernter Standardabweichung. Grund hierfür ist die deutlich geringere Standardabweichung. Wie Abbildung 5.21 zeigt, ist die Standardabweichung mit 0.1242 beim Erlernen rund zwölfmal größer als bei der rundenbasierten Variante, bei welcher die Standardabweichung am Ende 0.01 beträgt. Aufgrund der gewählten minimalen Standardabweichung von 0.01 und der Skalierung mit dem Faktor von 4.5, um das Stellen eines Winkels im kompletten Wertebereich zu ermöglichen, sind dennoch maximale Winkelsprünge von 0.225 rad $\approx 12.89^\circ$ möglich.

Abbildung 5.25 zeigt die mittlere Querablage und den mittleren Stellwinkel des Reglers vergleichend für beide Agenten.

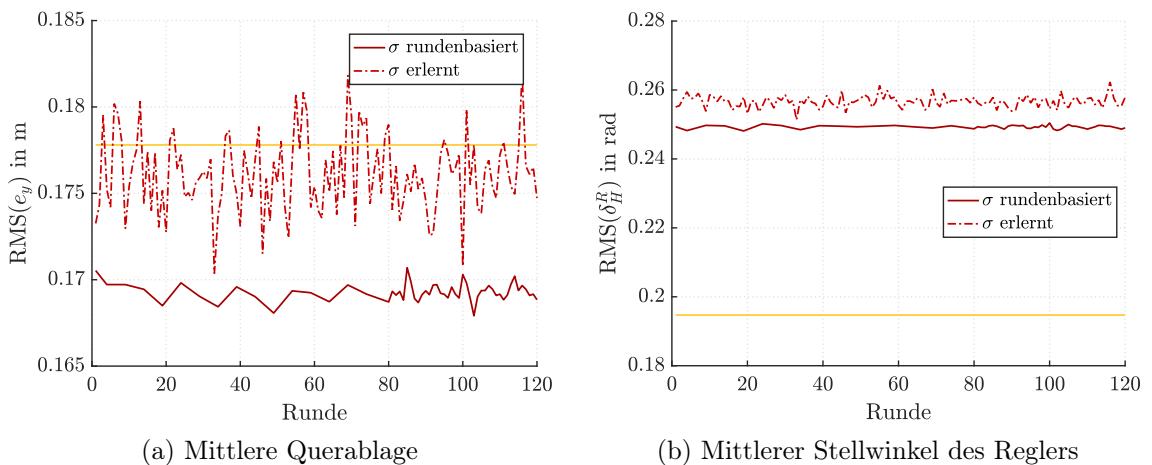


Abbildung 5.25: Rewardanteile von einem ACKTR-Agenten mit Architektur 1

Wie anhand des lateralen Gesamtrewards zu erwarten, sind sowohl die mittlere Querablage als auch der mittlere Stellwinkel des Reglers bei Verwendung der rundenbasierten Standardabweichung zur Aktionsvorhersage geringer als bei der erlernten Standardabweichung. Zum Abschluss des Trainings mit ACKTR lässt sich vorerst festhalten, dass die Ergebnisse

keine deutliche Verbesserung im Vergleich zu PGQL darstellen.

5.5 Vergleich der Ergebnisse und Fazit

Zum Abschluss des Kapitels zu den Simulationsergebnissen, welche auf Basis der Erweiterung eines bestehenden Trajektorienfolgeregelungskonzeptes durch verschiedene RL-Agenten erzielt wurden, sollen die besten Ergebnisse der drei Actor-Critic-Algorithmen DDPG, PGQL und ACKTR verglichen werden. In allen Versuchen konnte der RL-Agent mit Netzarchitektur 1, welcher 200 Neuronen in der ersten und 100 Neuronen in der zweiten versteckten Schicht enthält, im Vergleich zu den jeweils anderen getesteten Neuronenzahlungen den höchsten lateralen Reward erzielen. Die jeweils besten Agenten treffen dabei basierend auf dem reduzierten Eingangsvektor bestehend aus den Fahrzustands- und Streckeninformationen $v_x, v_y, \beta, \dot{\psi}, \ddot{\psi}, a_{y,beo}, a_{x,beo}, \kappa_0, \kappa_{50}, v_0, v_{50}, a_0, a_{50}, e_y$ und e_ψ , welcher mithilfe der Minimal- und Maximalwerte aus Tabelle 5.3 auf einen Bereich von -1 bis 1 skaliert wurde, die Wahl der nachfolgenden Aktion.

