HAW HAMBURG

Masterarbeit

Andre Rohden

Stabilisierung unkontrollierter Flugzustände mit Reinforcement Learning

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Andre Rohden

Stabilisierung unkontrollierter Flugzustände mit Reinforcement Learning

Masterarbeit eingereicht im Rahmen der Masterprüfung
im Studiengang Master of Science Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr.-Ing. Andreas Meisel
Zweitgutachter: Prof. Dr. Wolfgang Fohl

Eingereicht am: 25. Januar 2019

Andre Rohden

Thema der Arbeit

Stabilisierung unkontrollierter Flugzustände mit Reinforcement Learning

Stichworte

Reinforcement Learning, Deep Deterministic Policy Gradient, Experience Replay Speicher, Curriculum Learning, Quadcopter

Kurzzusammenfassung

Reinforcement Learning ermöglicht einem selbstlernenden Agenten ein unbemanntes Flugobjekt in unkontrollierten Flugzuständen zu stabilisieren. Um dies zu erreichen, wird ein Deep Deterministic Policy Gradient Algorithmus angewendet. Durch Erweiterung wie Experience Replay Speicher, parametrisiertem Rauschen, *Prioritized Experience Replay*, *Hindsight Experience Replay* und *Curriculum Learning* lassen sich darüberhinaus Umgegebung mit *sparse* Reward trainieren.

Andre Rohden

Title of Thesis

Stabilization of uncontrolled flight states with reinforcement learning

Keywords

reinforcement learning, deep deterministic policy gradient, experience replay memory, curriculum learning, quadcopter

Abstract

Reinforcement learning allows a self-learning agent to stabilize an unmanned aerial vehicle in uncontrolled flight states. To achieve this, a deep deterministic policy gradient algorithm is applied. Through extensions like experience replay memory, parameterized noise, prioritized experience replay, hindsight experience replay and curriculum learning, it is furthermore possible to train environments with sparse reward.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Die Ursache für einen unkontrollierten Flugzustand kann verschiedene Gründe haben. Ein möglicher Grund kann ein Totalausfall eines Quadcopters sein, woraufhin das unbemannte Flugobjekt hinabstürzt. Während des Falles steigt die Geschwindigkeit in Richtung Erde und der Quadcopter beginnt um die eigenen Achsen zu rotieren. Für die Stabilisierung von solchen unkontrollierten Flugzuständen gibt es mehrere Ansätze. Ein professioneller Modellflieger hat nach langem Training erlernt durch ein Flugmanöver die Kontrolle zurück zu erlangen. Zudem gibt es für viele Quadcopter bereits installierte Systeme, die basierend auf Regelungstechnik eine Stabilisierung vornehmen.
In dieser Arbeit wird ein selbstlernender Agent mit Reinforcement Learning entwickelt, der ähnlich einem Modellflieger lernt, wie ein Quadcopter stabilisiert werden kann. Der Vorteil des Agenten ist, dass dieser neue Flugobjekte mit anderer Flugdynamik auf gleiche Weise erlernen kann, ohne das eine Anpassung nötig ist. Des Weiteren kann ein Agent weitaus komplexere Aufgaben teilweise besser lösen als ein menschlicher Experte.

Reinforcement Learning (RL)[1] wird bereits in vielen Gebieten der Robotik eingesetzt, um Probleme zu lösen. Dabei kommen verschiedene Algorithmen für verschiedene Probleme zum Einsatz. Um die Aufgabe der Stabilisierung von unkontrollierten Flugzuständen zu lösen, wird eine Algorithmus benötigt, der über einen kontinuierlichen Zustands- und Aktionsraum verfügt. Aus diesem Grund wird der Deep Deterministic Policy Gradient (DDPG) Algorithmus [2] verwendet, der eine Actor-Critic Architektur besitzt.
Für die Implementierung des Algorithmus wird eine Reward-Funktion benötigt, die das gewünschte Verhalten des Agenten bestimmt. Um eine geeignete Reward-Funktion zu entwerfen, wird das Wissen über die Umgebung und die Dynamik von Objekten benötigt. Dadurch entsteht die Notwendigkeit eines Experten, der über das Wissen verfügt. Um die Menge an benötigtem Wissen zu reduzieren, werden *sparse* Reward-Funktionen eingesetzt. Diese belohnen positive und bestrafen negative Zielzustände. Da die Festlegung von Zielzuständen meist trivial ist, gestaltet sich der Entwurf einer Reward-Funktion in dem Falle einfacher. Jedoch entsteht durch eine *sparse* Reward-Funktionen für den Agenten

das Problem, dass es schwierig wird eine Strategie aufgrund mangelnder Belohnungen zu entwickeln. Um dieses Problem zu lösen kommen verschiedene Techniken wie *Prioritized Experience Replay* (PER) [3], *Hindsight Experience Replay* (HER) [4] und *Curriculum Learning* [5] zum Einsatz.

1.1 Ziele der Arbeit

Das Ziel der Arbeit ist die Implementierung eines selbstlernenden Agenten mit RL, der einen Quadcopter in unkontrollierten Flugzuständen stabilisiert. Dazu werden verschiedene Techniken kombiniert eingesetzt. Insbesondere soll der Agent erlernen in Umgebungen mit *sparse* Reward eine Strategie zu entwickeln.

1.2 Verwandte Arbeiten

In der Arbeit von Hwangbo u.a. [6] wurde ein ähnliches Problem behandelt. Das dort gegebene Szenario ist der Hochwurf eines Quadcopters mit anschließender Stabilisierung. Insbesondere wird die Installation der Strategie in einem echten Quadcopter beschrieben. Der verwendete RL-Algorithmus basiert ebenso auf einer Actor-Critic Architektur, jedoch wird von den Autoren abweichend zu dem Ansatz in dieser Arbeit ein Monte-Carlo Trainingsverfahren für den Critic eingesetzt. Die dargestellten Ergebnisse zeigen, dass die Nutzung des DDPG-Algorithmus nicht erfolgreich war. Zudem wird von den Autoren zur Unterstützung des Trainings ein PD-Controller eingesetzt.
Diese Arbeit unterscheidet sich zum einem durch die Nutzung des DDPG-Algorithmus und zum anderem durch den Verzicht von Hilfsmitteln aus der Regelungstechnik. Zudem kommen hier weitere Methoden und Techniken zum Einsatz, die in Kapitel 3 ausführlich beschrieben werden. Zu diesen gehört unter anderem ein parametrisiertes Rauschen und ein Experience Replay Speicher. Abweichend von der Arbeit der Autoren liegt hier der Schwerpunkt auf der Betrachtung von Umgebungen mit *sparse* Reward.

1.3 Inhaltlicher Aufbau der Arbeit

Zur Simulation der Flugbahnen des Quadcopters wird eine Beschreibung der Flugdynamik benötigt, welche in Kapitel 2 beschrieben wird. Des Weiteren sind dort die Grundla-

gen von Reinforcement Learning erläutert. Kapitel 3 gibt eine detaillierte Beschreibung des implementierten Algorithmus mit allen verwendeten Techniken und Methoden. Der Algorithmus wird in Kapitel 4 durch mehrere Simulationsdurchführungen evaluiert. Im letzten Kapitel 5 ist neben einer Zusammenfassung ein Ausblick für weitere Arbeiten gegeben.

2 Grundlagen

Zur Simulation der Flugbahnen eines Quadcopters wird ein Modell benötigt. In diesem Kapitel wird die Flugdynamik eines Quadcopters beschrieben und das darauf basierende mathematische Modell, welches in Python implementiert ist, vorgestellt.
Reinforcement Learning (RL) wird hier eingesetzt, um einen Agenten das Steuern des Quadcopters erlernen zu lassen. Die Grundlagen von RL werden in diesem Kapitel vorgestellt [1]. Der implementierte Algorithmus baut auf den Grundlagen auf und verwendet weitere Methoden, die im folgenden Kapitel erläutert werden.

2.1 Quadcopter

Ein Quadcopter ist ein unbemanntes Flugobjekt (UAV), dessen Merkmal vier Rotoren auf zwei Achsen sind. Durch die Anordnung der vier Rotoren sind verschiedene Manöver und das Schweben auf der Stelle möglich. Die Rotoren 1 und 3, welche sich auf der lokalen Achse $x' \in E_B$ befinden, drehen sich im Uhrzeigersinn. Während sich die Rotoren 2 und 4 auf der zweiten lokalen Achse $y' \in E_B$ gegen den Uhrzeigersinn drehen. Jeder der vier Rotoren erzeugt durch seine Rotation mit der Winkelgeschwindigkeit ω_i einen Schub F_i in Richtung der lokalen Achse $z' \in E_B$.
Im Inertialsystem E_I ist der Positionsvektor durch $\xi = \begin{pmatrix} x & y & z \end{pmatrix}^T \in E_I$ definiert. Die Orientierung ist im Bezugssystem des Flugkörpers E_B durch die Euler-Winkel mit $\eta = \begin{pmatrix} \phi & \theta & \psi \end{pmatrix}^T \in E_B$ definiert (siehe Abbildung 2.1) [7]. Die Euler-Winkel bestimmen dabei die Rotation um die eigenen Achsen wie folgt: Der Roll-Winkel ϕ um die x'-Achse, der Pitch-Winkel θ um die z'-Achse und der Yaw-Winkel ψ um die z'-Achse. Das Bezugssystem des Flugkörpers E_B hat als Zentrum den Schwerpunkt des Quadcopters und

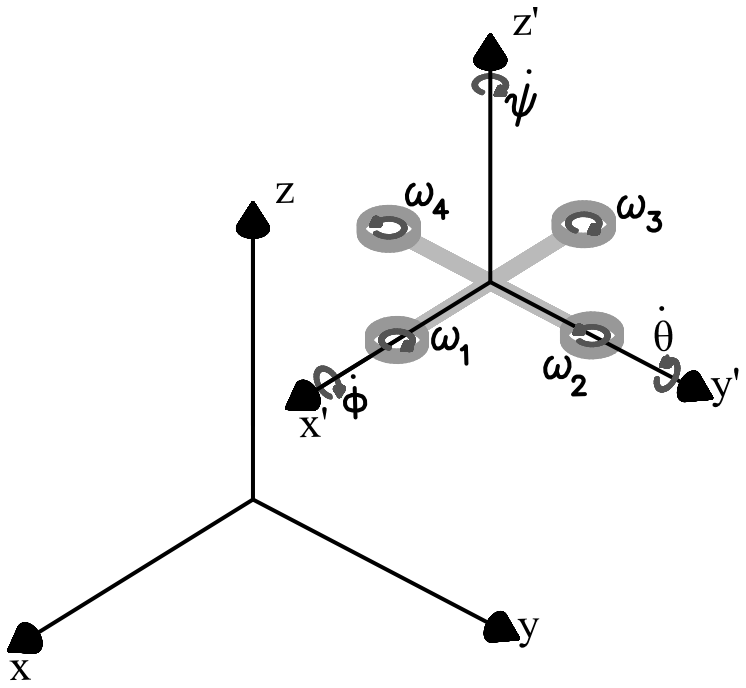


Abbildung 2.1: Position und Orientierung des Quadcopters

kann im Inertialsystem E_I mit der Rotationsmatrix

$$R = \begin{pmatrix} C_\theta C_\psi & S_\phi S_\theta C_\psi - C_\phi S_\psi & C_\phi S_\theta C_\psi + S_\phi S_\psi \\ C_\theta S_\psi & S_\phi S_\theta S_\psi + C_\phi C_\psi & C_\phi S_\theta S_\psi - S_\phi C_\psi \\ -S_\theta & S_\phi C_\theta & C_\phi C_\theta \end{pmatrix} \tag{2.1}$$

beschrieben werden, wobei $S_x = \sin(x)$ und $C_x = \cos(x)$ gilt.

2.1.1 Beschreibung der Umgebung

Die Umgebung, in der sich der Quadcopter befindet, ist ein unendlich großer, luftgefüllter Raum. Daher gibt es keine Objekte mit denen der Quadcopter in Kontakt kommen kann. Da es weder Boden noch Decke gibt, entfallen auch Effekte, die normalerweise in Boden- oder Deckenhöhe auftreten können. Des Weiteren gibt es keine Luftströmungen in dem Raum, die der Quadcopter ausgleichen muss. Die Flugdynamik basiert auf der Schwerkraft, der Schubkraft, dem Strömungswiderstand und den aerodynamischen Effekten, die im nächsten Abschnitt beschrieben werden.

Die Steuerung der Rotoren erfolgt direkt über deren Winkelgeschwindigkeiten ω_i. Dies bedeutet, dass kein Rotormotor simuliert wird, sondern die Winkelgeschwindigkeiten direkt gesetzt werden. Daher sind Sprünge in der Verlaufskurve der Winkelgeschwindigkeiten möglich.

2.1.2 Beschreibung der Flugdynamik

Durch Verändern der Differenzen zwischen den Winkelgeschwindigkeiten der Rotoren lässt sich der Quadcopter rollen (*roll*), nicken (*pitch*) und gieren (*yaw*). Die Abbildung 2.2 zeigt die verschiedenen Flugmänover, wobei Rotoren mit der gleichen Pfeilfarbe (schwarz, rot, und grün) die gleiche Rotorgeschwindigkeit haben. Für die Unterschiede der Rotoren gilt: grün > schwarz > rot.

Die Flugdynamik lässt sich mit Hilfe der Newton-Euler Methoden wie folgt beschreiben [8][9]:

$$m\ddot{\xi} = F_t + F_d + F_g \tag{2.2}$$

$$I\ddot{\eta} = -\dot{\eta} \times I\dot{\eta} + \Gamma_g + \tau \tag{2.3}$$

Die Kraft F_t entsteht durch den Schub F_z von allen vier Rotoren. Da die Kraft F_z im Bezugssystem E_B in z'-Richtung wirkt, wird diese durch R rotiert.

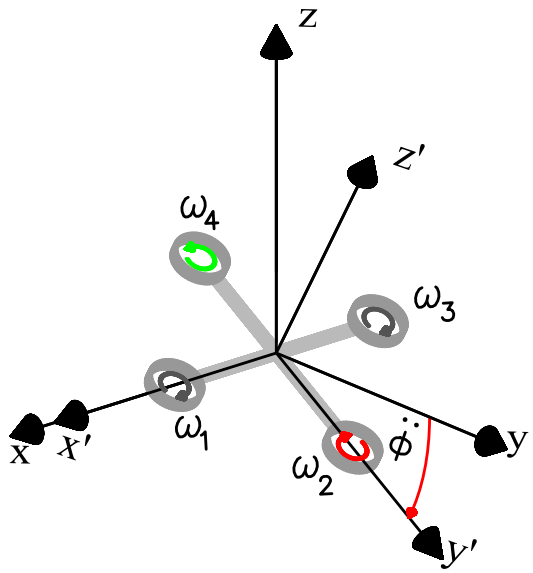
$$F_t = R \begin{pmatrix} 0 \\ 0 \\ F_z \end{pmatrix} \tag{2.4}$$

Die Schubkraft F_i der einzelnen Rotoren beträgt

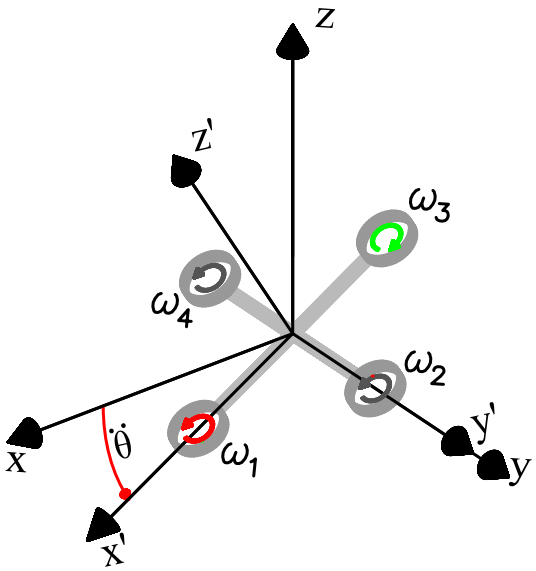
$$F_i = k_t \omega_i^2, \tag{2.5}$$

wobei k_t die Schubkonstante der Rotoren ist. Hiermit berechnet sich die gemeinsame Schubkraft F_z wie folgt:

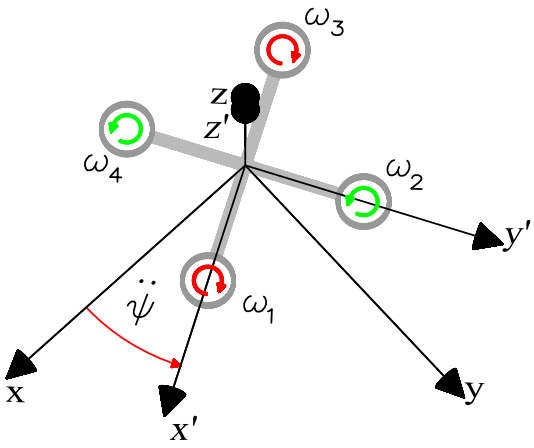
$$F_z = \sum_{i=1}^{4} F_i = k_t(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \tag{2.6}$$



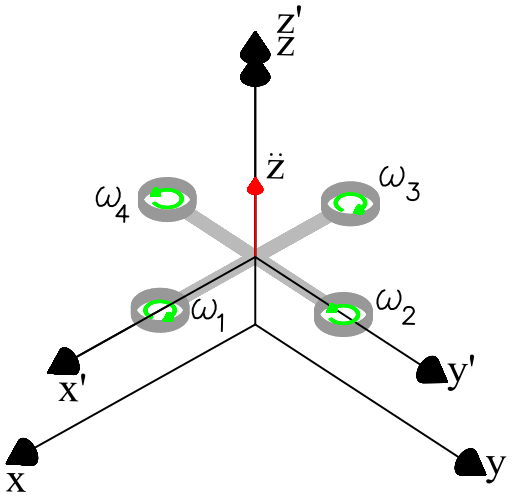
(a) Rollen: $\omega_4 > \omega_1 = \omega_3 > \omega_2$



(b) Neigen: $\omega_3 > \omega_2 = \omega_4 > \omega_1$



(c) Gieren: $\omega_2 = \omega_4 > \omega_1 = \omega_3$



(d) Schub: $\omega_1 = \omega_2 = \omega_3 = \omega_4$

Abbildung 2.2: Flugmänover

Die Kraft F_d entsteht durch den Strömungswiderstand und ist abhängig von der Geschwindigkeit des Quadcopters.

$$F_d = \begin{pmatrix} -k_{dx} & 0 & 0 \\ 0 & -k_{dy} & 0 \\ 0 & 0 & -k_{dz} \end{pmatrix} \dot{\xi} \tag{2.7}$$

k_{dx}, k_{dy} und k_{dz} sind die translatorischen Strömungswiderstandsfaktoren. Diese Faktoren bilden sich bezüglich der jeweiligen Richtung aus dem Strömungswiderstandskoeffizienten c_w, der Dichte p und der Fläche A wie folgt:

$$k_{d_i} = c_{w_i} A_i \frac{1}{2} p \tag{2.8}$$

Die Kraft F_g ist die Gravitationskraft

$$F_g = \begin{pmatrix} 0 \\ 0 \\ -mg \end{pmatrix}, \tag{2.9}$$

wobei g die Gravitation und m die Masse des Quadcopters ist.