Abbildung 5.26 zeigt zur Beurteilung der Performanz und des Trainingsfortschritts den lateralen Reward der jeweils besten RL-Agenten.

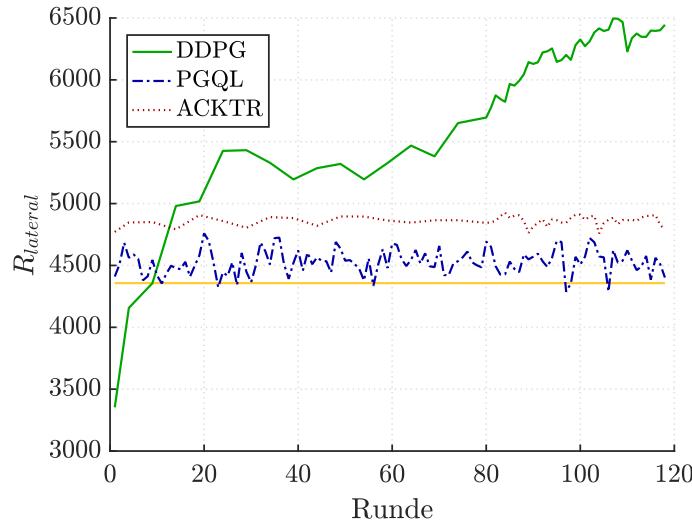


Abbildung 5.26: Lateraler Reward des besten Agenten für jeden Algorithmus

Hierbei zeigt sich deutlich, dass der DDPG-Agent als einziger den aufsummierten lateralen Reward nahezu kontinuierlich maximiert und am Ende einen deutlich höheren Reward als das Referenzfahrzeug erreicht.

Die Agenten, welche mit ACKTR und PGQL trainiert wurden und entsprechend eine stochastische Policy erlernen, wählen stark schwankende Aktionen (vgl. exemplarisch Abbildung 5.22a). Grund hierfür ist die prädizierte Standardabweichung, die bei beiden Verfahren über das Training nicht minimiert wird und entsprechend keine gerichtete Auswahl der Aktion ermöglicht. Trotz der rundenbasierten Reduktion der Standardabweichung beim letzten Versuch ACKTR konnte kein RL-Agent gefunden werden, der das Regelungskonzept basierend auf den additiven Stellwinkeln kontinuierlich verbessert. Infolgedessen und auf Basis der errechneten maximalen Winkelsprünge von annähernd 13° bleibt die Frage bestehen, ob mit einer deutlich geringeren Standardabweichung ein besseres Lernergebnis erzielt werden kann.

Im Gegensatz zu den Agenten mit stochastischer Policy konnte mit DDPG ein Agent

trainiert werden, welcher durch den additiven Stellwinkel (vgl. Abbildung 5.15c) die Performance der Trajektorienfolgeregelung verbessert. Da DDPG ein Off-Policy Algorithmus ist, können vorherige Erfahrungen zur Verbesserung der Policy wiederverwendet werden. Außerdem ermöglicht das Erlernen einer deterministischen Verhaltensweise die getrennte Betrachtung der Exploration. Aufgrund der Verwendung des OU-Prozesses anstelle einer Gaußverteilung erkundet der DDPG-Agent seine Umgebung deutlich zielgerichteter, wie in Kapitel 3.2 erläutert wurde. Trotz der erfolgversprechenden Ergebnisse mit DDPG sollte angemerkt werden, dass sich in Abbildung 5.15 der Einfluss der Initialisierung des Netzwerkes auf die letztendliche Problemlösung zeigt. In Zukunft sollten verschiedene Initialisierungen betrachtet werden, um möglicherweise bessere Ergebnisse, auch mit anderen Netzwerkarchitekturen, zu erzielen.