Die Matrix I beinhaltet die Trägheitsmomente bezüglich der Rotation um die jeweilige Achse.

$$I = \begin{pmatrix} I_{xx} & 0 & 0 \\ 0 & I_{yy} & 0 \\ 0 & 0 & I_{zz} \end{pmatrix} \tag{2.10}$$

Das Moment Γ_g entsteht durch den gyroskopischen Effekt

$$\Gamma_g = I_r \begin{pmatrix} \dot{\theta}/I_{xx} \\ -\dot{\phi}/I_{yy} \\ 0 \end{pmatrix} \omega_r, \tag{2.11}$$

wobei I_r das Trägheitsmoment der Rotoren und ω_r die Winkelgeschwindigkeit von allen Rotoren ist.

$$\omega_r = -\omega_1 + \omega_2 - \omega_3 + \omega_4 \tag{2.12}$$

Das Drehmoment τ ergibt sich aus folgenden Drehmomenten für jede Achse

$$\begin{pmatrix} \tau_x \\ \tau_y \\ \tau_z \end{pmatrix} = \begin{pmatrix} lk_t(\omega_4^2 - \omega_2^2) \\ lk_t(\omega_3^2 - \omega_1^2) \\ k_d(-\omega_1^2 + \omega_2^2 - \omega_3^2 + \omega_4^2) \end{pmatrix}, \tag{2.13}$$

wobei k_d der rotatorische Strömungswiderstandsfaktor ist. Der Abstand zwischen Schwerpunkt und Rotor wird durch l gemessen.

Durch Einsetzen der Kräfte und Momente in die Gleichungen 2.2 und 2.3 erhält man folgende sechs Differentialgleichungen, die als mathematisches Modell für den Quadcopter genutzt werden.

$$\begin{aligned}
\ddot{x} &= \frac{1}{m}[(\cos(\phi)\sin(\theta)\cos(\psi) + \sin(\phi)\sin(\psi))F_z - k_{dx}\dot{x}] \\
\ddot{y} &= \frac{1}{m}[(\cos(\phi)\sin(\theta)\sin(\psi) - \sin(\phi)\cos(\psi))F_z - k_{dy}\dot{y}] \\
\ddot{z} &= \frac{1}{m}[(\cos(\phi)\cos(\theta))F_z - k_{dz}\dot{z}] - g \\
\ddot{\phi} &= \frac{1}{I_{xx}}[(I_{yy} - I_{zz})\dot{\theta}\dot{\psi} - I_r\dot{\theta}\omega_r + \tau_x] \\
\ddot{\theta} &= \frac{1}{I_{yy}}[(I_{zz} - I_{xx})\dot{\phi}\dot{\psi} + I_r\dot{\phi}\omega_r + \tau_y] \\
\ddot{\psi} &= \frac{1}{I_{zz}}[(I_{xx} - I_{yy})\dot{\phi}\dot{\theta} + \tau_z]
\end{aligned} \tag{2.14}$$

Die Schubkraft F_z und die Drehmomente τ_x, τ_y und τ_z sind hier durch die Gleichungen 2.6 und 2.13 gegeben und abhängig von den Winkelgeschwindigkeiten ω_i der vier Rotoren. In Tabelle 2.1 sind alle verwendeten Parameter definiert [10].

2.1.3 Umsetzung und graphische Darstellung in Python

Das vorgestellte mathematische Modell eines Quadcopters wird verwendet, um die Flugdynamik zu simulieren. Es wird schrittweise simuliert, wobei ein Schritt jeweils $0{,}05s$ beträgt. Zur Simulation eines Schrittes wird der aktuelle Zustand s mit Position und Rotation und dessen Geschwindigkeiten benötigt. Konkret: $s = [x, y, z, \dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}]$. Zudem werden als Eingabe die Winkelgeschwindigkeiten ω_1, ω_2, ω_3 und ω_4 der vier Rotoren eingesetzt. Zur Berechnung des Folgezustandes s' werden die sechs Differential-

Tabelle 2.1: Definition der Parameter für die Flugdynamik

Symbol	Beschreibung	Wert	Einheit
m	Maße des Quadcopters	0,65	kg
l	Abstand zwischen Schwerpunkt und Rotor	0,23	m
k_d	rotatorischer Strömungswiderstandsfaktor	$7{,}5 \cdot 10^{-7}$	$kg\ m^2$
k_t	Schubkonstante	$3{,}13 \cdot 10^{-5}$	$kg\ m$
g	Gravitation	9,81	m/s^2
I_r	Trägheitsmoment der Rotoren	$6{,}0 \cdot 10^{-5}$	$kg\ m^2$
I_{xx}	Trägheitsmoment bei Rotation um die x-Achse	$7{,}5 \cdot 10^{-3}$	$kg\ m^2$
I_{yy}	Trägheitsmoment bei Rotation um die y-Achse	$7{,}5 \cdot 10^{-3}$	$kg\ m^2$
I_{zz}	Trägheitsmoment bei Rotation um die z-Achse	$1{,}3 \cdot 10^{-2}$	$kg\ m^2$
k_{dx}	Strömungswiderstandsfaktor in x-Richtung	0,1	kg/s
k_{dy}	Strömungswiderstandsfaktor in y-Richtung	0,1	kg/s
k_{dz}	Strömungswiderstandsfaktor in z-Richtung	0,1	kg/s

gleichungen 2.14 gelöst. Das Lösen der Gleichungen übernimmt die Funktion *odeint*[1] aus der Python-Bibliothek *SciPy*.

Der Flugverlauf des Quadcopters wird abgespeichert und kann zu jeder Zeit abgefragt werden. Für jeden Zeitpunkt t des Verlaufes gibt es folgende Information: x, y, z, $\dot{x}$, $\dot{y}$, $\dot{z}$, $\ddot{x}$, $\ddot{y}$, $\ddot{z}$, ϕ, θ, ψ, $\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$, $\ddot{\phi}$, $\ddot{\theta}$, $\ddot{\psi}$, ω_1, ω_2, ω_3 und ω_4. Diese können jeweils durch einen Graphen über die Zeitachse aufgetragen werden. Die verwendeten Graphen werden durch die Python-Bibliothek *matplotlib* bereitgestellt.
Eine anschaulichere Darstellung ist durch die Animation des Quadcopters, wie in Abbildung 2.3 gezeigt, gegeben. In einem 3D-Graphen wird die Position des Schwerpunktes vom Quadcopter mit einem Punkt markiert. Die Positionen der vier Rotoren werden abhängig von der Orientierung des Quadcopters ebenfalls markiert. Anschließend werden alle Rotoren mit dem Schwerpunkt durch Linien verbunden. Mit Hilfe der Klasse *animation* wird für jeden Zeitpunkt des Flugverlaufes eine Darstellung erzeugt. Durch Abspielen der Animation in Echtzeit wird die Flugbahn visualisiert.

[1] https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.odeint.htm (siehe CD)l

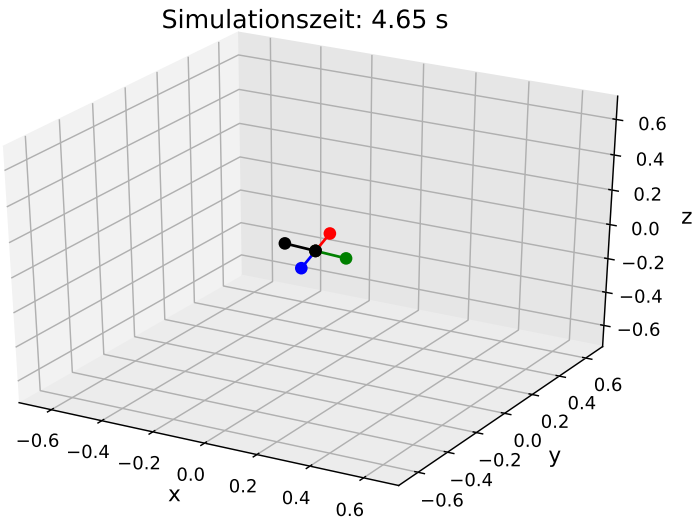


Abbildung 2.3: Flugsimulation

2.2 Reinforcement Learning

Reinforcement Learning beruht auf der Idee Erfahrungen zu sammeln und aus diesen zu lernen. Ein anschauliches Beispiel ist der Prozess wie ein Kleinkind das Laufen erlernt. Nur durch immer erneute Wiederholung einer Aufgabe lernt ein Agent diese optimal auszuführen. Der Lernprozess erfolgt durch Bestrafungen und Belohnungen. Beim Menschen kann die Belohnung zum Beispiel durch Lob und dadurch ausgelöste Glückshormone erfolgen. Die Bestrafung zum Beispiel durch körperliche Schmerzen, die durch einen Sturz entstehen. Beim maschinellen Lernen muss sich anderer Techniken für Bestrafung und Belohnung bedient werden, die im nächsten Abschnitt anhand des Agentenmodells verdeutlicht werden.
Für Reinforcement Learning wird kein Lehrer wie bei überwachten Lernverfahren verwendet, welche gewünschte Ein- und Ausgabewerte beim Lernen vorgeben. Dies hat den Vorteil, dass die Umgebung vorher nicht bekannt sein muss.

2.2.1 Agentenmodell

Das Agentenmodell (siehe Abb. 2.4) basiert auf einem Markow-Entscheidungsproblem (MDP) [1]. Der lernende Agent agiert durch Aktionen mit seiner Umgebung. Zu einem Zeitpunkt t befindet sich der Agent in einem Zustand $s_t \in \mathcal{S}$ und hat in diesem verschiedene Aktionen $a_t \in \mathcal{A}$ zur Auswahl. Durch Ausführen einer Aktion a_t gelangt der

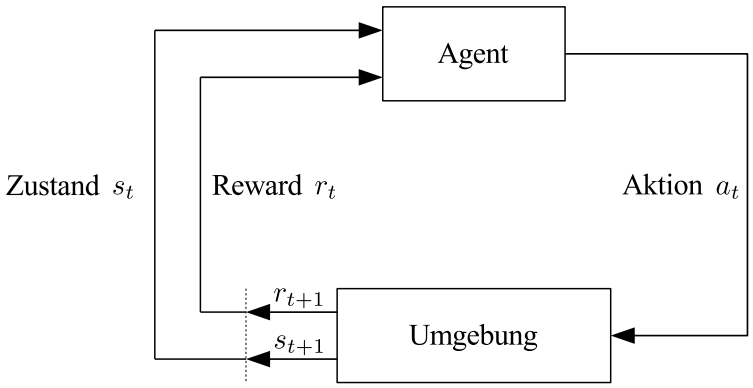


Abbildung 2.4: Agentenmodell

Agent in einen Folgezustand s_{t+1} und erhält eine direkte Belohnung $r_t = r(s_t, a_t)$. Die Übergangsfunktion wird mit $\delta(s_t, a_t) = s_{t+1}$ bezeichnet und gibt den Nachfolgezustand s_{t+1} für die Aktion a_t im Zustand s_t zurück. Die Funktion δ ist hier deterministisch und basiert auf dem mathematischen Modell des Quadcopters aus Abschnitt 2.1.2.

2.2.2 Reward

Die Belohnung für eine Aktion in einem Zustand (Zustand-Aktions-Paar), die der Agent direkt erhält nennt man direkte Belohnung (*immediate reward*). Für den Agenten ist aber vor allem die Gesamtbelohnung (*return*) von Bedeutung. Die Gesamtbelohnung R_t zum Zeitpunkt t ergibt sich aus der Addition der direkten Belohnung und aller zukünftigen Belohnungen:

$$R_t = r_{t+1} + \gamma \cdot r_{t+2} + \gamma^2 \cdot r_{t+3} + ... = \sum_{k=0}^{T} \gamma^i \cdot r_{t+k+1} \tag{2.15}$$

Der Diskontierungsfaktor $\gamma \in [0, 1]$ bestimmt, wie sehr zukünftige Belohnungen mit in die Gewichtung der Gesamtbelohnung eingehen. Ein hoher γ-Wert bedeutet dementsprechend, dass Belohnungen, die in ferner Zukunft liegen, mit in die Bewertung eines Zustand-Aktions-Paares einbezogen werden.

2.2.3 Strategie

Eine Strategie ist eine Funktion $\pi : \mathcal{S} \rightarrow \mathcal{A}$, die zu einem Zustand $s_t \in \mathcal{S}$ eine Aktion $a_t \in \mathcal{A}$ liefert. Somit bestimmt eine Strategie das Verhalten des Agenten.
Optimal ist eine Strategie, wenn sie durch die gewählten Aktionen eine maximale Gesamtbelohnung erreicht. Somit ist es erstrebenswert eine optimale Strategie zu finden.

2.2.4 Bewertungsfunktionen

Bewertungsfunktionen geben eine Aussage über die erwartete Gesamtbelohnung. Hier wird zwischen der Zustand-Wert-Funktion und der Aktion-Wert-Funktion unterschieden. Die Zustand-Wert-Funktion (V-Funktion) gibt für jeden Zustand $s_t \in \mathcal{S}$ an, wie hoch die erwartetet Gesamtbelohnung in diesem Zustand ist, wenn die Strategie π befolgt wird (siehe Gleichung. 2.16).

$$V_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^i \cdot r_{t+k+i} | s_t = s \right] \tag{2.16}$$

Die Aktion-Wert-Funktion (Q-Funktion) bewertet Zustand-Aktions-Paare und trifft eine Aussage über die Gesamtbelohnung, die für das Ausführen der Aktion $a \in \mathcal{A}$ im Zustand $s \in \mathcal{S}$ bezüglich der Strategie π erwartet wird. Die Gleichung 2.17 wird auch *Bellman equation* genannt.

$$Q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{k=0}^{T} \gamma^i \cdot r_{t+k+i} | s_t = s, a_t = a \right] \tag{2.17}$$

2.2.5 RL Algorithmen

Es gibt verschiedene Algorithmen beim Reinforcement Learning, um eine optimale Strategie zu finden. Hier kommt der DDPG-Algorithmus zum Einsatz. Dies ist ein Actor-Critic Algorithmus und gehört zu der Gruppe der Temporal Difference (TD) Learning Methoden. TD Methoden vergleichen Zustände mit einer kleinen zeitlichen Differenz. Es wird also ein Zustand s_t mit seinem Folgezustand s_{t+1} und den dazugehörigen Aktionen verglichen und je nach gewählter Methode erfolgt eine Bewertung über das Zustand-Aktions-Paar.

Um die Funktionsweise des DDPG-Algorithmus in Abschnitt 3.2.1 erklären zu können, wird zunächst der Q-Learning Algorithmus vorgestellt. Dies ist eine Critic-only Methode, die als Grundlage für den Critic im DDPG-Algorithmus dient. Anhand von Q-Learning lässt sich die Bedeutung der einzelnen Parameter und die Notwendigkeit eines neuronalen Netzes erläutern.

Q-Learning

Jedem Zustand-Aktions-Paar wird ein Q-Wert zugeordnet. In der einfachen Variante geschieht dies in einer Tabelle, welche alle Q-Werte abspeichert. Diese Q-Werte bewerten, wie gut eine Aktion a_t in einem Zustand s_t ist. Die Güte ist dabei durch die Höhe der möglichen Gesamtbelohnung definiert. Eine Strategie kann aufgrund dieser Werte eine Aktionsentscheidung für jeden Zustand treffen.

In jeder Iteration des Algorithmus wird der Q-Wert des aktuellen Zustand-Aktions-Paares aktualisiert. Die Aktualisierungsfunktion baut auf der Q-Funktion in Gleichung 2.17 auf. Mit jeder Iteration konvergiert der Q-Wert zur tatsächlichen Gesamtbelohnung.

$$Q_\pi(s_t, a_t) \leftarrow (1-\alpha) \cdot Q_\pi(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot \max_a Q_\pi(s_{t+1}, a)) \tag{2.18}$$

Die Gleichung 2.18 zeigt die Aktualisierung des Q-Wertes zum Zeitpunkt t für die Aktion a_t im Zustand s_t bezüglich der Strategie π. Durch die Ausführung der Aktion a_t gelangt der Agent in den Folgezustand s_{t+1} und erhält eine direkte Belohnung r_t.
Die Lernrate α steuert den Anteil der aktuellen Q-Werte und der neu gewonnenen Informationen, die in die Aktualisierung einfließen. Die neu gewonnenen Informationen bilden sich aus der direkten Belohnung r_t und dem maximal möglichen Q-Wert für den Folgezustand zusammen. Wie in Abschnitt 2.2.2 beschrieben, entspricht dies der möglichen Gesamtbelohnung, die abhängig vom Diskontierungsfaktor γ ist. Die Lernrate kann in dieser Umgebung $\alpha = 1{,}0$ gesetzt werden, da die Übergangsfunktion σ und die Reward-Funktion r deterministisch sind. In nicht-deterministischen Umgebungen ist es sinnvoll $\alpha < 1{,}0$ zu setzen, um Fehler, die durch stochastische Funktionen entstehen, auszugleichen.

2.2.6 Neuronale Netze

Ein großes Problem beim Reinforcement Learning ist die Dimensionalität des Zustandsraumes. Mit jeder Dimension wächst der Zustandsraum exponentiell. Zum Speichern der Q-Werte werden daher Tabellen schnell unbrauchbar, da sie zu groß werden. Ein weiterer großen Nachteil, den Tabellen haben, ist, dass sie nur mit diskreten Zustands- und Aktionswerten umgehen können. Dadurch würde es länger dauern bis jeder Q-Wert konvergiert.
Eine Lösung ist es die Q-Werte in einem neuronalen Netz zu speichern. Die Berechnung der Q-Werte bleibt dabei gleich. Beim Deep Q-Learning heißt ein solches Netz Deep Q-Network (DQN).

Künstliche neuronale Netze sind eine Abbildung der Funktionsweise des Gehirns. Neuronen, die in hierarchischen Schichten angeordnet sind, bilden ein Netz, welches Schichtweise miteinander verbunden ist. Legt man an der Eingangsschicht eines neuronalen Netzes ein Eingangssignal an, dann kommt durch die Vernetzung der Neuronen an der Ausgangsschicht ein Ausgangssignal wieder heraus. Man kann neuronale Netz so trainieren, dass ein Eingangssignal ein gewünschtes Ausgangssignal erzeugt. Beim Training werden Eingangs- und Ausgangssignal vorgegeben und die Neuronen so modifiziert, dass sich das Netz wie gewünscht verhält.

2.2.7 Target-Netz

Bei den meisten RL-Algorithmen, die mit neuronalen Netzen arbeiten, wird für jedes Netz ein weiteres Target-Netz verwendet. Dieses ist eine Kopie des eigentlichen Netzes und dementsprechend identisch in der Architektur und Initialisierung der Parameter. Sei beim Deep Q-Learning das Netz Q mit den Parametern θ^Q gegeben, dann ist dessen Target-Netz Q' mit den Parametern $\theta^{Q'}$. Die Aktualisierungsfunktion aus Gleichung 2.18 wird bei Verwendung eines Target-Netzes zu

$$Q_\pi(s_t, a_t|\theta^Q) \leftarrow r_t + \gamma \cdot \max_a Q'_\pi(s_{t+1}, a|\theta^{Q'}), \tag{2.19}$$

wobei die Lernrate $\alpha = 1{,}0$ gesetzt wird. Nach der Aktualisierung des Netzes Q wird das Target-Netz Q' langsam dem Netz Q angeglichen. Dies geschieht über die Aktualisierung der Parameter der Netze und dem Faktor τ mit:

$$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \tag{2.20}$$

Die Parameter der Netze bestehen aus den Gewichten und den Bias von allen Neuronen.

Target-Netze werden zur Stabilisierung des Trainings eingesetzt. Durch das langsame Annähern des Target-Netzes an das Trainingsnetz werden eventuelle Fehler oder Schwankungen nicht sofort mit in die Berechnung des Q-Wertes übernommen.

3 Algorithmus

In diesem Kapitel wird der Deep Deterministic Policy Gradient (DDPG) Algorithmus und dessen Implementierung vorgestellt. Um eine gute Lösung für die Architektur der Netze und die Details des Algorithmus festlegen zu können, wird zunächst eine Beschreibung des Problems benötigt.

3.1 Problembeschreibung

Abhängig von dem Problem gestalten sich Zustands- und Aktionsraum für den gewählten Algorithmus. Durch die Beschreibung des Problems wird auch das Ziel der Aufgabe definiert. Diese Informationen über das Ziel sind notwendig, um mögliche Definitionen der Reward-Funktion erarbeiten zu können. Auch andere Details des Algorithmus werden erst nach der Festlegung des gewünschten Zieles möglich. Dazu gehören unter anderem die Netzarchitektur, das Verhalten des Experience Replay Speichers, die gewählte Strategie und weitere.
Zunächst wird das Problem und die dafür notwendigen Zustands- und Aktionsräume erläutert. Weitere Details zum Algorithmus können anschließend festgelegt werden.

3.1.1 Problem

Das hier gegebene Problem besteht in der Stabilisierung eines Quadcopters. Es sollen unkontrollierte Flugzustände stabilisiert werden. Solche Zustände entstehen zum Beispiel bei einem Zusammenstoß mit einem anderem Objekt oder einem kurzzeitigen Rotorausfall. Der dadurch entstehende Absturz ist unkontrolliert. Dies bedeutet, dass sich der Quadcopter mit erhöhter Geschwindigkeit in der Umgebung bewegt und auch die Ausrichtung stark rotiert. Sobald die Rotoren nach dem Zwischenfall wieder steuerbar sind, muss der Quadcopter wieder stabilisiert werden. Das bedeutet, dass die Rotationsgeschwindigkeiten um die eigenen Achsen reduziert werden müssen, bis eine dauerhafte

Ausrichtung nach oben wieder gegeben ist und anschließend die Geschwindigkeit des Quadcopters auf Null gebracht werden kann. Die genaue Endposition ist hierbei nicht von Interesse, da lediglich eine stabile Lage gefunden werden soll.

3.1.2 Aktionsraum

Steuerbare Faktoren des Quadcopters sind die Winkelgeschwindigkeiten der vier Rotoren $\omega = \begin{pmatrix} \omega_1 & \omega_2 & \omega_3 & \omega_4 \end{pmatrix}^T$. Aufgrund der Definition der Flugdynamik und der Umgebung bestimmen allein diese Werte die Flugbahn des Quadcopter (siehe Abschnitte 2.1.2 und 2.1.1). Da es keine Simulation der Rotormotoren gibt, werden durch eine Aktion alle vier Winkelgeschwindigkeiten direkt gesetzt. Der DDPG-Algorithmus ermöglicht dafür einen kontinuierlichen Aktionsraum.

Zur Steuerung der Rotorwinkelgeschwindigkeiten ω liefert der Actor eine Aktion a_t. Der Actor verfügt über i Dimensionen am Ausgang mit den Werten $a_{t_i} \in [-1, +1]$. Es werden zwei Varianten vorgestellt und in Abschnitt 4.2.3 verglichen. Die erste Variante verfügt über vier Dimensionen und die zweite Variante über fünf Dimensionen. Die beiden Varianten sind mit den Konstanten $g_\omega = 50\frac{1}{s}$, $k_\omega = 5\frac{1}{s}$ und dem Basiswert $b_\omega = 225{,}678\frac{1}{s}$ wie folgt definiert:

Variante A mit Aktion $a_t = \begin{pmatrix} a_{t_1} & a_{t_2} & a_{t_3} & a_{t_4} \end{pmatrix}^T$ (4 Dimensionen):

$$\omega = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix} = \begin{pmatrix} b_\omega + a_1 \cdot g_\omega \\ b_\omega + a_2 \cdot g_\omega \\ b_\omega + a_3 \cdot g_\omega \\ b_\omega + a_4 \cdot g_\omega \end{pmatrix} \tag{3.1}$$

Variante B mit Aktion $a_t = \begin{pmatrix} a_{t_1} & a_{t_2} & a_{t_3} & a_{t_4} & a_{t_5} \end{pmatrix}^T$ (5 Dimensionen):

$$\omega = \begin{pmatrix} \omega_1 \\ \omega_2 \\ \omega_3 \\ \omega_4 \end{pmatrix} = \begin{pmatrix} b_\omega + a_5 \cdot g_\omega + a_1 \cdot k_\omega \\ b_\omega + a_5 \cdot g_\omega + a_2 \cdot k_\omega \\ b_\omega + a_5 \cdot g_\omega + a_3 \cdot k_\omega \\ b_\omega + a_5 \cdot g_\omega + a_4 \cdot k_\omega \end{pmatrix} \tag{3.2}$$

Aufgrund der Netzarchitektur gilt für die Werte $|a_{t_i}| \leq 1$. Die Werte für die Winkelgeschwindigkeiten der Rotoren liegen jedoch in stabiler Position bei circa $225\frac{1}{s}$. Deswegen

wird der Basiswert b_ω zum Ausgleich verwendet. Zur Definition des Basiswertes wird eine stabile Position vorausgesetzt. Das bedeutet, dass keine Rotation und keine Bewegung des Quadcopters stattfindet und dieser nach oben ausgerichtet ist. Alle vier Rotoren drehen sich dann mit gleicher Geschwindigkeit, sodass keine Manöver zur Seite möglich sind. Somit kann durch die Winkelgeschwindigkeit lediglich die Höhe verändert werden. Für die Gleichung aus 2.14 ergibt sich mit den Bedingungen $\ddot{z} = 0\frac{m}{s^2}$, $\dot{z} = 0\frac{m}{s}$, $\phi = 0°$ und $\theta = 0°$:

$$\begin{aligned} \ddot{z} &= \frac{1}{m}[(\cos(\phi)\cos(\theta))F_z - k_{dz}\dot{z}] - g \\ \Leftrightarrow F_z &= m \cdot g \end{aligned} \tag{3.3}$$

Da die vier Winkelgeschwindigkeiten den gleichen Wert haben ($\omega_1 = \omega_2 = \omega_3 = \omega_4$) folgt nach Gleichung 2.6:

$$\begin{aligned} F_z &= k_f(\omega_1^2 + \omega_2^2 + \omega_3^2 + \omega_4^2) \\ &= k_f \cdot 4 \cdot \omega_1^2 \end{aligned} \tag{3.4}$$

Setzt man Gleichung 3.4 in Gleichung 3.3 ein, so ergibt sich:

$$\begin{aligned} k_f \cdot 4 \cdot \omega_1^2 &= m \cdot g \\ \Leftrightarrow \omega_1^2 &= \frac{m \cdot g}{4 \cdot k_f} \\ \Leftrightarrow \omega_1 &= \sqrt{\frac{m \cdot g}{4 \cdot k_f}} \end{aligned} \tag{3.5}$$

Durch Einsetzen der Parameterwerte aus Tabelle 2.1 ergibt sich für den Basiswert auf drei Nachkommastellen gerundet: $b_\omega = 225{,}678\frac{1}{s}$

Die Konstante g_ω ermöglicht den einzelnen Winkelgeschwindigkeiten um $\pm 50\frac{1}{s}$ vom Basiswert b_ω abzuweichen. Bei Variante A können dadurch die Differenzen der einzelnen Rotoren sehr hoch sein. In Variante B wird die Konstante g_ω auf alle vier Rotoren gleichermaßen angewendet und die kleinere Konstante k_ω definiert die möglichen Differenzen der Winkelgeschwindigkeiten.
Variante B benötigt zwar mehr Dimensionen, jedoch ist die Vermutung, dass durch die kleineren Differenzen zwischen den einzelnen Rotoren ein insgesamt stabilerer Flugverlauf stattfindet. Diese Vermutung wird in Abschnitt 4.2.3 evaluiert.

3.1.3 Zustandsraum

Durch den vorgegebenen Aktionsraum sind Manöver in allen Dimensionen des Quadcopters möglich. Daher sollte der Zustandsraum im Idealfall auch alle Dimensionen abdecken. Aufgrund der Problemdefinition braucht die genaue Position im Raum nicht beachtet werden, da lediglich eine stabile Lage gefordert ist. Dadurch erhält man folgenden Zustandsraum: $[\dot{x}, \dot{y}, \dot{z}, \ddot{x}, \ddot{y}, \ddot{z}, \phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}, \ddot{\phi}, \ddot{\theta}, \ddot{\psi}]$ (15 Dimensionen)

Eines der größten Probleme beim RL und allgemein beim Trainieren von neuronalen Netzen ist die Dimensionalität. Zum einen steigt mit jeder zusätzlichen Dimensionen die Trainingszeit stark an und zum anderen sinkt die Wahrscheinlichkeit eines erfolgreichen Trainings. Deswegen ist eine zweite Variante mit einem verkleinerten Zustandsraum hier zum Vergleich gegeben. In diesem wird auf einige Dimensionen verzichtet und ist wie folgt definiert: $[\dot{x}, \dot{y}, \dot{z}, \phi, \theta, \psi, \dot{\phi}, \dot{\theta}, \dot{\psi}]$ (9 Dimensionen)
Da in den Geschwindigkeitsvariablen ($\dot{\xi}$ und $\dot{\eta}$) bereits die notwendigen Informationen enthalten sind, wird auf Beschleunigungsvariablen ($\ddot{\xi}$ und $\ddot{\eta}$) verzichtet. Durch diesen Verzicht wird erwartet, dass der Lernprozess zwar schneller wird, jedoch die Endergebnisse sich verschlechtern. In Abschnitt 4.2.3 werden die beiden Varianten miteinander verglichen.

Um den Wertebereich des Zustandsraumes einzugrenzen werden die Winkel in η wie folgt begrenzt:

$$\phi \leftarrow ((\phi + 180°) \mod 360°) - 180° \tag{3.6a}$$

$$\theta \leftarrow ((\theta + 180°) \mod 360°) - 180° \tag{3.6b}$$

$$\psi \leftarrow ((\psi + 180°) \mod 360°) - 180° \tag{3.6c}$$

Somit ergibt sich $\phi, \theta, \psi \in [-180°, +180°]$ und erleichtert das Training für das Netz. Dies ist möglich, da bei einer vollständigen Rotation um die Achse die Zustände identisch sind.

3.1.4 Episode

Eine Episode beschreibt einen Zeitraum, in dem der lernende Agent in einen Startzustand versetzt wird und dann mit der Umgebung agiert, bis entweder der Zielzustand erreicht ist

oder eine festgelegte Zeit abgelaufen ist. Ein Zielzustand kann dabei sowohl ein positives als auch ein negatives Ziel sein. Das Betreten eines ungewollten Zustandes kann ebenfalls ein Abbruchkriterium für eine Episode sein.

Da der Quadcopter nach einem erfolgreichen Training je nach Ausgangslage circa $3s$ benötigt, um sich zu stabilisieren, sollte eine Episode mindestens solange andauern. Durch den Startzustand einer neuer Episode können neue Flugbahnen entdeckt werden. Daher sollte regelmäßig eine neue Episode gestartet werden. Eine passende Episodenlänge ist mit $10s$ gegeben. Die Simulationsschrittgröße beträgt dabei $0,05s$, was eine geeignete Größe zum Messen der temporären Differenzwerte (TD) zwischen zwei Schritten ist.

Zielzustand

Bei einem Problem, wo der Agent im Schachspiel siegen muss, ist der Zielzustand klar durch einen Sieg oder Niederlage definiert. Bei der Stabilisierung von Flugzuständen ist der gewünschte positive Zielzustand der absolute Stillstand in der Luft. Diesen exakten Zustand zu erreichen ist unwahrscheinlich. Stattdessen könnte ein Zielbereich definiert werden. Jeder Zustand in diesem Zielbereich ist damit ein Zielzustand. Allerdings besteht das Problem mit der Unerreichbarkeit des Zieles bei einem zu kleinem Zielbereich weiterhin. Wird der Zielbereich zu groß definiert, sind alle Zustände am Rand des Zielbereiches keine stabilen Flugzustände. Angenommen die Größe des Zielbereiches ist optimal: Auch dann wird beim ersten Zustand am Rand des Zielbereiches abgebrochen und der Agent lernt nie einen Zustand mit absoluten Stillstand kennen.
Aufgrund der Schwierigkeit einen geeigneten positiven Zielzustand zu definieren, ist eine Episode nach Ablauf einer festgelegten Zeit beendet und somit eine kontinuierliche Aufgabe. Außerdem ist durch die konstante Episodenlänge ein Vergleichen der Episoden bezüglich des Erfolges besser messbar. In Abschnitt 3.2.3 werden zwei Varianten von Reward-Funktionen vorgestellt. Eine Variante benutzt Zielzustände, jedoch ist dies auch dort kein Abbruchkriterium für die Episode. Um stets die gleiche Episodenlänge zu gewährleisten, führt auch ein negativer Zielzustände zu keinem Abbruch der Episode. Ein solcher negativer Zielzustand kann bei Verlassen eines festgelegten Wertebereiches für Position oder Geschwindigkeit definiert sein.

Startzustand

Der Startzustand einer Episode kann beliebig gewählt werden. Am geeignetsten ist es diesen zufällig zu wählen, damit nicht nur ein Weg zum Ziel gefunden wird, sondern eine Erkundung (*Exploration*) möglichst vieler Zustände stattfindet. Die Definition möglicher Startzustände entscheidet dabei über den Schwierigkeitsgrad für den Agenten das Problem zu lösen. Bei einem klein definierten Zustandsbereich in der Nähe des Zieles wird der Agent schneller einen Weg zum Ziel finden, als wenn der Startzustand ein beliebiger Zustand im Raum sein kann. Jedoch ist es sinnvoll den möglichen Startbereich groß zu wählen, damit der Agent lernt in allen Zuständen gute Aktionsentscheidungen zu treffen. Der Schwierigkeitsgrad, der sich durch den Startzustand ergibt, ist in folgende Kategorien eingeteilt:

- Einfach: Startzustände mit geringer Geschwindigkeit und Beschleunigung von Position und Rotation, sowie aufrechter Orientierung.
- Normal: Startzustände mit geringer und hoher Geschwindigkeit und Beschleunigung von Position und Rotation, sowie Orientierung in alle Richtungen.
- Schwierig: Startzustände mit hoher Geschwindigkeit und Beschleunigung von Position und Rotation, sowie Orientierung in alle Richtungen.

Tabelle 3.1: Grenzwert der Startzustände bezüglich des gewählten Schwierigkeitsgrades

	einfach	normal	schwierig
$\dot{\xi}_{max}$	0,01	0,3	3,0
$\ddot{\xi}_{max}$	0,01	0,1	1,0
η_{max}	18	180	180
$\dot{\eta}_{max}$	0,01	0,3	3,0
$\ddot{\eta}_{max}$	0,01	0,1	1,0

Zu Beginn einer Episode wird ein Startzustand zufällig gesetzt. Der Startzustand für jedes Zustandselement ist durch die folgenden Gleichungen gegeben:

$$\xi_{Start} = \begin{pmatrix} x_{Start} \\ y_{Start} \\ z_{Start} \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 0 \end{pmatrix} \cdot 1m \tag{3.7a}$$

$$\dot{\xi}_{Start} = \begin{pmatrix} \dot{x}_{Start} \\ \dot{y}_{Start} \\ \dot{z}_{Start} \end{pmatrix} = random_3(-\dot{\xi}_{max}, \dot{\xi}_{max}) \cdot \frac{1m}{s} \tag{3.7b}$$

$$\ddot{\xi}_{Start} = \begin{pmatrix} \ddot{x}_{Start} \\ \ddot{y}_{Start} \\ \ddot{z}_{Start} \end{pmatrix} = random_3(-\ddot{\xi}_{max}, \ddot{\xi}_{max}) \cdot \frac{1m}{s^2} \tag{3.7c}$$

$$\begin{pmatrix} \phi_{Start} \\ \theta_{Start} \end{pmatrix} = random_2(-\eta_{max}, \eta_{max}) \cdot 1^\circ \tag{3.7d}$$

$$\psi_{Start} = random_1(-180, 180) \cdot 1^\circ \tag{3.7e}$$

$$\dot{\eta}_{Start} = \begin{pmatrix} \dot{\phi}_{Start} \\ \dot{\theta}_{Start} \\ \dot{\psi}_{Start} \end{pmatrix} = random_3(-\dot{\eta}_{max}, \dot{\eta}_{max}) \cdot \frac{1^\circ}{s} \tag{3.7f}$$

$$\ddot{\xi}_{Start} = \begin{pmatrix} \ddot{\phi}_{Start} \\ \ddot{\theta}_{Start} \\ \ddot{\psi}_{Start} \end{pmatrix} = random_3(-\ddot{\eta}_{max}, \ddot{\eta}_{max}) \cdot \frac{1^\circ}{s^2} \tag{3.7g}$$

Die Funktion $random_n(a, b)$ liefert einen n-dimensionalen Vektor mit Zufallszahlen aus den gegebenen Grenzwerten a und b. Dabei sind ξ_{max}, $\dot{\xi}_{max}$, $\ddot{\xi}_{max}$, η_{max}, $\dot{\eta}_{max}$ und $\ddot{\eta}_{max}$ die Grenzwerte für den Zustandsraum. Diese sind in drei Schwierigkeitsgrade eingeteilt und in der Tabelle 3.1 aufgelistet.
Der Startzustand für die Positon x, y und z ist immer im Nullpunkt (siehe Gleichung 3.7a). Diese Werte können auch beliebig gesetzt werden, da sie nicht zum Zustandsraum gehören und auch keine Auswirkung auf eine stabile Lage haben. Jedoch ist es für spätere Evaluierungen unkomplizierter, wenn der Startpunkt immer im Nullpunkt ist. Der Startbereich für den Wert von Yaw-Winkel ψ ist immer der maximal mögliche Bereich, da dieser selbst im Zielbereich alle Winkelstellungen für eine stabile Lage zulässt (siehe Gleichung 3.7e).

Für die Evaluierung im nächsten Kapitel wird der normale Schwierigkeitsgrad gewählt. In Abschnitt 4.2.1 werden die beiden anderen Kategorien ausgewertet. Eine Alternative für die Startzustände bietet *Curriculum Learning*. Mit dieser Methode wird der Schwierigkeitsgrad für den Agenten schrittweise erhöht. Diese Methode wird in Abschnitt 3.2.6 vorgestellt.

3.2 Implementierung des Algorithmus

Der DDPG-Algorithmus wird für diese Problemstellung eingesetzt, da dieser die Aktionen aus einem kontinuierlichen Aktionsraum wählen kann. Zu der Implementierung gehören die Reward-Funktion und das verwendete Rauschen, welche problembezogen angepasst werden müssen. Der Algorithmus verwendet einen Experience Replay Speicher, der hier vorgestellt wird. Zu dem Speicher gehören die Techniken *Prioritized Experience Replay* (PER) und *Hindsight Experience Replay* (HER), die Auswirkungen auf das Lernverhalten des Agenten haben. Eine weitere Technik, die in dieser Implementierung die Startzustände reguliert, nennt sich *Curriculum Learning*.

3.2.1 Deep Deterministic Policy Gradient

DDPG ist eine Actor-Critic modellfreier Algorithmus [2]. Modellfrei bedeutet, dass die Umgebung dem Agenten nicht bekannt ist und nur durch die Interaktion diese erkundet wird. Der Algorithmus besteht aus einem Actor und einem Critic. Der Critic bewertet mit Q-Werten die Zustand-Aktions-Paare. Dieses Verfahren baut auf dem bereits vorgestellten Deep Q-Learning auf, wobei Deep für den Einsatz von tiefen neuronalen Netzen steht. Der zusätzliche Actor übernimmt hier die Ermittlung einer Aktion. Diese Trennung von Critic und Actor ermöglicht einen kontinuierlichen Aktionsraum. Bei Critic-only Algorithmen (wie Q-Learning) sind nur diskrete Aktionsräume möglich. Bei diesen wird eine konkrete Anzahl von Zustand-Aktions-Paaren bewertet und je nach Strategie eine Aktion aufgrund der jeweiligen Q-Werte ausgewählt.

In Abbildung 3.1 ist die Actor-Critic Architektur graphisch gezeigt [1]. Der Actor wählt aufgrund seiner Strategie eine Aktion für den Zustand. Die Aufgabe des Critics ist es diese Aktion für den Zustand mit Hilfe einer Q-Funktion zu bewerten. Um die beiden Netze zu trainieren wird der TD-Fehler, der sich aus dem Vergleich der Q-Werte ergibt, verwendet.