6 Zusammenfassung und Ausblick

Mit zunehmender Automatisierung der Fahraufgabe steigt die Anzahl an zu lösenden Aufgaben aus den Bereichen Wahrnehmung, Planung und Regelung für das Fahrzeug. Um jederzeit ein sicheres und stabiles Fahrzeugverhalten zu garantieren, werden vermehrt modellbasierte und modellfreie Konzepte kombiniert, um eine robuste Fahrzeugregelung zu garantieren. Infolgedessen finden vermehrt Verfahren aus dem Bereich des maschinellen Lernens, speziell aus dem Reinforcement Learning, Anwendung.

Während in der Literatur überwiegend End-to-End-Learning-Ansätze verwendet werden, bei welchem mithilfe des RL-Agenten die komplette Regelung übernommen wird, wird der RL-Agent in dieser Arbeit als additives Stellglied verwendet. untersucht. Durch Ergänzung des Regelungskonzeptes bestehend aus einer Vorsteuerung und einem PID-Regler sollen Modellungenauigkeiten ausgeglichen und infolgedessen die Performanz der Regelung verbessert werden. Basierend auf dem Stand der Technik, welcher eine Auswahl von RL-Anwendungen im Bereich der Robotik und Fahrzeugführung aufzeigte, sowie den theoretischen Grundlagen des Reinforcement Learnings wurden drei Actor-Critic Algorithmen ausgewählt, welche bezüglich ihrer Anwendbarkeit als zusätzliches Regelglied einer Querdynamikregelung evaluiert werden sollten. Da zur Umsetzung der erlernten Policy sowie der Value-Function künstliche neuronale Netze verwendet werden, wurden die entsprechenden Grundlagen ebenfalls erläutert. Anhand dem Continuous Mountain Car, wurden erste Besonderheiten und Einschränkungen in der Anwendung der Methoden in einem kontinuierlichen Zustands- und Aktionsraum detektiert.

Anschließend erfolgte die Fusion der Agenten mit der Fahrzeugsimulationsumgebung. In verschiedenen simulativen Fahrversuchen auf einer Strecke mit abwechslungsreichem Streckenprofil wurden RL-Agenten mit verschiedenen Hyperparametern und den drei RL-Verfahren trainiert. Als Referenz zur Beurteilung der Verbesserung der Querdynamikregelung steht ein Fahrzeug mit dem bestehenden Trajektorienfolgeregelungskonzeptes ohne additives RL-Stellglied auf der identischen Strecke zur Verfügung. Als Bewertungskriterium wird der aufsummierte laterale Reward über eine Runde verwendet, welcher neben der Abweichung zwischen Soll- und Isttrajektorie ebenfalls die Aktivität des PID-Reglers berücksichtigt.

Hierbei zeigte sich, dass mittels Verfahren, welche eine stochastische Policy erlernen, kein verbessertes Verhalten des Fahrzeugs erzielt werden konnte. Aufgrund der fehlenden Minimierung der durch ein KNN prädizierten Standardabweichung, welche in Kombination mit einem prädizierten Mittelwert und einer Gaußverteilung für das Aktionsampling verwendet wird, trifft der Agent überwiegend zufällige Aktionen. Im Rahmen weiterer Arbeiten sollte neben der Verwendung einer anderen Verteilung eine abweichende Initialisierung mit deutlich geringerer Standardabweichung betrachtet werden. Dabei sollte jedoch jederzeit berücksichtigt werden, dass die Exploration durch das Sampling aus der Verteilung gewährleistet werden muss.

Mit Deep Deterministic Policy Gradient wurde ein RL-Verfahren gefunden, mit welchem die Performanz des Fahrzeugs deutlich optimiert wurde. Die Methode bietet den Vorteil, dass eine deterministische Policy erlernt und entsprechend eine separate Betrachtung der Exploration möglich ist. Durch Verwendung eines Ornstein-Uhlenbeck-Prozesses, welcher ein zielgerichtetes Erkunden des Zustandsraum ermöglicht, konnte mit DDPG ein Agent trainiert werden, welcher als additives Regelglied in der vorhandenen Trajektorienregelung die Querführung des automatisierten Fahrzeugs in der Simulation verbessern konnte.