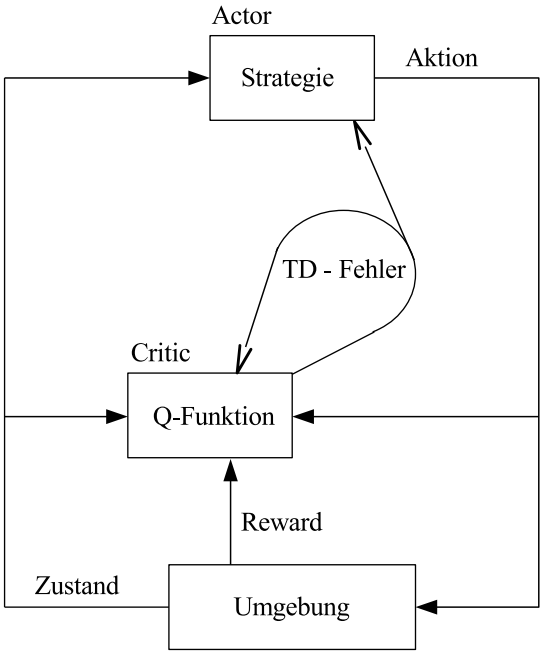


Abbildung 3.1: Actor-Critic Architektur

Der detaillierte Ablauf von DDPG ist in Algorithmus 1 beschrieben [2][4]. Das Critic-Netz Q und das Actor-Netz μ mit den Parameter θ^Q und θ^μ besitzen jeweils ein Target-Netz Q' und μ'. Der Experience Replay Speicher R ist ein FIFO-Buffer, der in jedem Schritt mit einer Transition gefüllt wird. Eine Transition ist ein Tupel mit den folgenden Elementen für den Zeitpunkt t: Zustand s_t, Aktion a_t, Reward r_t und Folgezustand s_{t+1}. Aus dem Speicher R werden die Transitionen für die Minibatches zum Trainieren der Netze entnommen. Die genaue Funktionsweise des Speichers wird in Abschnitt 3.2.5 erklärt.
Der Algorithmus durchläuft nach den benötigten Initialisierungen M Episoden, wobei jede Episode T Schritte hat. Zu Begin jeder Episode wird ein neuer Startzustand s_{Start} zufällig gesetzt (siehe Abschnitt 3.1.4). Das Rauschen (*noise*) $\mathcal{N}$ ermöglicht dem Agenten eine bessere Erkundung (*Exploration*). Im Abschnitt 3.2.4 werden verschiedene Varianten von Rauschen vorgestellt.

In der ersten For-Schleife jeder Episode wird diese komplett simuliert. Zu jedem Simulationsschritt t wird abhängig vom Zustand s_t und der Strategie π die Aktion a_t ermittelt. Durch Simulation der Aktionsausführung wird der Folgezustand s_{t+1} und die dazugehörige direkte Belohnung r_t erhalten und als Transition (s_t, a_t, r_t, s_{t+1}) in R gespeichert.

Algorithmus 1 DDPG-Algorithmus

Initialisiere zufällig das Critic-Netz $Q(s, a|\theta^Q)$ mit den Parametern θ^Q
Initialisiere zufällig das Actor-Netz $\mu(s|\theta^\mu)$ mit den Parametern θ^μ
Initialisiere die Target-Netze Q' und μ' mit den Parametern $\theta^{Q'} \leftarrow \theta^Q$ und $\theta^{\mu'} \leftarrow \theta^\mu$
Initialisiere den Experience Replay Speicher R
for $Episode = 1, M$ **do**
  Initialisiere ein Rauschen $\mathcal{N}$
  Setze einen Startzustand s_1
  for $t = 1, T$ **do**
    Wähle Aktion $a_t = \mu_\pi(s_t|\theta^\mu)$ gemäß der Strategie π und dem Rauschen $\mathcal{N}_t$
    Führe Aktion a_t aus und erhalte die Belohnung r_t und den Folgezustand s_{t+1}
    Speichere Transition (s_t, a_t, r_t, s_{t+1}) in R
  end for
  for $t = 1, T$ **do**
    Erstelle ein Minibatch von N Transitionen (s_i, a_i, r_i, s_{i+1}) aus R
    Setze das Target y_i:

$$y_i = r_i + \gamma {Q'}_\pi(s_{i+1}, {\mu'}_\pi(s_{i+1}|\theta^{\mu'})|\theta^{Q'}) \tag{3.8}$$

    Aktualisiere Critic durch Minimieren des Fehlbetrags L:

$$L = \frac{1}{N}\sum_i (y_i - Q_\pi(s_i, a_i|\theta^Q))^2 \tag{3.9}$$

    Aktualisiere den Actor durch Policy Gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q_\pi(s, a|\theta^Q)|s = s_i, a = \mu_\pi(s_i)\nabla_{\theta^\mu}\mu_\pi(s|\theta^\mu)|s_i \tag{3.10}$$

    Aktualisiere die Target-Netze:

$$\begin{aligned} \theta^{Q'} &\leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'} \\ \theta^{\mu'} &\leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'} \end{aligned} \tag{3.11}$$

  end for
end for

In der zweiten For-Schleife werden die Netze des Critics und des Actors trainiert. Für das Training des Critics wird der Q-Wert verwendet. Der Q-Wert ist, wie beim Q-Learning, die erwartete Gesamtbelohnung (Gesamtreward) für das Ausführen der Aktion a_t im Zustand s_t. Das Netz Q wird durch Gradient Descent trainiert. Der dabei zu minimierende

Fehlerbetrag (*loss*) L ist in Gleichung 3.9 zu sehen. Die hier verwendete Verlustfunktion ist *mean squared error* (MSE). Weitere Verlustfunktionen und die Architektur der Netze Q und Q' werden in Abschnitt 3.2.2 vorgestellt.
Das Target y_i ist der Gesamtreward, der sich aus dem direkten Reward und der Vorhersage des zukünftigen Gesamtrewards ergibt. Das Target, welches auch beim Q-Learning verwendet wird (siehe Abschnitt 2.2.5), unterscheidet sich von dem hier verwendeten Target. Zur Berechnung des Targets y_i (siehe Gleichung 3.8) werden die Target-Netze Q' und μ' verwendet. Die Aktion a_{t+1} für den Folgezustand s_{t+1} wird dabei durch das Target-Netz μ' gewählt. Bevor das Target y_i für das Training verwendet wird, findet ein Target-Clipping statt. Dies trägt dazu bei, dass das Target nur Werte annimmt, die im möglichen Wertebereich liegen. Somit wird ein berechnetes Target y durch die Funktion $target\_clipping(y)$ in den Wertebereich, der durch y_{min} und y_{max} festgelegt ist, geschnitten.

$$target\_clipping(y) = clip(y, y_{min}, y_{max}) \tag{3.12}$$

Die Funktion *clip* ist wie folgt definiert:

$$clip(x, a, b) = \begin{cases} a & falls\ x \leq a, \\ b & falls\ x \geq b, \\ x & sonst. \end{cases} \tag{3.13}$$

Zur Festlegung der Grenzwerte des Targets werden dessen minimale und maximale Werte ermittelt. Da das Target dem Gesamtreward entspricht, berechnet es sich nach Gleichung 2.15 mit dem Diskontierungsfaktor γ wie folgt:

$$\begin{aligned} y_{min} &= \frac{r_{min}}{1-\gamma} \\ y_{max} &= \frac{r_{max}}{1-\gamma} \end{aligned} \tag{3.14}$$

Die Werte r_{min} und r_{max} ergeben sich aus dem Wertebereich der Reward-Funktion mit $r(s,a) \in [r_{min}, r_{max}]$, welche in Abschnitt 3.2.3 definiert wird.
Der Actor wird durch Policy Gradient trainiert. Die Approximation für den Gradienten ist durch Gleichung 3.10 gegeben, wobei J die Startverteilung ist [2]. Der Gradient entsteht durch eine Verkettung der Gradienten der beiden Netze wie folgt [11]:

$$\nabla_{\theta^\mu} \mu_\pi(s) = \nabla_a Q_\pi(s, a|\theta^Q) \nabla_{\theta^\mu} \mu_\pi(s|\theta^\mu) \tag{3.15}$$

Nach dem Training der Netze werden die Target-Netze Q' und μ' langsam den Trainingsnetzen Q und μ angenähert. Dazu werden wie in den Zuweisungen 3.11 die jeweiligen Parameter angepasst. Der Wert τ mit $\tau \ll 1$ entscheidet darüber wie schnell diese Annäherung geschieht.

3.2.2 Netzarchitektur

Abhängig von der Problemstellung und der Definition von Aktions- und Zustandsraum kann eine Netzarchitektur für Actor und Critic gewählt werden. Die Definition der Netzarchitektur basiert jeweils auf der ersten Variante mit 15 Dimensionen für den Zustandsraum und vier Dimensionen für den Aktionsraum.

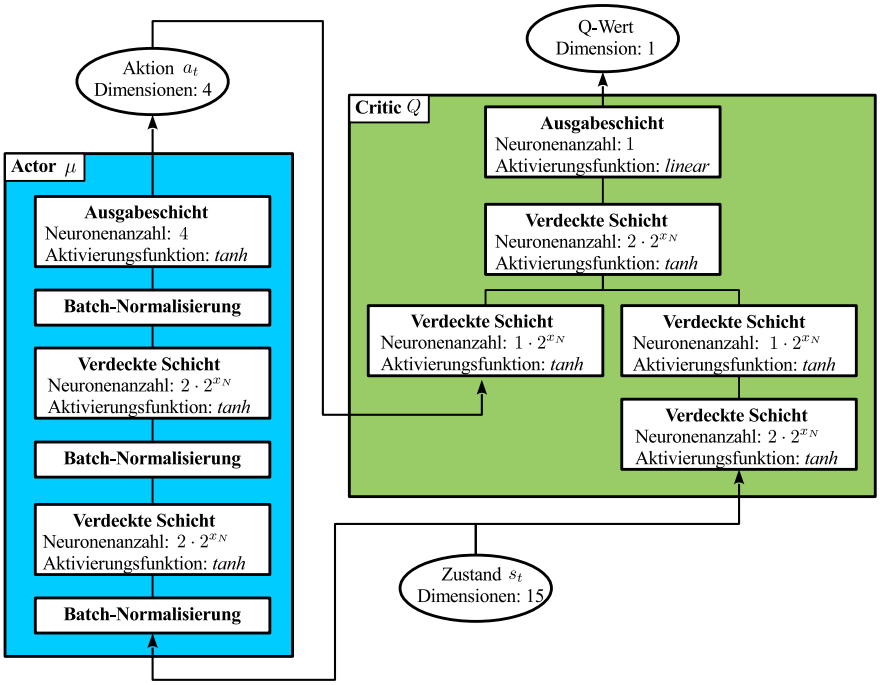


Abbildung 3.2: Netzarchitektur

Die Abbildung 3.2 zeigt die Netzarchitektur mit allen Schichten (*Layer*). Jede Schicht ist komplett mit seinen benachbarten Schichten verbunden (*Fully Connected Layer*). Das bedeutet, dass jedes Neuron einer Schicht jeweils eine Verbindung zu jedem Neuron in der vorherigen und folgenden Schicht besitzt. Die Anzahl an Neuronen ist variabel einstellbar. Dazu ist der Faktor $x_N \in \mathbb{N}$ gegeben. Eine geeignete Netzgröße mit 645 Neuronen ist durch $x_N = 6$ gewählt (siehe Abschnitt 4.2.2).

In Python sind die Schichten mit Hilfe der *Dense-Layer* aus der *Keras*-Bibliothek implementiert. *Keras* bietet eine vereinfachte Schnittstelle zum Erstellen von neuronalen Netzen und verwendet dabei selbst *Tensorflow* als Backend.
Die Ausgabeschicht für den Actor wird durch eine tanh-Funktion aktiviert. Somit sind die Werte für die Aktion im Bereich $a_t \in [-1, +1]$. Der Wertebereich des Q-Wertes ist abhängig von der Reward-Funktion. Um alle möglichen Werte aus dem reellen Zahlenraum abzudecken, wird die Ausgabeschicht des Critics mit einer linearen Funktion aktiviert. Für alle verdeckten Schichten hat sich die tanh-Funktion als geeignetste Aktivierungsfunktion herausgestellt.
Zur Normalisierung der Netzeingänge wird beim Actor vor jede Schicht eine zusätzliche Schicht mit Batch-Normalisierung eingesetzt. Dadurch wird der Wertebereich der Eingangssignale der folgenden Schichten normalisiert. Jedoch ist dies beim Critic nicht einsetzbar, da ansonsten das Training nicht konvergiert. Eine weitere Maßnahme, um die Gewichte der Netze in einen kleinen Bereich von $[-1, +1]$ zu bringen, wird durch die Initialisierung erreicht. Zur Initialisierung wird *glorot-uniform* eingesetzt. Diese verwendet eine Normalverteilung mit Mittelwert 0 und einer Varianz, die sich nach der Netzgröße richtet.
Zur Bestimmung einer Aktion a_t in einem Zustand s_t wird nur der Actor μ benötigt. Dieser legt den Zustand an sein Netz an und wählt durch Weiterleiten (*Forwarding*) eine Aktion. Der Critic kann anschließend durch Weiterleiten der gewählten Aktion und des gegebenen Zustandes eine Bewertung dieses Zustand-Aktions-Paares treffen. Dem Critic ist dabei eine zusätzliche Schicht nur mit dem Zustand als Eingabe gegeben, bevor die Aktionen und Zustände zusammen in eine Schicht gelangen [2]. Dies gibt dem Critic die Möglichkeit die Eigenschaften der Zustände unabhängig von den Aktionen zu trainieren.

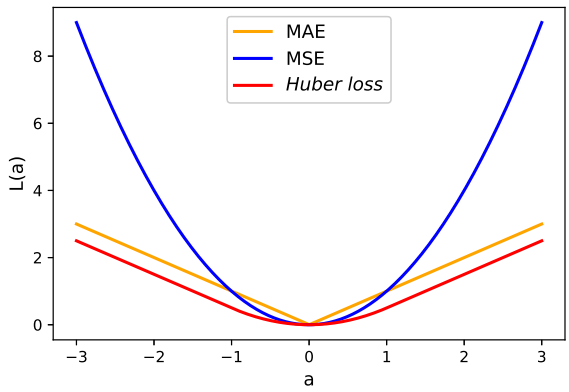


Abbildung 3.3: Verlustfunktionen

Die Trainingsverfahren für den Actor und den Critic sind in Abschnitt 3.2.1 vorgestellt. Der Gradient Descent Algorithmus des Critics ist in *Keras* durch den *Adam Optimizer* mit einer Lernrate von 0,001 implementiert. Im vorgestellten Algorithmus 1 wurde als Verlustfunktion MSE verwendet (Gleichung 3.17). Alternativen dazu sind MAE (Gleichung 3.16) oder *Huber loss* (Gleichung 3.18). *Huber loss* soll ein robusteres Lernen ermöglichen, indem es weniger auf Ausreißer reagiert [12]. Eine Verlustfunktion dient zur Bestimmung der Genauigkeit eines beobachteten Wertes y im Vergleich zu dessen vorhergesagten Wert $\hat{y}$. Die Funktionen sind wie folgt definiert:

$$L(y, \hat{y}) = |y - \hat{y}| \tag{3.16}$$

$$L(y, \hat{y}) = (y - \hat{y})^2 \tag{3.17}$$

$$L_{\delta_h}(y, \hat{y}) = \begin{cases} \frac{1}{2}(y - \hat{y})^2 & falls\ |y - \hat{y}| \leq \delta_h, \\ \delta_h(|y - \hat{y}| - \frac{1}{2}\delta_h) & sonst. \end{cases} \tag{3.18}$$

Für *Huber loss* gilt hier $\delta_h = 1$. Die Abbildung 3.3 zeigt die drei Funktionsgraphen im Vergleich, wobei a die Differenz von y und $\hat{y}$ mit $a = y - \hat{y}$ ist. Wie sich die drei Funktionen auf das Training des Agenten auswirken, wird in Abschnitt 4.2.2 evaluiert.

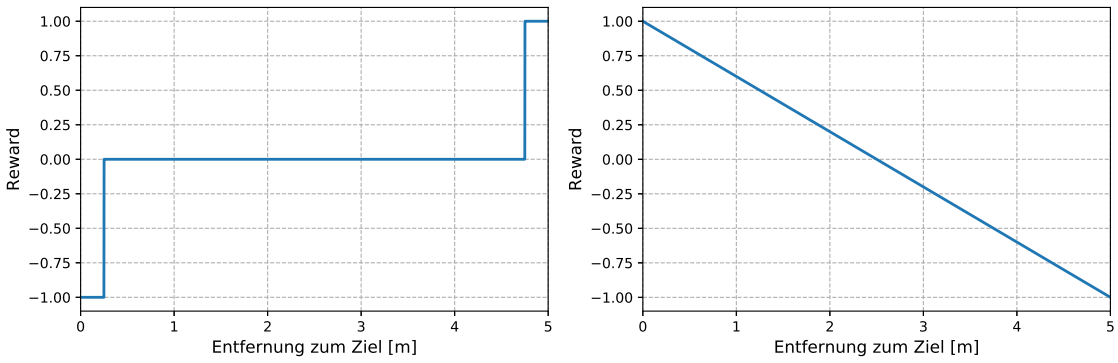
3.2.3 Reward-Funktion

Die Reward-Funktion ist einer der wichtigsten Funktionen des Algorithmus und entscheidet darüber, ob ein Training zielführend ist [13]. Eine ungeeignete Funktion kann den Agenten ein falsches Ziel lernen lassen. Die Reward-Funktion belohnt oder bestraft einen Agenten für das Betreten eines Zustandes. Somit weist die Reward-Funktion jedem Zustand eine direkte Belohnung (*immediate reward*) zu. Positive Zustände werden dementsprechend belohnt (*positiv reward*) und Fehler werden wiederum bestraft (*negativ reward*). Ein Fehler wäre hier das Verlassen des erlaubten Zustandsraumes.

Shaped und *sparse* Reward-Funktionen

In der Regel unterscheidet man zwischen *sparse* und *shaped* Reward-Funktionen. Eine *sparse* (spärliche) Reward-Funktion liefert bei Erreichen des Ziels einen positiven Reward (z.B. $+1$) und bei einem Fehler einen negativen Reward (z.B. -1). Alle weiteren Zustände werden neutral (Reward ist 0) behandelt. Je nach Problemstellung kann die

sparse Reward-Funktion angepasst werden. Bei episodischen Aufgaben, wo die Episode bei einem Fehler abgebrochen wird, wird jeder Zustand, der kein Fehler ist, belohnt (Reward ist z.B. +1). Da der Agent versucht die Gesamtbelohnung zu maximieren, wird er versuchen Fehler zu meiden. Bei kontinuierlichen Aufgaben sollen auch Fehler vermieden werden, jedoch wird bei einem Fehler die Episode nicht abgebrochen, aber bestraft (Reward ist z.B. −1). Alle positiven Zustände werden nicht bestraft (Reward ist z.B. 0). So wird der Agent auch versuchen Fehler zu meiden.
Eine *shaped* (geformte) Reward-Funktion liefert abhängig von der Variante ebenfalls negative Rewards für Fehler (Reward ist z.B. −1) und positive Rewards für das Erreichen des Ziel (Reward ist z.B. +1). Der Unterschied liegt in der Bewertung der restlichen Zustände. Diese werden nicht neutral bewertet, sondern der Reward richtet sich nach dem Zustandsabstand zum Ziel. Die Definition des Zustandsabstands ist je nach Problemstellung unterschiedlich. So kann bei einer Aufgabe, in der eine Position erreicht werden soll, die Entfernung zu der Zielposition als Maß für die Reward-Funktion genommen werden. Auch bei einer *shaped* Reward-Funktion gibt es beliebige Varianten, wie für episodische und kontinuierliche Aufgaben, die den Wertebereich der Reward-Funktion festlegen.