Ein weiterer Aspekt, welcher genauer untersucht werden sollte, ist die gewählte Reward-

metrik. Da der RL-Algorithmus als additives Stellglied eingesetzt wird, ist der erhaltene Reward nicht ausschließlich von der gewählten Aktion des Agenten abhängig. Neben der aktuell berücksichtigten Aktivität des PID-Reglers sollten die Aktivität des RL-Agenten sowie Unterschiede beider Anteile einbezogen werden, da infolgedessen Einflüsse auf die ausgeführte Handlung und entsprechend den erhaltenen Reward identifiziert werden können. Alternativ kann eine Baseline ermittelt und subtrahiert werden, welche die Rewardanteile des klassischen Regelungskonzeptes repräsentiert.

Ein weiterer zu untersuchender Aspekt ist der Aufbau der künstlichen neuronalen Netze zur Abbildung der Policy sowie der Value-Functions. Im Rahmen dieser Arbeit wurde ein fully-connected KNN mit zwei versteckten Schichten verwendet. Neben weiterer Variation der Aktivierungsfunktionen sowie der Neuronenanzahlen sollten abweichende Architekturen, wie beispielsweise rekurrente Netze, betrachtet werden. Durch das Speichern von Neuronenausgaben in vergangenen Prädiktionen bietet sich dem rekurrenten neuronalen Netz die Möglichkeit zeitliche Zusammenhänge abzubilden. Aufgrund dieser Vergangenheitsinformationen wird das Einbeziehen von Dynamiken möglich, die möglicherweise die Aktionsauswahl verbessern.

Zur weiteren Optimierung der Performanz der RL-Agenten sollte ebenfalls der Zustandsraum genauer untersucht werden. Durch eine umfassendere Analyse der messbaren Fahrzeuggrößen und entsprechend verfügbaren Zustandsgrößen bezüglich deren Einfluss auf die Querführung eines Fahrzeugs ist gegebenenfalls die Verwendung anderer Größen zur Wahl einer Aktion hilfreich.

Mit modellbasierten Ansätzen existieren weitere Verfahren, welche zum Lösen von Reinforcement Learning Problemen eingesetzt werden können. Bei modellbasierten Verfahren lernt der Agent basierend auf der Interaktion der Umwelt ein Modell dieser, wodurch Aktionen langfristiger und entsprechend gewinnbringender geplant werden können. Durch Erlernen eines Modells, welches die Differenz zwischen verwendetem Fahrzeugmodell und dem realen Fahrzeug repräsentiert, entsteht eine neue Option, um die Performanz zu steigern ohne die Sicherheit des Systems zu verlieren.

Zukünftig sollte die vorgestellte Algorithmik in Realfahrversuchen erprobt werden, da erst anhand dessen die simulativ gezeigte Eignung von DDPG zur Verbesserung des bestehenden Trajektorienfolgeregelungskonzeptes bestätigt werden kann.