(a) *Sparse* Reward (b) *Shaped* Reward

Abbildung 3.4: Beispielaufgabe „Topfschlagen“: Vergleich *sparse* und *shaped* Reward

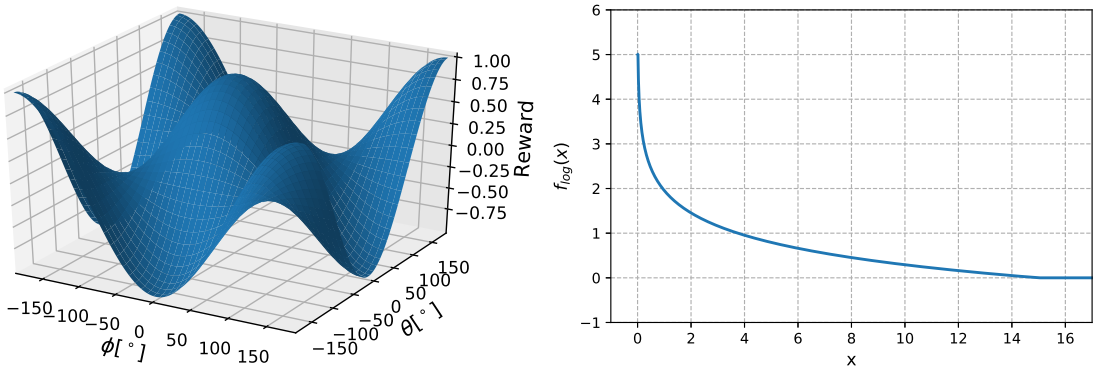
Sei als Beispielaufgabe das Kinderspiel „Topfschlagen“ gewählt. In diesem Spiel ist die Aufgabe mit verbundenen Augen einen Topf im Spielfeld zu finden. Die Reward-Funktion basiert dabei auf der Entfernung in Meter zum Ziel. Den direkten Reward liefern bei diesem Spiel die Zuschauer in Form von Zurufen für den Spieler. Bei einer *sparse* Reward-Funktion wie in Abbildung 3.4a erhält der Spieler so gut wie keine Rückmeldung von den Zuschauern. Lediglich bei Erreichen des Ziels oder bei Verlassen des Spielfelds erhält der Spieler die Zurufe „sehr warm“ und „sehr kalt“. Somit ist es für den Spieler sehr schwierig sich auf dem Spielfeld Richtung Ziel zu orientieren. Bei einer *shaped* Reward-Funktion

wie in Abbildung 3.4b erhält der Spieler durchgehend Zurufe wie „warm“ und „kalt“ und zusätzlich können die Zuschauer die Information mitgeben, ob der aktuelle Zustand „wärmer“ oder „kälter“ als der vorherige ist.

Der Vorteil einer *shaped* Reward-Funktion ist die Möglichkeit für den Agenten einfach die Rewardunterschiede zwischen zwei Zuständen zu erkennen und somit auch schneller das Ziel zu finden und das Training zu beschleunigen. Der Nachteil ist, dass für eine Definition der *shaped* Reward-Funktion die Umgebung bekannt sein muss. Des Weiteren muss das Wissen vorhanden sein, wie sich verschiedene Zustände im Verhältnis auf das Erreichen des Zieles auswirken. Aber auch, wenn dieses Wissen vorhanden ist, ist der Entwurf einer geeigneten Reward-Funktion nicht trivial. Dies ist zugleich der Vorteil der *sparse* Reward-Funktion. Bei dieser müssen lediglich positive wie negative Zielzustände definiert werden. Diese werden dann mit einem positiven bzw. negativen Reward versehen. Somit ist beim Entwurf kein Wissen über die Umgebung nötig, sonder nur was das gewünschte Ziel ist. Ein weiterer Vorteil durch diese einfache Definition ist, dass durch diese Reward-Funktion das Ziel und nicht der Weg zum Ziel definiert wird. So wird dem Agenten Freiraum zum Entwickeln eigener Lösungen gegeben.

Die Hilfsfunktionen f_{dir} und f_{log}

Das in Abschnitt 3.1 vorgestellte Problem wird als kontinuierliche Aufgabe behandelt, dessen Episoden nach Ablauf einer festgelegten Zeit enden. Für diese Aufgabenstellung ist das Verwenden von *shaped* und *sparse* Reward-Funktionen möglich.



(a) Funktion f_{dir} (siehe Gleichung 3.20)

(b) Funktion f_{log} (siehe Gleichung 3.21)

Abbildung 3.5: Hilfsfunktionen für die Reward-Funktion

In beiden Varianten wird der Agent bestraft, wenn der Quadcopter nach unten gerichtet ist. Der Wert von f_{dir} gibt die Information, ob der Quadcopter nach oben oder unten gerichtet ist. Zur Berechnung wird ein gedachter Punkt $p_z = \begin{pmatrix} 0 & 0 & 1 \end{pmatrix}^T$ mit der Rotationsmatrix R aus Gleichung 2.1 gedreht. Der Punkt p_z befindet sich auf der z'-Achse im Bezugssystems E_B des Quadcopters einen Meter über dessen Schwerpunkt. Die Definition ist wie folgt:

$$\begin{pmatrix} * \\ * \\ f_{dir} \end{pmatrix} = R \cdot p_z = \begin{pmatrix} C_\theta C_\psi & S_\phi S_\theta C_\psi - C_\phi S_\psi & C_\phi S_\theta C_\psi + S_\phi S_\psi \\ C_\theta S_\psi & S_\phi S_\theta S_\psi + C_\phi C_\psi & C_\phi S_\theta S_\psi - S_\phi C_\psi \\ -S_\theta & S_\phi C_\theta & C_\phi C_\theta \end{pmatrix} \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \tag{3.19}$$

In Gleichung 3.19 ist durch die *Don't-Cares* $(*)$ zu erkennen, dass f_{dir} nur von ϕ und θ abhängig ist und dadurch ist die Funktionsdefinition wie folgt gegeben:

$$f_{dir}(\phi, \theta) = \cos(\phi)\cos(\theta) \tag{3.20}$$

Für den Wertebereich gilt $f_{dir} \in [-1, +1]$. Dabei entsteht für f_{dir} der Wert $+1$, falls der Quadcopter direkt nach oben gerichtet ist, und -1, falls der Quadcopter nach unten gerichtet ist. Der Funktionsverlauf abhängig von ϕ und θ ist in Abbildung 3.5a dargestellt. ϕ und θ sind durch den Zustandsraum (siehe Abschnitt 3.1.3) auf den Wertebereich von $[-180°, +180°]$ begrenzt. Aufgrund der Verwendung von Euler-Winkeln kann eine Orientierung durch verschiedene Drehungen erzeugt werden. So ist zum Beispiel die nach oben gerichtete Orientierung durch fünf folgende Winkelstellungen repräsentiert:

1. $\phi = 0°, \theta = 0°$
2. $\phi = 180°, \theta = 180°$
3. $\phi = 180°, \theta = -180°$
4. $\phi = -180°, \theta = 180°$
5. $\phi = -180°, \theta = -180°$

Diese fünf Winkelstellungen sind in der Abbildung 3.5a durch die Funktionsspitzen mit dem Wert $+1$ zu erkennen. Allerdings ist aufgrund der Modulorechnung der Winkel ϕ und θ der Randbereich der Funktion überlappend und somit sind die Winkelstellungen 2-5 als eine Orientierung zu sehen.

Die Zustandselemente $\dot{x}$, $\dot{y}$ und $\dot{z}$ sind von besonderem Interesse für die Reward-Funktion, da das Ziel der Stillstand des Quadcopters ist. Dies bedeutet, dass die Geschwindigkeit und die Beschleunigung gegen 0 geht. Die Hilfsfunktion f_{log}, die Teil der *shaped* Reward-Funktion ist, ist nur von $\dot{x}$, $\dot{y}$ und $\dot{z}$ abhängig. Die Funktion liefert abhängig von der höchsten Geschwindigkeit einen Wert zwischen 0 und 5, der umso höher ist, je langsamer der Quadcopter ist. Die Hilfsfunktion f_{log} ist wie folgt definiert:

$$f_{log}(\dot{x}, \dot{y}, \dot{z}) = clip(\log_{\frac{1}{4}}(\frac{\max(|\dot{x}|, |\dot{y}|, |\dot{z}|)}{15}), 0, 5) \tag{3.21}$$

Die Funktion *clip* begrenzt die Funktionswerte $f_{log} \in [0, 5]$ und ist in Gleichung 3.13 definiert. Da es in der Python Bibliothek *numpy* keine Funktion $\log_{\frac{1}{4}}$ gibt, ist diese durch die vorhandene Funktion $\log_2$ wie folgt nach den Logarithmengesetzen ersetzt:

$$\log_{\frac{1}{4}}(x) = \frac{\log_2(x)}{\log_2(\frac{1}{4})} \tag{3.22}$$

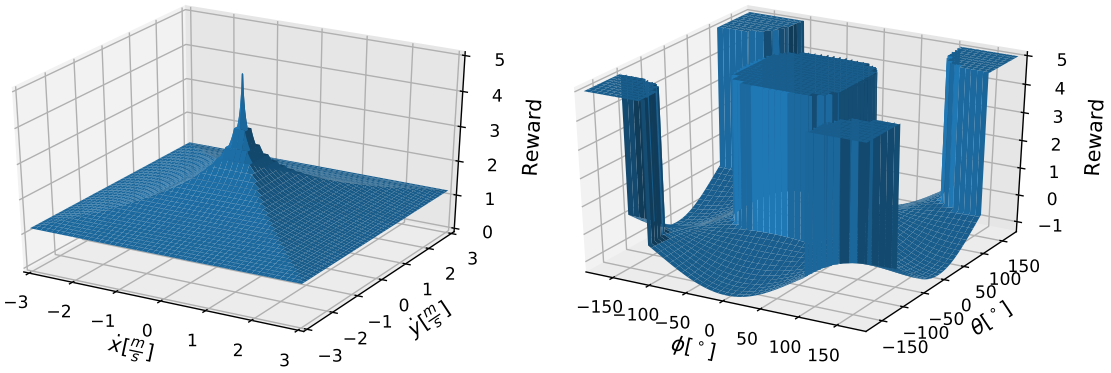
In Abbildung 3.5b ist die Hilfsfunktion f_{log} abgebildet, die in der *shaped* Reward-Funktion den positiven Reward abdeckt. Da nur der positive Reward durch diese Hilfsfunktion gegeben sein soll, ist ein Reward-Clipping nach unten nötig. Ein Clipping nach oben ist benötigt, da $\lim_{x \to 0} \log_{\frac{1}{4}}(x) = +\infty$, und zu hohe Rewards ein Training instabil machen.
Die Entscheidung nur den höchsten Geschwindigkeitswert in der Reward-Funktion zu berücksichtigen, basiert auf der Erfahrung, dass bei einem Mittelwert aus allen drei Geschwindigkeitswerten, das Trainingsergebnis schlechter ist. Der Faktor $\frac{1}{15}$ unterstützt, dass auch hohe Geschwindigkeiten bis zu $15\frac{m}{s}$ sich durch die Rewards vergleichen können. Andernfalls wäre durch das Reward-Clipping für alle hohen Geschwindigkeiten ein neutraler Reward von 0 gegeben.

Definition der Reward-Funktionen

Die Reward-Funktion, welche für diese Aufgabe verwendet wird, gibt es in einer *shaped* und *sparse* Variante. Beide Varianten sind von den Zustandselementen $\dot{x}$, $\dot{y}$, $\dot{z}$, ϕ und θ abhängig. Um die Reward-Funktionen in Funktionsgraphen zu veranschaulichen, ist zu jeder Variante ein Graph, der die Abhängigkeit von der Geschwindigkeit anzeigt und ein Graph, der die Abhängigkeit von den Winkeln anzeigt, gegeben. Der Graph für die Abhängigkeit von der Geschwindigkeit ist dabei auf $\dot{x}$ und $\dot{y}$ in der horizontalen Ebene beschränkt. Die restlichen Zustandselemente, von denen der Graph nicht abhängig ist, werden auf $0\frac{m}{s}$ bzw. $0°$ gesetzt.

Die *shaped* Reward-Funktion ist durch Gleichung 3.23 gegeben und in Abbildung 3.6 veranschaulicht. Diese Reward-Funktion gibt einen negativen Reward durch die Hilfsfunktion f_{dir}, falls der Quadcopter nach unten oder ganz leicht nach oben gerichtet ist. Dabei wird der negative Reward größer, je weiter der Quadcopter nach unten geneigt ist. Dies soll dabei unterstützen eine aufrechte Position zu erlangen. Dieser Verlauf der Funktion ist in den Tälern des Funktionsgraphen in Abbildung 3.6b zu erkennen. Sobald eine ausreichend aufrechte Position erreicht ist, gibt es durch die Hilfsfunktion f_{log} einen positiven Reward. In Abbildung 3.6b ist dieser maximal ($r = 5$), da für diese Abbildung die Geschwindigkeit in alle Richtungen auf $0\frac{m}{s}$ gesetzt wurde. In Abbildung 3.6a ist der Quadcopter aufrecht orientiert und der Reward ist nur von der Hilfsfunktion f_{log} abhängig.

$$r_{shaped}(\dot{x}, \dot{y}, \dot{z}, \phi, \theta) = \begin{cases} f_{dir}(\phi, \theta) - 0{,}25 & falls\ f_{dir}(\phi, \theta) \leq 0{,}25\ , \\ f_{log}(\dot{x}, \dot{y}, \dot{z}) & sonst. \end{cases} \tag{3.23}$$



(a) *Shaped* Reward $r_{shaped}(\dot{x}, \dot{y}, \dot{z}, \phi, \theta)$ mit $\dot{z} = 0\frac{m}{s}$, $\phi = 0°$ und $\theta = 0°$

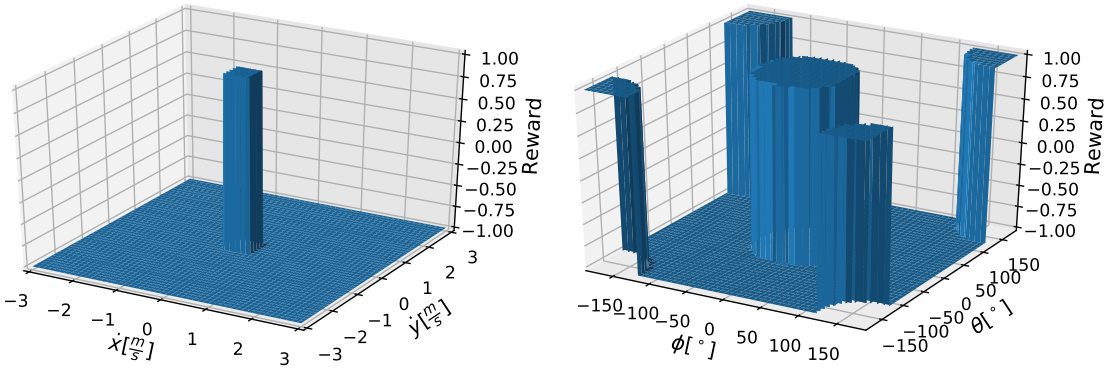
(b) *Shaped* Reward $r_{shaped}(\dot{x}, \dot{y}, \dot{z}, \phi, \theta)$ mit $\dot{x} = 0\frac{m}{s}$, $\dot{y} = 0\frac{m}{s}$ und $\dot{z} = 0\frac{m}{s}$

Abbildung 3.6: *Shaped* Reward

Die *sparse* Reward-Funktion ist durch Gleichung 3.24 gegeben und in Abbildung 3.7 veranschaulicht. In dieser Reward-Funktion gibt es nur einen positiven Reward von $+1$, falls der Quadcopter zu einem bestimmten Grad nach oben gerichtet ist und die Geschwindigkeit in alle Richtungen über eine bestimmte Schwelle nicht überschreitet. Ansonsten gibt es einen negativen Reward von -1. Die Geschwindigkeitsschwelle wird in Abbildung 3.7a gezeigt. Der gewünschte Richtungsgrad, welcher von den Neigungswinkel ϕ und θ

abhängig ist, ist in Abbildung 3.7b veranschaulicht.

$$r_{sparse}(\dot{x},\dot{y},\dot{z},\phi,\theta) = \begin{cases} +1 & falls\ f_{dir}(\phi,\theta) \geq 0{,}5 \wedge |\dot{x}| \leq 0{,}2 \wedge |\dot{y}| \leq 0{,}2 \wedge |\dot{z}| \leq 0{,}2\ , \\ -1 & sonst. \end{cases} \tag{3.24}$$



(a) *Sparse* Reward $r_{sparse}(\dot{x},\dot{y},\dot{z},\phi,\theta)$ mit $\dot{z} = 0\frac{m}{s}$, $\phi = 0°$ und $\theta = 0°$

(b) *Sparse* Reward $r_{sparse}(\dot{x},\dot{y},\dot{z},\phi,\theta)$ mit $\dot{x} = 0\frac{m}{s}$, $\dot{y} = 0\frac{m}{s}$ und $\dot{z} = 0\frac{m}{s}$

Abbildung 3.7: *Sparse* Reward

Durch die Definition für die Reward-Funktion r ergeben sich jeweils die Grenzwerte r_{min} und r_{max}, die für das Target-Clipping (siehe Abschnitt 3.1) benötigt werden. Es gilt nach den gegebenen Definitionen: $r_{shaped} \in [-1{,}25; 5]$ und $r_{sparse} \in \{-1, +1\}$.

3.2.4 Exploration und Exploitation

Die Erkundung (*Exploration*) der Umgebung ist ein wichtiger Punkt beim Training eines Agenten. Ist die Ausnutzung (*Exploration*) einer bestehenden Strategie zu groß, so besteht das Risiko, dass der Agent gegen ein lokales Optimum konvergiert und dadurch andere bessere Optima nicht kennen lernt. Deswegen ist es eine entscheide Aufgabe ein geeignetes Verhältnis zwischen *Exploration* und *Exploitation* zu finden.
Im Normalfall existiert zu Beginn des Trainings noch keine brauchbare Strategie für die Aktionsauswahl. Aus diesem Grund werden die Entscheidungen zufällig getroffen. Diese stochastische Strategie trägt zur *Exploration* bei, da die Umgebung erkundet wird. Aber auch im Laufe des Trainings kann *Exploration* eingesetzt werden, um neue Flugbahnen zu erkunden.

Im Gegensatz zur *Exploration* nutzt die *Exploitation* die bereits entwickelte Strategie aus. Dies wird dann als *greedy* Strategie bezeichnet. Eine solche Strategie wählt in jedem Schritt die Aktion, die die voraussichtlich höchste Gesamtbelohnung verspricht, aus.

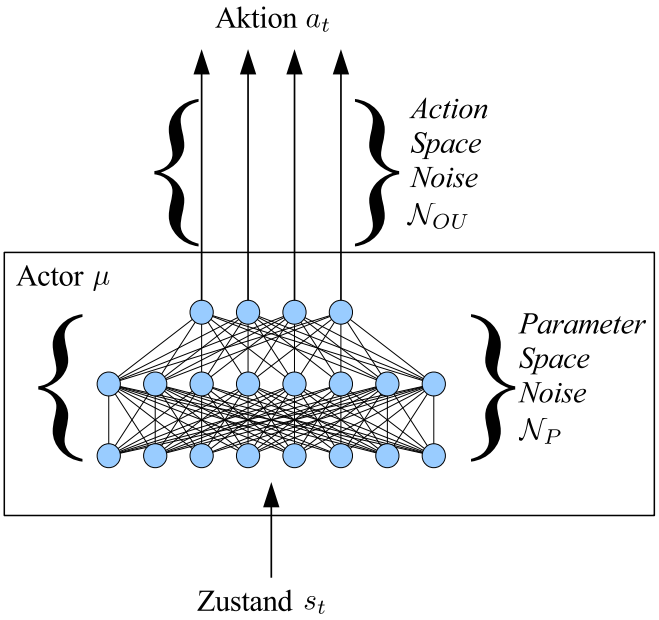


Abbildung 3.8: Rauschen: *Action Space Noise* und *Parameter Space Noise*

Um ein geeignetes Verhältnis zwischen *Exploration* und *Exploitation* sind verschiedene Techniken anwendbar. Durch Erzeugen eines Rauschen (*noise*) werden die ursprünglichen *greedy* Aktionswerte des Actors verändert. Je nach Stärke des Rauschens weichen die neuen Aktionen von der optimalen Strategie ab und erkunden die Umgebung (*Exploration*). Hier werden zwei Varianten von Rauschen verwendet, die auf unterschiedliche Weise erzeugt werden und auch an verschiedenen Stellen wirken. Während *Action Space Noise* den Ausgang des Actor-Netzes mit einem Rauschen belegt, manipuliert *Parameter Space Noise* die Gewichte des Actor-Netzes (siehe Abbildung 3.8). Beide Varianten werden in Abschnitt 4.2.1 miteinander verglichen. Eine Gemeinsamkeit ist die Verwendung der gaußschen Normalverteilung dessen Wahrscheinlichkeitsdichte mit

$$p(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.25}$$

berechnet wird und abhängig von dem Mittelwert μ und der Varianz σ^2 ist.

Action Space Noise

Action Space Noise ist ein Rauschen, welches direkt auf die Aktionswerte, die der Actor ausgibt, wirkt. Das Rauschen wird mit dem Ornstein-Uhlenbeck-Prozess [1], der aus der Stochastik stammt, erzeugt. Das Rauschen $\mathcal{N}_{OU}$ für eine Aktion a_t ist dementsprechend wie folgt definiert:

$$\mathcal{N}_{OU} = \theta_{OU} \cdot (\mu_{OU} - a_t) + \sigma_{OU} \cdot g(\mu_G, \sigma_G^2) \tag{3.26}$$

Dabei sind $\mathcal{N}_{OU}$, μ_{OU} und a_t Vektoren, deren Dimensionsgröße von der Anzahl an Aktionselementen in a_t abhängt. Dies sind je nach Variante des Aktionsraumes (siehe Abschnitt 3.1.2) vier oder fünf Dimensionen. Die Funktion $g(\mu_G, \sigma_G^2)$ liefert einen Vektor mit Zufallsvariablen, die durch die gaußsche Normalverteilung mit einem Mittelwert $\mu_G = 0$ und einer Varianz $\sigma_G^2 = 1$, gegeben sind. Die Festlegung des Wertes μ_{OU} ist abhängig von dem Mittelwert der gewünschten Aktionswerte in a_t. Da ein geeigneter Basiswert für die Rotorgeschwindigkeit in Abschnitt 3.1.2 durch b_ω festgelegt ist, ist μ_{OU} ein Nullvektor. θ_{OU} ist ein Parameter, der bestimmt wie stark die Aktion Richtung gewünschten Mittelwert μ_{OU} neigt. Der Parameter σ_{OU} bestimmt wie stark das Rauschen, welches durch die Normalverteilung bedingt ist, wirkt. Hier ist $\theta_{OU} = 0{,}75$ und $\sigma_{OU} = 0{,}25$ gesetzt.

Nach Erzeugen des Rauschens $\mathcal{N}_{OU}$ wird dieses direkt mit der Ausgabe des Actors addiert. Die Ausgabe des Actors μ mit den Netzparameter θ^μ ist in Abhängigkeit des Zustandes s_t und nach der Strategie π die Aktion mit $a_t = \mu_\pi(s_t|\theta^\mu)$. Die Ausgabe wird durch Vektoraddition mit dem Rauschen behaftet. Jedoch werden die Aktionswerte nicht in jedem Schritt mit einem Rauschen behaftet. Die Entscheidung, ob ein Rauschen erzeugt wird, geschieht durch eine $\epsilon-$*greedy* Strategie. Mit Hilfe der Explorationsrate $\epsilon \in [-1, +1]$ entscheidet die Strategie, ob eine *Exploration* oder *Exploitation* stattfindet. Die ϵ-greedy-Strategie ist zu ϵ eine zufällige Strategie und zu $(1-\epsilon)$ eine *greedy* Strategie. Eine ϵ-Reduktion sorgt dafür, dass der Wert von ϵ im Laufe des Trainings sinkt.

$$\epsilon_{t+1} = \epsilon_t - (\frac{\epsilon_{Start} - \epsilon_{Ende}}{N}) \tag{3.27}$$

Die Gleichung 3.27 zeigt die Abhängigkeit von dem aktuellen Zeitpunkt t für ϵ, wobei ϵ_{Start} und ϵ_{Ende}, die gesetzten Start- und Endwerte für ϵ sind. Der Startwert von ϵ wird vor dem Training gesetzt: $\epsilon_{t=0} = \epsilon_{Start}$. N ist die Anzahl an Schritten während des

[1] geht zurück auf das Paper von G. E. Uhlenbeck, L. S. Ornstein: *On the theory of Brownian Motion.(1930)*

gesamten Trainings und ergibt sich aus $N = T \cdot M$ mit T, der Anzahl an Schritten pro Episode, und M, der Anzahl an Episoden pro Training.

$$a_t = \begin{cases} \mu_\pi(s_t|\theta^\mu) + \mathcal{N}_{OU} & falls\ \epsilon_t \leq random_1(0,1), \\ \mu_\pi(s_t|\theta^\mu) & sonst. \end{cases} \tag{3.28}$$

Die Gleichung 3.28 zeigt, wie sich die $\epsilon-greedy$ Strategie auf die Aktion a_t auswirkt. Die Funktion $random_1(a, b)$ liefert eine Zufallszahl aus den gegebenen Grenzwerten a und b. Es gilt mit $\epsilon_{Start} = 0{,}99$, dass zu Trainingsbeginn stark explorativ vorgegangen wird. Weiterhin gilt mit $\epsilon_{Ende} = 0{,}05$, dass zum Ende des Trainings zwar eher die bestehende Strategie ausgenutzt wird (*greedy*), aber dennoch durch gelegentliche Erkundung der Umgebung eine Versteifung der Strategie verhindert werden soll.

Parameter Space Noise

Im Vergleich zum *Action Space Noise* wird beim *Parameter Space Noise* [14] die Aktion a_t, die der Actor μ ausgibt, nicht mit einem Rauschen behaftet. Stattdessen werden die Netzparameter θ^μ des Actors μ mit einem Rauschen $\mathcal{N}_P$ versetzt. Die dadurch gestörten Netzparameter $\tilde{\theta}^\mu$ (siehe Gleichung 3.29) liefern eine Aktion a_t in Abhängigkeit von der aktuellen Strategie π und dem aktuellen Zustand s_t wie in Gleichung 3.30 gezeigt.

$$\tilde{\theta}^\mu = \theta^\mu + \mathcal{N}_P \tag{3.29}$$

$$a_t = \mu_\pi(s_t|\tilde{\theta}^\mu) \tag{3.30}$$

Das Rauschen $\mathcal{N}_P$, basiert auf der gaußschen Normalverteilung mit dem Mittelwert μ_G und der Varianz σ_G^2 (siehe Gleichung 3.31). $\mathcal{N}_P$ ist ein mehrdimensionaler Vektor, dessen Dimensionsgröße von der Anzahl an Netzparametern abhängt.

$$\mathcal{N}_P = g(\mu_G, \sigma_G^2) \tag{3.31}$$

Der Mittelwert ist mit $\mu_G = 0$ gesetzt. Einen geeigneten Wert für σ_G zu ermitteln gestaltet sich deutlich schwieriger. Dieser Wert ist ausschlaggebend für die Stärke des Rauschens und sollte dem Netz angepasst werden. Ein weiteres Problem bei der Skalierung von σ_G ist, dass sich die Netze während des Trainings verändern können. Die Lösung ist eine adaptive Skalierungsmethode für σ_G. Diese wird zum Ende jeder Episode aufgerufen

und berechnet das $\sigma_{G_{k+1}}$ für die nächste Episode wie folgt:

$$\sigma_{G_{k+1}} = \begin{cases} \alpha_P \cdot \sigma_{G_k} & falls\ d(\theta^\mu, \tilde{\theta}^\mu) \leq \delta_P \\ \frac{1}{\alpha_P} \cdot \sigma_{G_k} & sonst. \end{cases} \tag{3.32}$$

Der Faktor $\alpha_P = 1{,}01$ dient zum Erhöhen und Senken des Wertes σ_G. Falls der Distanzmesswert d einen bestimmten Schwellwert $\delta_P = 0{,}75$ nicht überschreitet, wird σ_G erhöht und andernfalls gesenkt. Zu Beginn des Trainings wird $\sigma_{G_{k=0}} = 0{,}25$ gesetzt.
Der Distanzmesswert d ist ein Maß dafür wie sehr die Aktionswerte eines Actors μ mit den Netzparameter θ^μ von den gestörten Netzparametern $\tilde{\theta}^\mu$ abweichen. Die Definition von dem Distanzmesswert d lautet wie folgt:

$$d(\theta^\mu, \tilde{\theta}^\mu) = \sqrt{\frac{1}{N_a} \sum_{i=1}^{N_a} \mathbb{E}_s \left[(\mu_\pi(s|\theta^\mu)_i - \mu_\pi(s|\tilde{\theta}^\mu)_i)^2 \right]} \tag{3.33}$$

Zur Bestimmung des Distanzmesswertes d wird ein Minibatch von N Transitionen aus dem Experience Replay Speicher R zufällig entnommen. Aus diesem Minibatch bildet sich dann der Erwartungswert $\mathbb{E}_s$ für den Fehler der beiden Aktionswerte in Abhängigkeit von den Zuständen s. N_a ist die Dimensionsgröße des Aktionsraumes.

Diese Skalierungsmethode behebt nicht nur das Problem der Skalierung des Wertes, sondern unterstützt gleichzeitig ein geeignetes Verhältnis zwischen *Exploration* und *Exploitation* zu halten. Die Distanzberechnung d bietet dabei einen Anhaltspunkt wie hoch die *Exploration* ist. Falls diese zu hoch steigt, kann mit dieser Methode das Rauschen verringert werden und die Strategie geht wieder mehr Richtung *Exploitation*.

3.2.5 Experience Replay Speicher

Experience Replay Speicher ist ein FIFO-Buffer, der vergangene Erfahrungen speichert [15]. Die Erfahrungen sind Transitionen (s_t, a_t, r_t, s_{t+1}) mit denen der Agent trainiert wird. Ohne diesen Speicher würde in jedem Trainingsschritt immer die aktuelle Transition zum Training verwendet werden und anschließend verworfen werden. Mit dem Speicher kann beim Training eine beliebige Transition geladen werden. Allerdings wird nicht nur eine Transition zum Trainieren der Netze verwendet, sondern mehrere Transitionen werden zufällig ausgewählt und zu einem Minibatch zusammengefügt. Die Anzahl von

Transitionen für das Netztraining wird Minibatchgröße genannt und ist beliebig wählbar. Hier wird eine Minibatchgröße von 128 Transitionen pro Training verwendet. Durch das zufällige Auswählen der Transitionen geht die Korrelation zwischen den Transitionen verloren, was zu einem stabileren Training führt.
In jedem Simulationsschritt wird der Speicher R mit einer Transition (s_t, a_t, r_t, s_{t+1}) gefüllt (siehe Algorithmus 1). Die Speichergröße kann dabei beliebig festgelegt werden. Bei einer maximalen Größe können alle während des Trainings erzeugten Transitionen abgespeichert werden und somit wird keine Transition verworfen. Bei einer kleineren Speichergröße verhält sich der Speicher wie ein FIFO-Buffer und überschreibt die ältesten Transitionen, falls der Speicher voll ist. Drei Varianten von Speichergrößen werden in Abschnitt 4.2.4 miteinander verglichen.

Aufwärmphase

Die Aufwärmphase ist eine Phase vor dem Training, in der bereits Flugdaten gesammelt werden. Das bedeutet, dass in jedem Simulationsschritt eine Transitionen erzeugt wird und in dem Speicher abgelegt wird. Jedoch findet in dieser Phase kein Training statt. Für die Flugbahnen, die durch diese Phase entstehen, folgt der Agent in dieser Phase keiner Strategie und jede Aktion ist zufällig gewählt. Für die Aktionswahl ist keine der in Abschnitt 3.2.4 vorgestellten Rauschen anwendbar, da keine Strategie besteht. Stattdessen werden für die Aktionswerte zufällige Werte aus dem erlaubten Aktionsraum verwendet. Dies hat zur Folge, dass aufgrund einer fehlenden Strategie kaum stabile Flugzustände erreicht werden, was wiederum zur *Exploration* beiträgt.
Die gesammelten Flugdaten der Aufwärmphase werden vor der Trainingssimulation in den Experience Replay Speicher geladen. Voraussetzung dafür ist, dass der Speicher ausreichend groß ist, um alle Transitionen der Aufwärmphase abzuspeichern.
Ein weiterer Vorteil ergibt sich dadurch, dass zu Beginn der Trainingssimulation bereits Transitionen zum Trainieren zur Verfügung stehen und somit viel schneller erste Lernerfolge erzielt werden können.
Die Länge der Aufwärmphase entscheidet darüber, wie viele Transitionen vorweg gesammelt werden. So kann es auch sein, dass durch die Aufwärmphase alleine mehr Flugdaten gesammelt werden, als durch das gesamte Training. Die Episodenlänge beträgt während der Aufwärmphase nur $3s$ statt $10s$, da ohne Strategie nach kurzer Zeit Flugbereiche erreicht werden, die für das Training irrelevant sind. Bei einer Aufwärmphase von 2000 Episoden entstehen so 120.000 Transitionen.

Ein Nachteil der Flugdaten aus der Aufwärmphase ist, dass diese aufgrund der zufälligen Strategie, eventuell zu wenig Transitionen aus dem Zielbereich enthalten. So wird zwar schnell die grobe Richtung zum Ziel erlernt, aber es gibt zu wenig Flugdaten, die das Training in Zielnähe ermöglichen. Daher ist es wichtig eine passende Länge für die Aufwärmphase zu wählen, um ein geeignetes Verhältnis zu den Transitionen aus der Trainingsphase zu gewähren. Ein Vergleich verschiedener Längen der Aufwärmphasen befindet sich in Abschnitt 4.2.4.

Prioritized Experience Replay

Statt die Transitionen zufällig aus dem Speicher zu wählen, können Transitionen, die vielversprechend sind, bevorzugt ausgewählt werden. Dieses Vorgehen wird *Prioritized Experience Replay* (PER) genannt [3]. Vielversprechend bedeutet, dass diese Transition dem Training mehr Informationen beiträgt als eine andere Transition. Daher wird eine Transition aus einem bisher unbekannten Zustand bevorzugt, da diese das Netz entsprechend anpasst. Umgekehrt wird eine Transition aus einem oft besuchtem Zustandsgebiet, welche keine neuen Information beisteuert, vernachlässigt.

Um Festzustellen, ob eine Transition nützlich ist, wird beim Training jeder Transition dessen Fehler e notiert. Der Fehler ist die Differenz zwischen dem vorhergesagten Q-Wert des Critics Q und dem Target y_i (siehe Gleichung 3.8) für die jeweilige Transition. Somit ist der Fehler wie folgt definiert:

$$e = |Q_\pi(s_i, a_i) - y_i| \tag{3.34}$$

Bei einem hohen Fehler ist die Notwendigkeit ebenfalls hoch, dass das Netz an dieser Stelle optimiert werden muss. Daher wird jeder Transition i eine Priorität p_i zugewiesen und mit in den Speicher geschrieben. Dieser Prioritätswert bestimmt welche Transitionen bevorzugt in den Minibatch für das Training gewählt werden.

$$p_i = (e + \epsilon_{PER})^{\alpha_{PER}} \tag{3.35}$$

Die Gleichung 3.35 definiert den Prioritätswert einer Transition basierend auf dem Fehler e. Die Konstate $\epsilon_{PER} = 0{,}1$ stellt sicher, dass keine Transition eine Priorität von 0 hat. Der Parameter α_{PER} regelt wie hoch der Unterschied zwischen hoch und niedrig priorisierten Transitionen ist. Somit bedeutet $\alpha_{PER} = 0$, dass alle Transitionen die gleiche Priorität erhalten und keine Transition bevorzugt wird. Hier ist $\alpha_{PER} = 0{,}7$ gesetzt, um

im Minibatch ein geeignetes Verhältnis von priorisierten und nicht priorisierten Transitionen herzustellen.
Da der Fehler jeder Transition nur berechnet wird, wenn sich diese im Minibatch befindet, muss es einen initialen Prioritätswert für jede Transition geben. Eine Möglichkeit ist es den Initialwert mit dem Betrag des Rewards $|r|$ zu setzen. Allerdings werden dadurch einige Transitionen eventuell niemals in den Minibatch gewählt. Eine andere Möglichkeit ist den Initialwert einer Transition sehr hoch zu setzen, sodass diese auf jeden Fall einmal trainiert wird und dann den durch Gleichung 3.35 definierten Prioritätswert zugewiesen bekommt. Der Initialwert jeder neuen Transition wird hier auf $p = 1$ gesetzt.
Somit hat hat jede Transition i die Wahrscheinlichkeit

$$P_i = \frac{p_i}{\sum_{k=1}^{K} p_k} \tag{3.36}$$

in den Minibatch gewählt zu werden, wobei K die Menge von allen Transitionen im Experience Replay Speicher ist.

Um effektiv Minibatches abhängig von der Priorität der Transitionen zu erstellen, werden die Transitionen in einem Binärbaum gespeichert[1]. Dies ermöglicht das Erstellen von Minibatches in $O(\log(n))$. Der Binärbaum ist vollständig, was bedeutet, dass jedes Blatt die gleiche Tiefe hat. Jeder Knoten kennt sein Elter und seine Kinder. Zudem hat jeder Knoten eine Priorität. In den Blättern werden die Transitionen gespeichert. Die Priorität der Blätter entspricht der Priorität der enthaltenen Transition. Befindet sich keine Transition in einem Blatt, so ist dessen Priorität 0. Alle anderen Knoten haben als Priorität die addierten Prioritäten ihrer Kinder. Somit hat die Wurzel als Priorität die Gesamtpriorität von allen Transitionen. Ein Beispiel mit vier Transitionen ist in Abbildung 3.9 gezeigt.
Die Blätter sind von links nach rechts beginnend mit 0 durchnummeriert. Wird eine Transition in den Baum eingefügt, so gelangt sie in ein freies Blatt, mit der niedrigsten Blattnummer. Wenn jedes Blatt mit einer Transition belegt ist, wird nach dem FIFO-Prinzip die älteste Transition gelöscht, um Platz für eine neue zu machen. Die Größe des Baumes richtet sich nach der benötigten Größe an Speicher:

$$Baumhöhe = \lceil \log_2(benötigteSpeichergröße) \rceil + 1 \tag{3.37}$$

Dadurch ergibt sich eine Blätteranzahl von $2^{Baumhöhe-1}$.

[1] https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/ (siehe CD)

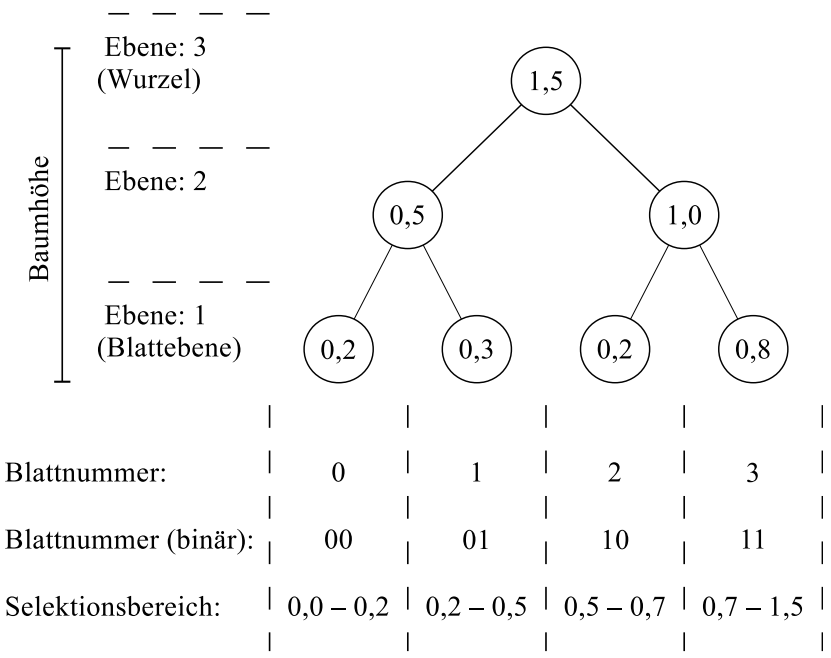


Abbildung 3.9: Beispiel für einen Binärbaum mit vier Transitionen

Um eine neue Transition mit seiner Priorität in den Baum einzufügen, wird die Funktion aus Algorithmus 2 mit den folgenden Parametern aufgerufen: *transition_einfügen*(Wurzel, Blattnummer, Baumhöhe-1, Priorität, Transition). Die Blattnummer ist dabei als binäre Zahl in eine Liste der Länge Baumhöhe-1 eingetragen. Die Funktion ruft sich selbst rekursiv auf, bis sie die Tiefe 1, welches die Blattebene ist, erreicht hat. Dort wird dann die Priorität angepasst und die neue Transition eingefügt. Die Differenz zwischen der alten und der neuen Priorität wird durch das rekursive Verfahren an jeden beteiligten Elterknoten zurückgereicht, damit diese ebenfalls ihre Priorität aktualisieren.