Literatur

- [1] M. Wagner und B. Dion, „Drive Safely,” ANSYS Advantage, Nr. 1, S. 15–19, 2018. [Online]. URL: <https://www.ansys.com/-/media/ansys/corporate/resourcelibrary/article/ansys-advantage-autonomous-vehicles-aa-v12-i1.pdf>
- [2] S. Pendleton et al., „Perception, Planning, Control, and Coordination for Autonomous Vehicles,” Machines, Vol. 5, Nr. 1, S. 6, 2017.
- [3] K.-J. Kim und I. Tagkopoulos, „Application of machine learning in rheumatic disease research,” The Korean journal of internal medicine, Vol. 34, Nr. 4, S. 708–722, 2019. [Online]. URL: <http://www.kjim.org/upload/kjim-2018-349.pdf>
- [4] R. S. Sutton und A. G. Barto, Reinforcement Learning: An Introduction. Cambridge, Mass. and London: MIT Press, 2018.
- [5] D. Silver, „Lecture 4: Model-Free Prediction,” London, 2015.
- [6] D. Silver, „Lecture 1: Introduction to Reinforcement Learning,” London, 2015.
- [7] K. Osawa, „Introducing K-FAC: A Second-Order Optimization Method for Large-Scale Deep Learning,” 2018. [Online]. URL: <https://towardsdatascience.com/introducing-k-fac-and-its-application-for-large-scale-deep-learning-4e3f9b443414>
- [8] M. Mitschke und H. Wallentowitz, Dynamik der Kraftfahrzeuge, 4. Aufl., Serie VDI. Berlin: Springer, 2004.
- [9] M. Templer, „Untersuchung von Deep Reinforcement Learning zur Potentialsteigerung eines bestehenden Querdynamikregelungskonzeptes,” Masterarbeit, Technische Universität Braunschweig, Braunschweig, 2018.
- [10] DPMA, „Mobilität der Zukunft - Made in Germany: Teil 1: Technik und Perspektiven,” 2019. [Online]. URL: <https://www.dpma.de/dpma/veroeffentlichungen/hintergrund/autonomesfahren-technikteil1/>
- [11] E. Donges, „Fahrerverhaltensmodelle,” in Handbuch Fahrerassistenzsysteme, Serie ATZ/MTZ-Fachbuch, H. Winner et al., Hrsgg. Wiesbaden: Vieweg + Teubner, 2012, Vol. 78, S. 15–23.
- [12] P. Zhao et al., „Design of a Control System for an Autonomous Vehicle Based on Adaptive-PID,” International Journal of Advanced Robotic Systems, Vol. 9, Nr. 2, S. 44, 2012.
- [13] K. Behrendt et al., „A deep learning approach to traffic lights: Detection, tracking, and classification,” in IEEE International Conference on Robotics and Automation (ICRA). Piscataway, NJ: IEEE, 2017, S. 1370–1377.
- [14] M. B. Jensen et al., „Evaluating State-of-the-Art Object Detector on Challenging Traffic Light Data,” in CVPRW 2017. Piscataway, NJ: IEEE, 2017, S. 882–888.
- [15] P. Sermanet und Y. LeCun, „Traffic sign recognition with multi-scale Convolutional Networks,” in The 2011 International Joint Conference on Neural Networks. IEEE, 31.07.2011 - 05.08.2011, S. 2809–2813.

- [16] H. S. Lee und K. Kim, „Simultaneous Traffic Sign Detection and Boundary Estimation Using Convolutional Neural Network,” *IEEE Transactions on Intelligent Transportation Systems*, Vol. 19, Nr. 5, S. 1652–1663, 2018.
- [17] C. You et al., „Advanced planning for autonomous vehicles using reinforcement learning and deep inverse reinforcement learning,” *Robotics and Autonomous Systems*, Vol. 114, S. 1–18, 2019.
- [18] A. Dosovitskiy et al., „CARLA: An Open Urban Driving Simulator.” [Online]. URL: <http://arxiv.org/pdf/1711.03938v1>
- [19] X. Liang et al., „CIRL: Controllable Imitative Reinforcement Learning for Vision-based Self-driving.” [Online]. URL: <http://arxiv.org/pdf/1807.03776v1>
- [20] Q. Khan et al., „Latent Space Reinforcement Learning for Steering Angle Prediction.”
- [21] D. Li et al., „Reinforcement Learning and Deep Learning Based Lateral Control for Autonomous Driving [Application Notes],” *IEEE Computational Intelligence Magazine*, Vol. 14, Nr. 2, S. 83–98, 2019.
- [22] S. Wang et al., „Deep Reinforcement Learning for Autonomous Driving.” [Online]. URL: <http://arxiv.org/pdf/1811.11329v2>
- [23] D. Vissière et al., „Experimental autonomous flight of a small-scaled helicopter using accurate dynamics model and low-cost sensors,” *IFAC Proceedings Volumes*, Vol. 41, Nr. 2, S. 14 642–14 650, 2008.
- [24] P. Abbeel et al., „An Application of Reinforcement Learning to Aerobatic Helicopter Flight,” *Advances in Neural Information Processing Systems*, S. 1–8, 2006.
- [25] R. Tedrake et al., „Learning to walk in 20 minutes,” *Proceedings of the Fourteenth Yale Workshop on Adaptive and Learning Systems*, Vol. 95585, S. 1939–1412, 2005.
- [26] R. Hafner und M. Riedmiller, „Reinforcement learning on a omnidirectional mobile robot,” in *2003 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Piscataway, NJ: IEEE, 2003, S. 418–423.
- [27] M. Riedmiller et al., „Learning to Drive a Real Car in 20 Minutes,” in *2007 Frontiers in the Convergence of Bioscience and Information Technologies*. IEEE, 11.10.2007 - 13.10.2007, S. 645–650.
- [28] L. P. Kaelbling et al., „Reinforcement Learning: A Survey,” *Journal of artificial intelligence research*, Vol. 4, Nr. 4, S. 237–285, 1996.
- [29] A. Burkov, *The hundred-page machine learning book*, 2019.
- [30] R. Rojas, „Vorlesung 5,” Berlin, 2017.
- [31] M. Tiedemann, „Machine Learning Methoden - Teil 2/3: Classification & Regression,” 2019. [Online]. URL: <https://www.alexanderthamm.com/de/artikel/machine-learning-methoden-teil-2-3-classification-regression>
- [32] D. Silver et al., „Mastering the game of Go with deep neural networks and tree search,” *Nature*, Vol. 529, Nr. 7587, S. 484–489, 2016.