Zum Erstellen eines Minibatches von N Transitionen wird aus Gleichung 3 die Funktion *erstelle_minibatch*(N) aufgerufen. Diese definiert einen Selektor, der die Transitionen bestimmt. Jedes Blatt ist einem bestimmten Selektionsbereich zugewiesen (siehe Abbildung 3.9). Der gesamte Selektionsbereich geht von 0 bis zur Gesamtpriorität, wobei der Selektionsbereich eines Blattes durch dessen Priorität gegeben ist. Die N Transitionen werden dann einzeln durch die Funktion *erhalte_knoten* (siehe Algorithmus 4) aufgrund der Selektorwerte aus dem Baum abgerufen. Die Funktion läuft rekursiv baumabwärts und liefert den Blattknoten zurück. Der Selektor bestimmt dabei den Pfad zur Blattebene.

Alle Blätter, die eine Transition in das Minibatch eingetragen haben, werden in einer Liste temporär gespeichert. Nachdem das Training der Transitionen abgeschlossen ist, werden anhand der jeweiligen Fehler die neuen Prioritäten für die Transitionen berechnet. Anschließend wird jedes Blatt aus der Liste aktualisiert. Da jedes Blatt sein Elter kennt, kann die neue Priorität baumaufwärts weitergereicht werden. Jeder Knoten auf dem Pfad aktualisiert seine Priorität aufgrund der Differenz zwischen der alten und neuen Blattpriorität. Die Laufzeit für die Aktualisierung beträgt ebenfalls $O(\log(n))$.

Algorithmus 2 Einfügen einer Transition in den Binärbaum

```
Funktion transition_einfügen(Knoten, Blattnummer, Tiefe, Priorität, Transition):

if Tiefe == 1 then
  Prioritätsdifferenz = Priorität - Knoten.Priorität
  Knoten.Priorität = Priorität
  Knoten.Transition = Transition
  return Prioritätsdifferenz
else
  if Blattnummer[Baumhöhe - Tiefe] == 1 then
    Prioritätsdifferenz = transition_einfügen(Knoten.rechtes_Kind, Blattnummer,
    Tiefe-1, Priorität, Transition)
  else
    Prioritätsdifferenz = transition_einfügen(Knoten.linkes_Kind, Blattnummer,
    Tiefe-1, Priorität, Transition)
  end if
  Knoten.Priorität += Prioritätsdifferenz
  return Prioritätsdifferenz
end if
```

Algorithmus 3 Erstellen eines Minibatches

```
Funktion erstelle_minibatch(Minibatchgröße):

Gesamtpriorität = Wurzel.Priorität
Selektorbreite = Gesamtpriorität / Minibatchgröße
for i = 1, Minibatchgröße do
  Selektor = Selektorbreite · random_1(0, 1) + i
  Knoten = erhalte_knoten(Wurzel, Selektor, Baumhöhe-1)
  Minibatch[i] = Knoten.Transition
end for
return Minibatch
```

Algorithmus 4 Abrufen eines Knotens mit Hilfe eines Selektors

```
Funktion erhalte_knoten(Knoten, Selektor, Tiefe):

if Tiefe == 1 then
  return Knoten
else
  if Knoten.linkes_Kind.Priorität ≤ Selektor then
    return erhalte_knoten(Knoten.linkes_Kind, Selektor, Tiefe-1)
  else
    return erhalte_knoten(Knoten.rechtes_Kind, Selektor, Tiefe-1)
  end if
end if
```

Hindsight Experience Replay

Hindsight Experience Replay (HER) ist eine Technik mit der vor allem *sparse* Reward-Funktionen schneller erlernt werden sollen [4]. In einer Umgebung mit *sparse* Rewards besteht die Gefahr, dass der Agent nie einen Zielzustand erreicht und daher keine Strategie entwickeln kann. HER löst dieses Problem, indem es zu jeder Transition einer Episode ein Ziel $g \in \mathcal{G} \subseteq \mathcal{S}$ hinzufügt. Dieses Ziel ist jeweils der gewünschte Zielzustand. Zusätzlich werden die gleichen Transitionen mehrfach mit anderen Zielen in den Experience Replay Speicher geschrieben. Ein weiteres Ziel, welches definiert wird, kann der tatsächlich erreichte Endzustand einer Episode sein.
Ein Vergleich ist ein Basketballspieler, der lernt wie man einen Korb wirft. Angenommen er hat als Rückmeldung nur die Information, ob er getroffen hat, dann ist das Training sehr schwierig für ihn. Ein zusätzliches Ziel wäre der Landepunkt des Balles oder andere Punkte auf der Flugbahn. Somit hätte der Spieler die Rückmeldung: „Du hast zwar nicht getroffen, aber hättest du versucht dahin zu werfen, dann hättest du dort getroffen.“ Diese Technik ist mit jedem *off-policy* RL-Algorithmus verwendbar. Um HER mit dem vorgestellten DDPG-Algorithmus (siehe Algorithmus 1) zu verwenden, muss dieser in ein paar Punkten modifiziert werden. Der Algorithmus 5 zeigt den modifizierten DDPG-Algorithmus. Dieser soll nur den veränderten Ablauf durch die HER Technik darstellen. Daher werden einige Methoden und das detaillierten Trainings nicht explizit erwähnt. Aber dennoch werden diese bei HER analog zum DDPG-Algorithmus angewendet.

Zu Beginn jeder Episode wird ein Ziel g definiert. Dieses beschreibt den gewünschten Zielzustand und ist wie ein Zustand definiert. Die Netze, die zuvor s_t als Eingabe hatten, erhalten bei HER $s_t||g$. Durch den Operator $||$ werden der Zustand s_t und der Zielzustand

g konkateniert. Somit haben die Netze eine höher dimensionierte Eingabeschicht. Um die Dimension zu reduzieren, werden von dem Zielzustand g nur die Elemente $[\dot{x}, \dot{y}, \dot{z}, \phi, \theta]$, welche für die Berechnung des Rewards benötigt werden, an s_t konkateniert.
Im HER-Algorithmus wird jede Episode dreimal vollständig durchlaufen. Im ersten Durchlauf wird die Episode simuliert. Dabei werden sich für jeden Episodenschritt t jeweils der Zustand s_t, die Aktion a_t und der Folgezustand s_{t+1} gemerkt. Der zweite Durchlauf beinhaltet die Hauptaufgabe von HER. Für jeden Simulationsschritt wird ein Reward mit einer Reward-Funktion berechnet, die als zusätzlichen Parameter den Zielzustand g hat. Dadurch ist der Reward immer abhängig von dem gegebenen Ziel. Diese Reward-Funktion kann ebenso mit einer *shaped*, wie mit einer *sparse* Reward-Funktion, implementiert werden. Die Transitionen werden dann in den Speicher geschrieben, wobei an den Zustand s_t und den Folgezustand s_{t+1} der Zielzustand g konkateniert wird. Da HER mit mehreren Zielen arbeitet, wird eine Liste mit weiteren Zielen $G \subseteq \mathcal{G}$ erstellt. Für jedes dieser weiteren Ziele wird ebenfalls ein Reward berechnet und die Transition gespeichert. Im dritten Durchlauf werden, wie in Abschnitt 3.2.1 beschrieben, die Netze mit Minibatchen trainiert. Auch hier erhalten die Netze als Eingabe den konkatenierten Zielzustand. Insgesamt wird pro Episode das Training T mal aufgerufen, wobei T die Anzahl an Schritten pro Episode ist.

Algorithmus 5 Hindsight Experience Replay

Initialisiere Critic Q, Actor μ und Speicher R
for $Episode = 1, M$ **do**
  Setze einen Startzustand s_1 und ein Ziel g
  for $t = 1, T$ **do**
    Führe Aktion $a_t = \mu_\pi(s_t||g|\theta^\mu)$ aus und erhalte den Folgezustand s_{t+1}
  end for
  for $t = 1, T$ **do**
    $r_t = r(s_t, a_t, g)$
    Speichere Transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in R
    Erstelle weitere Ziele $G \subseteq \mathcal{G}$
    for $g' \in G$ **do**
      $r_t = r(s_t, a_t, g)$
      Speichere Transition $(s_t||g', a_t, r_t, s_{t+1}||g')$ in R
    end for
    for $t = 1, T$ **do**
      Trainiere Q und μ mit einem Minibatch aus R
    end for
  end for
end for

Zum Erstellen weiterer Ziele, die in die Transitionen geschrieben werden, sind vier Methoden gegeben [4]. Bei jeder Methode werden k Zustände aus einer Episode als ein Ziel g' definiert und in G hinzugefügt. Jede Transition wird somit k weitere Male mit anderen Zielen in den Speicher geschrieben. Bei der methodischen Auswahl weiterer Ziele für eine Transition ist dabei bekannt, in welcher Episode sie sich befindet und welche Transitionen zu dieser Episode gehören.

- *final*: Wähle den letzten Zustand der aktuellen Episode.
- *episode*: Wähle zufällig k beliebige Zustände aus der aktuellen Episode.
- *future*: Wähle zufällig k beliebige Zustände aus der aktuellen Episode, welche sich nach dem Zustand der aktuellen Transition befinden.
- *random*: Wähle zufällig k beliebige Zustände aus beliebigen Episoden.

In der Evaluierung 4.3 wird die *future* Methode mit $k = 8$ verwendet, da diese in dem zitierten Paper die besten Ergebnisse erzielt hat.

3.2.6 Curriculum Learning

Bei *Curriculum Learning* wird die Aufgabe für den Agenten schrittweise erschwert [5]. Durch das Training von Teilaufgaben soll der Agent langsam an die komplexe Aufgabe herangeführt werden. Dies kann auf verschiedene Weise geschehen. Eine Methode ist das Erhöhen des Schwierigkeitsgrades der Startzustände nach jeder Episode. Somit wird der Schwierigkeitsgrad zu Trainingsbeginn auf einfach gesetzt und steigert sich auf den schwierigsten Grad. Die Definition für den Schwierigkeitsgrad des Startzustandes ist in Abschnitt 3.1.4 gegeben.
Die Erhöhung des Schwierigkeitsgrades ist dabei abhängig von der Explorationsrate ϵ, welche ebenfalls zur *Exploration* durch Rauschen eingesetzt wird (siehe Abschnitt 3.2.4). Zum Zeitpunkt t einer neuen Episode wird der Startzustand mit

$$s_{Start} = (1 - \epsilon_t)(s_{schwierig} - s_{einfach}) + s_{einfach} \tag{3.38}$$

gesetzt, wobei $s_{einfach}$ und $s_{schwierig}$ die Startzustände aus dem jeweiligen Schwierigkeitsgrad sind.
Durch diese Methode erlernt der Agent zunächst die Zustände im Zielbereich. Dies hat zum Vorteil, dass bei Erhöhen des Schwierigkeitsgrades nur ein kleiner Zustandsbereich

neu hinzukommt und der Agent schnell die Orientierung zum Zielbereich finden kann. Die guten Ergebnisse, die sich mit dieser Methode erzielen lassen, zeigen sich in Abschnitt 4.2.1.

4 Evaluierung

In diesem Kapitel wird der Algorithmus evaluiert. Durch die gegebene Problembeschreibung wird eine Standardsimulation erstellt. An dieser Simulation wird überprüft, wie der Agent die Aufgabe löst. In der Implementierung des Algorithmus wurden teilweise verschiedene Varianten vorgestellt. In der Evaluierung zeigt sich, dass einige davon große Auswirkungen auf das Training haben. Insbesondere wird untersucht, wie die Techniken PER und HER das Training von Umgebungen mit *sparse* Reward ermöglichen.
Um einen Vergleich zwischen verschiedenen Varianten zu ermöglichen, wird jede modifizierte Simulation mehrmals durchgeführt. Die Simulationsergebnisse werden dann zu einem Mittelwert zusammen geführt, um das Ergebnis zu validieren.

4.1 Standardsimulation

Die Standardsimulation dient dem Vergleich für die Auswertung von anderen Simulationen. In den folgenden Abschnitten wird gezeigt, wie sich verschiedene Methoden auf die Simulation auswirken.

4.1.1 Definition der Standardsimulation

Die hier definierten Einstellungen für die Simulation gelten als Standard:

- Simulationszeit: 10.000 Episoden
- Episodenlänge: $10s$ (Das entspricht 200 Schritten pro Episode bei einer Simulationsschrittgröße von $0{,}05s$.)
- Aktionsraum: 4 Dimensionen
- Zustandsraum: 15 Dimensionen

- Startzustand : normaler Schwierigkeitsgrad
- kein *Curriculum Learning*
- Diskontierungsfaktor: $\gamma = 0{,}9$
- Reward-Funktion: r_{shaped}
- Rauschen: *Parameter Space Noise*
- Länge der Aufwärmphase: 2.000 Episoden (Das entspricht 120.000 Transitionen bei einer Episodenlänge von $3s$ während der Aufwärmphase.)
- Experience Replay Speichergröße: 320.000 Transitionen
- keine Verwendung von PER und HER
- Minibatchgröße: 128 Transitionen
- Netzgrößenfaktor: $x_N = 6$
- Verlustfunktion für das Training des Critics: *Huber loss*

Bei Auswertungen, deren Methoden vom Standard abweichen, werden diese explizit genannt.

4.1.2 Evaluierung der Standardsimulation

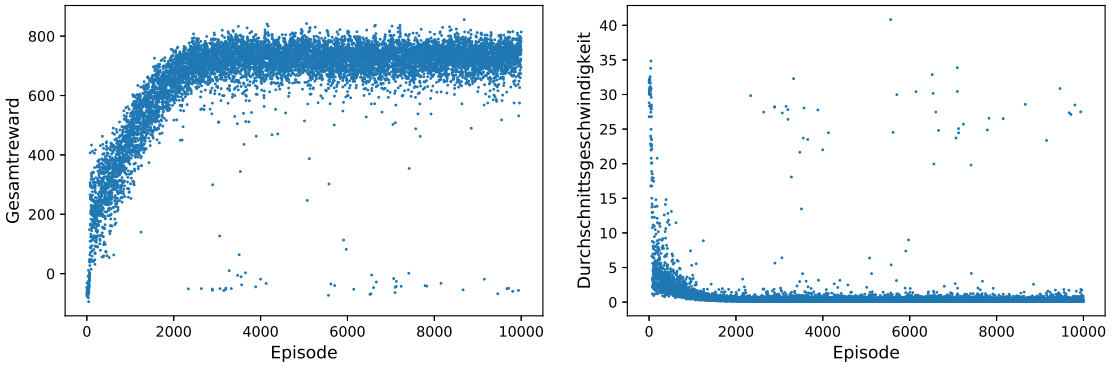
Die Durchführung einer Simulation beträgt circa neun Stunden. Die Rechenzeit variiert je nach eingesetztem PC-System und dessen Auslastung durch parallele Durchführungen. Insgesamt kamen drei verschiedene PC-Systeme mit folgenden CPUs zum Einsatz:

Tabelle 4.1: CPUs der eingesetzten PC-Systeme

CPU	Kerne	Threads	Basisgeschwindigkeit
Intel Core i7-6820HQ	4	8	2,71 GHz
Intel Core i5-4570	4	4	3,2 GHz
AMD Ryzen 7 2700X	8	16	3,7 GHz

Trotz Verfügbarkeit einer leistungsstarker GPU hat sich die Simulationszeit durch diese nicht verbessert. Der Grund dafür ist, dass bei jedem Training die Trainingsdaten an

die GPU übermittelt werden müssen. Daher lohnt sich ein GPU-Einsatz erst bei großen Mengen von Trainingsdaten. Diese Simulation beinhaltet nur 128 Transitionen pro Minibatch. Somit findet das gesamte Training auf der CPU statt.



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.1: Ergebnis der Standardsimulation

Als Maßstab für die Simulationsergebnisse ist der Gesamtreward (siehe Abbildung 4.1a) und die Durchschnittsgeschwindigkeit (siehe Abbildung 4.1b) gegeben. Der Gesamtreward ergibt sich aus den summierten Rewards jeder Episode. Dadurch ergibt sich aufgrund der Definition der *shaped* Reward-Funktion ein maximaler Gesamtreward von 1000. Der minimale Wert liegt bei -250. Die Durchschnittsgeschwindigkeit berechnet sich durch den Betrag des Geschwindigkeitsvektors wie folgt:

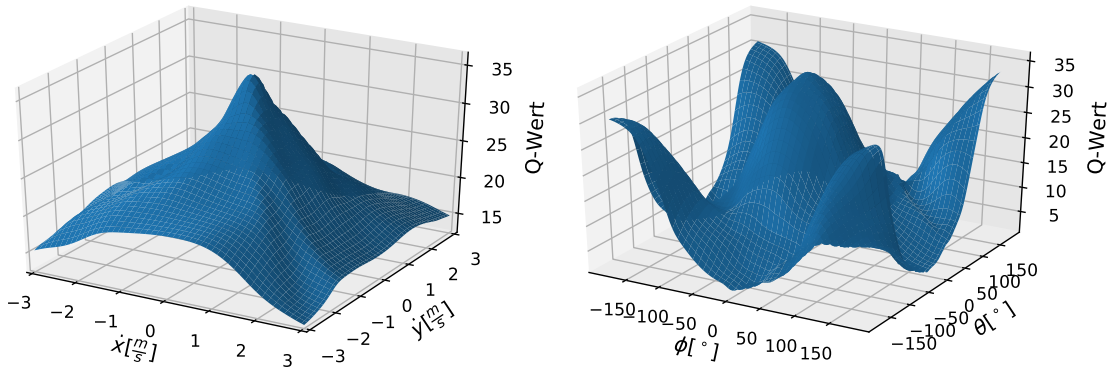
$$|\dot{\xi}| = \left| \begin{pmatrix} \dot{x} \\ \dot{y} \\ \dot{z} \end{pmatrix} \right| = \sqrt{\dot{x}^2 + \dot{y}^2 + \dot{z}^2} \tag{4.1}$$

Für jede Episode ist der durchschnittliche Betrag des Geschwindigkeitsvektors eingetragen. Dieser sollte idealerweise bei 0 liegen.

Die Simulationsergebnisse in Abbildung 4.1 zeigen eine schöne Lernkurve. Gleich zu Beginn in den ersten 100 Episoden ist ein schneller Anstieg zu sehen. In dieser Phase wird zunächst die Orientierung nach oben erlernt. In der nächsten Phase bis circa Episode 3000 wird die Strategie optimiert. Ab dann ist keine Steigerung mehr vorhanden. Insgesamt wird nur ein maximaler Gesamtreward von circa 800 erreicht. Dies liegt jedoch an dem Rauschen, welches die Aktionswahl während des Trainings beeinflusst. Bei einer Simulation nach dem Training ohne Rauschen erlangt der Agent durch die gewonnene Strategie auch höhere Gesamtrewards.