- [33] J. Hui, „RL - Reinforcement Learning Terms,” 2019. [Online]. URL: https://medium.com/@jonathan_hui/rl-reinforcement-learning-terms-242baac11907
- [34] M. Ashraf, „Reinforcement Learning Demystified: Markov Decision Processes (Part 1): Episode 2, demystifying Markov Processes, Markov Reward Processes, Bellman Equation, and Markov Decision Processes.” 2018. [Online]. URL: <https://towardsdatascience.com/reinforcement-learning-demystified-markov-decision-processes-part-1-bf00dda41690>
- [35] C. Watkins, „Learning from delayed rewards,” Ph.D. dissertation, 1989.
- [36] C. J. C. H. Watkins und P. Dayan, „Q-learning,” Machine Learning, Vol. 8, Nr. 3-4, S. 279–292, 1992.
- [37] D. Silver, „Lecture 5: Model-Free Control,” London, 2015.
- [38] D. Silver, „Lecture 7: Policy Gradient,” London, 2015.
- [39] A. Sharma, „Understanding Activation Functions in Deep Learning.” [Online]. URL: <https://www.learnopencv.com/understanding-activation-functions-in-deep-learning/>
- [40] K. Rasul, „Machine Learning: Deep Learning: Lecture 11/15,” Berlin, 2018.
- [41] X. Glorot und Y. Bengio, „Understanding the difficulty of training deep feedforward neural networks,” Proceedings of the 13th International Conference on Artificial Intelligence and Statistics, Nr. 9, 2010.
- [42] K. He et al., „Delving Deep into Rectifiers: Surpassing Human-Level Performance on ImageNet Classification,” Proceedings of the IEEE international conference on computer vision, S. 1026–1034, 2015.
- [43] C. Nicholson, „A Beginner’s Guide to Neural Networks and Deep Learning.” [Online]. URL: <https://skymind.ai/wiki/neural-network>
- [44] R. Rojas, „Vorlesung zum Thema Backpropagation,” Berlin, 2017.
- [45] D. P. Kingma und J. Ba, „Adam: A Method for Stochastic Optimization.” [Online]. URL: <https://arxiv.org/pdf/1412.6980.pdf>
- [46] S. Ioffe und C. Szegedy, „Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift.” [Online]. URL: <https://arxiv.org/pdf/1502.03167.pdf>
- [47] N. Srivastava, „Improving Neural Networks with Dropout,” Ph.D. dissertation, University of Toronto, Toronto, 2013. [Online]. URL: http://www.cs.utoronto.ca/~nitish/msc_thesis.pdf
- [48] N. Srivastava et al., „Dropout: A Simple Way to Prevent Neural Networks from Overfitting,” Journal of Machine Learning Research, Vol. 15, S. 1929–1958, 2014. [Online]. URL: <https://www.cs.toronto.edu/~hinton/absps/JMLRdropout.pdf>
- [49] „PyTorch: Github-Repository.” [Online]. URL: <https://github.com/pytorch/pytorch>
- [50] N. Ketkar, „Introduction to PyTorch,” in Deep Learning with Python, Serie For professionals by professionals, N. Ketkar, Hrsg. Berkeley, CA: Apress, 2017, S. 195–208.

- [51] V. Mnih et al., „Asynchronous Methods for Deep Reinforcement Learning.” [Online]. URL: <http://arxiv.org/pdf/1602.01783v2>
- [52] T. P. Lillicrap et al., „Continuous control with deep reinforcement learning.”
- [53] Y. Wu et al., „Scalable trust-region method for deep reinforcement learning using Kronecker-factored approximation.” [Online]. URL: <http://arxiv.org/pdf/1708.05144v2>
- [54] B. O’Donoghue et al., „Combining policy gradient and Q-learning.” [Online]. URL: <http://arxiv.org/pdf/1611.01626v3>
- [55] J. Schulman et al., „Proximal Policy Optimization Algorithms.” [Online]. URL: <http://arxiv.org/pdf/1707.06347v2>
- [56] J. Schulman et al., „Trust Region Policy Optimization,” Proceedings of the 31st International Conference on Machine, S. 1889–1897, 2015. [Online]. URL: <http://arxiv.org/pdf/1502.05477v5>
- [57] M. Zhu et al., „Safe, Efficient, and Comfortable Velocity Control based on Reinforcement Learning for Autonomous Driving.” [Online]. URL: <http://arxiv.org/pdf/1902.00089v1>
- [58] L. Buşoniu et al., „Reinforcement learning for control: Performance, stability, and deep approximators,” Annual Reviews in Control, Vol. 46, S. 8–28, 2018.
- [59] K. Frans, „What is the natural gradient, and how does it work?” 2016. [Online]. URL: <http://kvfrans.com/what-is-the-natural-gradient-and-where-does-it-appear-in-trust-region-policy-optimization/>
- [60] D. Silver et al., „Deterministic Policy Gradient Algorithms,” Proceedings of the 31 st International Conference on Machine, 2014. [Online]. URL: <http://proceedings.mlr.press/v32/silver14.pdf>
- [61] V. Mnih et al., „Human-level control through deep reinforcement learning,” Nature, Vol. 518, S. 529 EP –, 2015.
- [62] G. E. Uhlenbeck und L. S. Ornstein, „On the Theory of the Brownian Motion,” Physical Review, Vol. 36, Nr. 5, S. 823–841, 1930.
- [63] S. Kullback und R. A. Leibler, „On Information and sufficiency,” Annals of Mathematical Statistics, Vol. 22, Nr. 1, S. 79–86, 1951.
- [64] R. Grosse und J. Martens, „A Kronecker-factored approximate Fisher matrix for convolution layers.” [Online]. URL: <https://arxiv.org/pdf/1602.01407.pdf>
- [65] J. Martens und R. Grosse, „Optimizing Neural Networks with Kronecker-factored Approximate Curvature.” [Online]. URL: <https://arxiv.org/pdf/1503.05671.pdf>
- [66] OpenAI, „Getting Started with Gym.” [Online]. URL: <https://gym.openai.com/docs/>
- [67] „MountainCarContinuous v0: Github-Repository.” [Online]. URL: <https://github.com/openai/gym/wiki/MountainCarContinuous-v0>
- [68] R. Isermann, Fahrdynamik-Regelung. Wiesbaden: Vieweg, 2006.

- [69] P. Riekert und T. E. Schunck, „Zur Fahrmechanik des gummibereiften Kraftfahrzeugs,” *Ingenieur-Archiv*, Vol. 11, Nr. 3, S. 210–224, 1940.
- [70] L. Menhour et al., „An efficient model-free setting for longitudinal and lateral vehicle control. Validation through the interconnected pro-SiVIC/RTMaps prototyping platform.” [Online]. URL: <http://arxiv.org/pdf/1705.03216v1>
- [71] Abbildung der Volkswagen Konzernforschung.