In der letzten Phase kommen zusätzlich vereinzelte Ausreißer dazu. Ein Ausreißer ist eine Episode, die trotz weit fortgeschrittenem Training eine Durchschnittsgeschwindigkeit von $5\frac{m}{s}$ überschreitet. Der Grund für einen Ausreißer ist meist das Betreten eines unerkundeten Zustandsbereiches.
Die finale Ausreißerquote definiert den prozentualen Anteil an Ausreißern im letzten Fünftel des Trainings. Mit Hilfe dieser Quote können Simulationen miteinander verglichen werden. In der Standardsimulation liegt die finale Ausreißerquote bei 0,97%.

Die Abbildungen 4.2 und 4.3 zeigen einen Bereich von dem was das Critic-Netz und das Actor-Netz nach 10.000 Episoden trainiert haben. Aufgrund der hohen Dimensionalität ist es nicht möglich die beiden Netze vollständig zu visualisieren. Daher werden jeweils nur zwei Dimensionen aus dem Zustandsraum genommen, die über die Achsen des Graphen dargestellt werden. Diese sind zum einem $\dot{x}$ und $\dot{y}$ und zum anderem ϕ und θ. Alle restlichen Elemente aus dem Zustands- und Aktionsraum sind 0. Dadurch wird für jeden Graphen ein Ausschnitt aus dem Zentrum erstellt.
Die Abbildung 4.2 zeigt die Q-Werte des Critics im Zentrum. Diese weisen eine hohe Ähnlichkeit mit den Graphen der *shaped* Reward-Funktion (siehe Abbildung 3.6) auf, was ein Indiz für ein erfolgreiches Training ist.

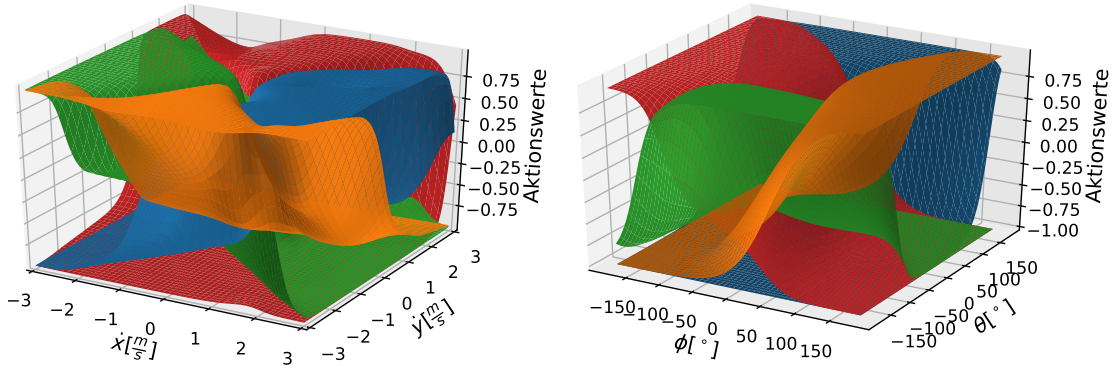


(a) $Q_\pi(s, a|\theta^Q)$ mit allen Zustands- und Aktionswerten gleich 0 bis auf $\dot{x}$ und $\dot{y}$

(b) $Q_\pi(s, a|\theta^Q)$ mit allen Zustands- und Aktionswerten gleich 0 bis auf ϕ und θ

Abbildung 4.2: Q-Werte des Critics im Zentrum

Die vier Aktionswerte des Actors im Zentrum sind in Abbildung 4.3 gezeigt. Es ist zu erkennen, dass die Ausrichtung von den vier Aktionselementen jeweils um 90° gedreht ist. Dies lässt ebenfalls auf ein erfolgreiches Training des Actors vermuten, da andernfalls die Kurvenverläufe ungeordnet wären.



(a) $\mu_\pi(s|\theta^\mu)$ mit allen Zustandswerten gleich 0 bis auf $\dot{x}$ und $\dot{y}$

(b) $\mu_\pi(s|\theta^\mu)$ mit allen Zustandswerten gleich 0 bis auf ϕ und θ

Abbildung 4.3: Aktionswerte des Actors im Zentrum

4.2 Evaluierung von Anpassungen des Algorithmus

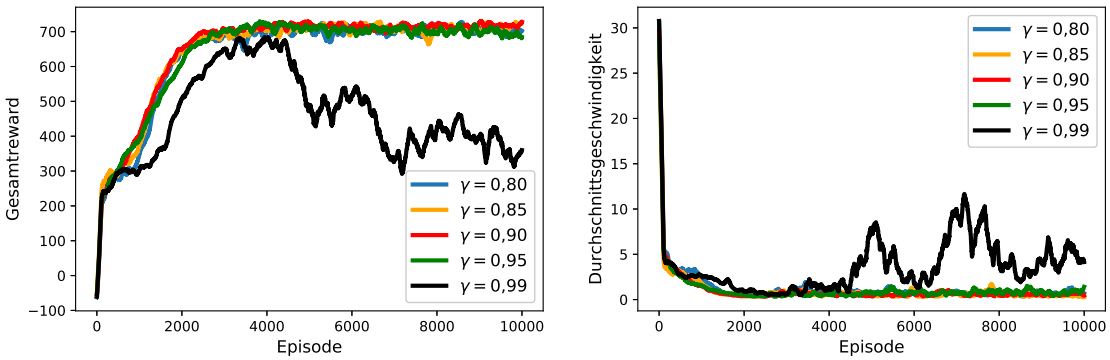
Der Algorithmus bietet viele Möglichkeiten für Variationen, wovon einige große Auswirkungen auf das Trainingsverhalten des Agenten haben. Zu den Anpassungen gehören Parameter des Algorithmus, Netzeigenschaften, Dimensionalität von Zustands- und Aktionsraum, sowie verschiedene Einstellungen für den Experience Replay Speicher.

4.2.1 Evaluierung von Parametern und Methoden

In diesem Abschnitt werden verschiedene Parameter und Methoden miteinander verglichen. Die gezeigten Lernkurven für den Gesamtreward und die Durchschnittsgeschwindigkeit sind zur Übersichtlichkeit zu einem Mittelwert zusammengesetzt. Der Mittelwert für eine Episode ist jeweils der Durchschnitt aus 100 benachbarten Episoden. Zudem bildet jede Lernkurve ihren Mittelwert aus bis zu fünf Simulationsdurchläufen.

Diskontierungsfaktor

Der Diskontierungsfaktor γ bestimmt die Gewichtung zukünftiger Rewards. Daher sollte der Faktor möglichst groß gewählt werden. Abbildung 4.4 zeigt aber auch, dass ein zu hoher Faktor keine guten Ergebnisse bringt (schwarze Lernkurve). Die niedrigste finale Ausreißerquote mit 0,58% ist bei $\gamma = 0{,}85$.

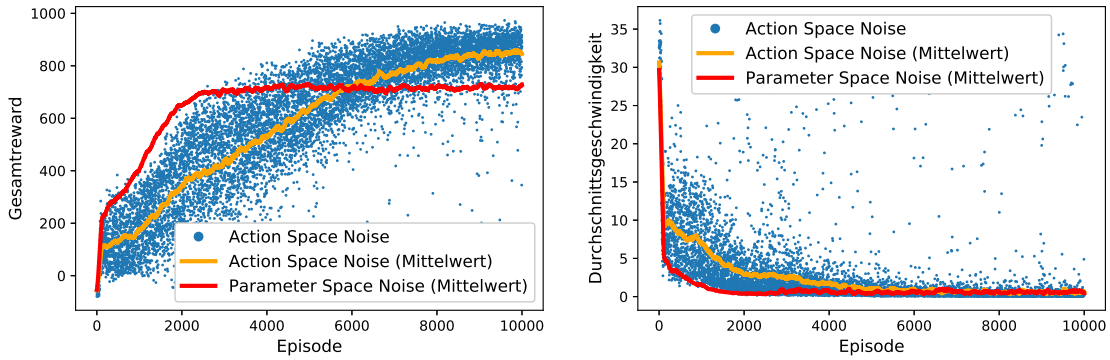


(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.4: Vergleich von verschiedenen Diskontierungsfaktoren

Rauschen

Die Aktionen für den Agenten sind mit einem Rauschen behaftet. In der Standardsimulation wurde *Parameter Space Noise* eingesetzt. Die Abbildung 4.5 zeigt die Verwendung von *Action Space Noise* im Vergleich. Die Beschreibung der beiden Rauschmethoden ist in Abschnitt 3.2.4. Bei der Verwendung von *Action Space Noise* ist der Anstieg der Lernkurve zwar gleichmäßig, aber im Vergleich deutlich langsamer. Dafür scheint es, als ob diese Methode die besseren Ergebnisse zum Trainingsende erlangt. Jedoch entstehen diese hohen Gesamtrewards durch die Verwendung von ϵ-Reduktion, was bedeutet, das bei *Action Space Noise* zum Trainingsende kaum mehr Rauschen vorhanden ist. Ein Agent der mit *Parameter Space Noise* trainiert wurde, erzielt nach dem Training mit seiner Strategie gleichwertige Gesamtrewards.

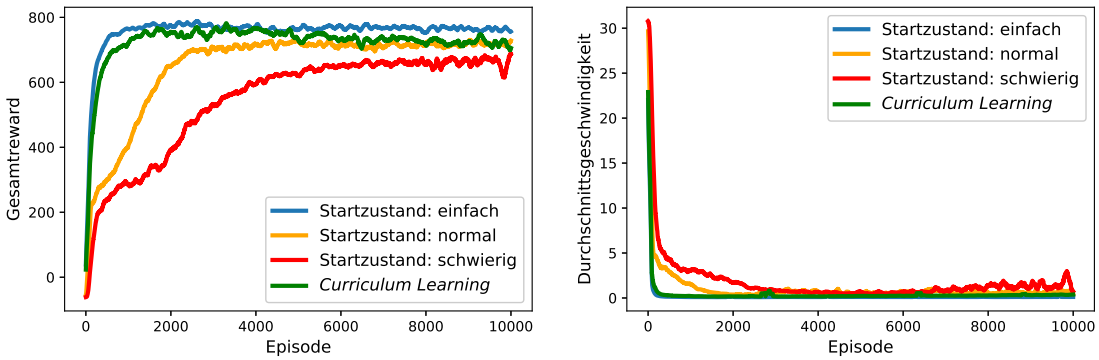


(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.5: Vergleich von *Action Space Noise* und *Parameter Space Noise*

Startzustand

Der Bereich für den Startzustand einer Episode lässt sich in drei Schwierigkeitsgraden definieren. Der Verlauf der Lernkurven (siehe Abbildung 4.6) ist erwartungsgemäß entsprechend dem Schwierigkeitsgrad gestaffelt. Das Training der schwierigen Startzustände benötigt zwar ein wenig mehr Episoden, aber auch diese Zustände werden vom Agenten erlernt.



(a) Gesamtreward pro Episode

(b) Durchschnittsgeschwindigkeit pro Episode

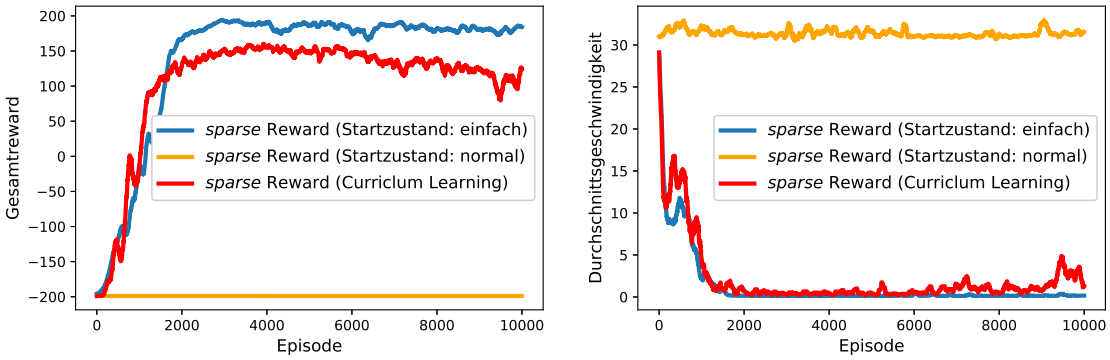
Abbildung 4.6: Vergleich von verschiedenen Startzuständen

Besonders positiv fällt die Verwendung von *Curriculum Learning* auf. Da die Startzustände langsam schwieriger werden, ist zu Trainingsbeginn eine ähnliche Lernkurve wie bei einfachen Startzuständen zu sehen. Dadurch hat der Agent die Zustände im Zielbereich früh trainiert und kann so die schwierigeren Zustände insgesamt schneller erlernen. Dies zeigt sich auch in der finalen Ausreißerquote bei *Curriculum Learning* von 0,05%. Im Gegensatz dazu hat der Agent bei den schwierigen Startzuständen eine Quote von 3,92%.

Reward-Funktion

Die zuvor gezeigten Simulationen liefen unter Verwendung der *shaped* Reward-Funktion. Die Abbildung 4.7 zeigt, wie schwer es unter normalen Umständen für den Agenten ist einen *sparse* Reward zu entdecken. Durch Vereinfachen der Aufgabe lernt der Agent Zielzustände kennen und kann diese trainieren. Aus diesem Grund ist auch das *Curriculum Learning* bei *sparse* Reward-Funktionen erfolgreich. Jedoch ist die finale Ausreißerquote

hier mit $4,45\%$ recht hoch. In Abschnitt 4.3.2 wird gezeigt, wie sich *sparse* Reward-Funktionen besser erlernen lassen.



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.7: Simulationsergebnisse bei einer *sparse* Reward-Funktion

4.2.2 Evaluierung von Netzeigenschaften

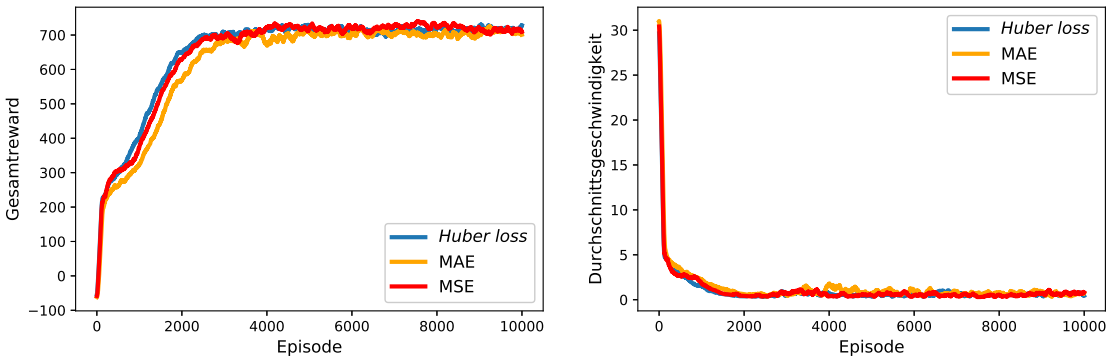
Für das Training des Critic Netzes wird ein Verlust (*loss*) minimiert. Neben den üblichen Verlustfunktion MAE und MSE, wird *Huber loss* als alternative Verlustfunktion evaluiert. Die Größe des Netzes und des Minibatches können ebenfalls das Training beeinflussen.

Verlustfunktion

Die Verlustfunktion wird beim Training des Critic Netzes eingesetzt. Die Abbildung 4.8 zeigt, dass die Wahl der Verlustfunktion bei der Standardsimulation keine Auswirkung auf das Training hat. Auch die finale Ausreißerquote ist bei den drei Funktionen gleichwertig. Eventuell bietet *Huber loss* einen größeren Vorteil in Kombination mit anderen Methoden oder bei einer anderen Netzarchitektur.

Netzgröße

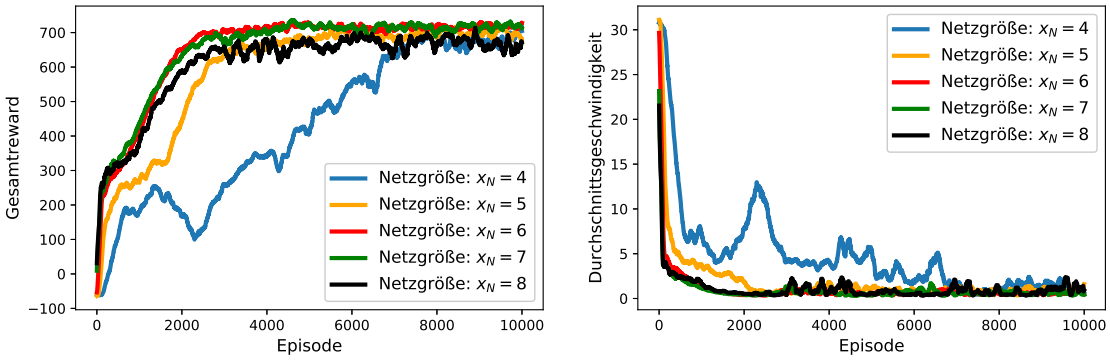
Der Faktor x_N bestimmt die Größe der beiden Netze von Actor und Critic. Die gesamte Anzahl von Neuronen in beiden Netzen ist dabei: $10 \cdot 2^{x_N} + 5$. Dies ist durch die Netzarchitektur in Abschnitt 3.2.2 gegeben. Die Abbildung 4.9 zeigt, dass ein zu kleines Netz mit



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.8: Vergleich von verschiedenen Verlustfunktionen

einem Faktor von $x_N = 4$ das Training zu sehr verlangsamt. Ab einem Wert von $x_N = 6$ ergeben sich keine weiteren Vorteile durch das Vergrößern der Netze. Im Gegenteil steigt die benötigte Rechenzeit ab diesem Punkt massiv an. Die Rechenzeit, welche in Tabelle 4.2 enthalten ist, ist die Zeit, die eine CPU (hier: AMD Ryzen 7 2700X) benötigt, um beide Netze einmal mit einen Minibatch zu trainieren.



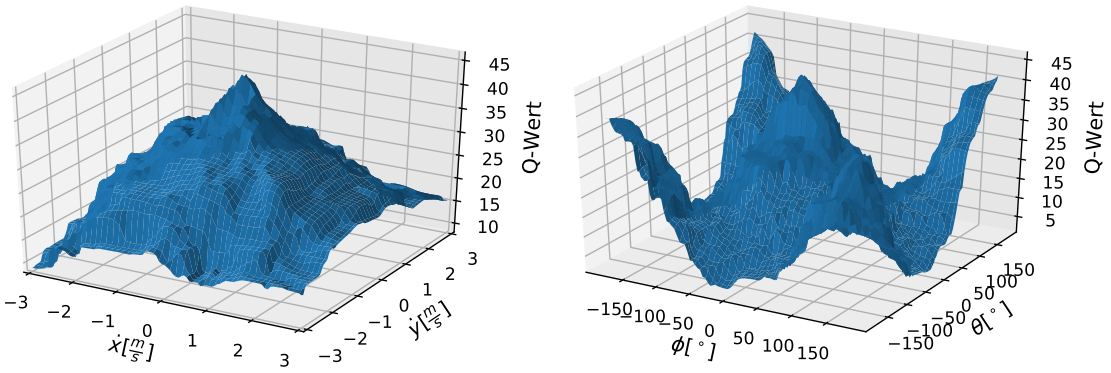
(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.9: Vergleich von verschiedenen Netzgrößen

Des Weiteren wird mit steigender Netzgröße die Approximation der Q-Werte durch das Critic-Netz feinstufiger. So sind in Abbildung 4.10 die Q-Werte für eine Simulation mit einem Netz der Größe $x_N = 7$ gezeigt. Die Approximation der Standardsimulation (siehe Abbildung 4.2) ist im Vergleich deutlich glatter.

Tabelle 4.2: Simulationen mit verschiedenen Netzgrößen

Netzgröße	Neuronenanzahl	Rechenzeit	finale Ausreißerquote
$x_N = 4$	165	4,0ms	4,17%
$x_N = 5$	325	4,4ms	2,08%
$x_N = 6$	645	5,5ms	0,97%
$x_N = 7$	1285	7,7ms	1,27%
$x_N = 8$	2565	15,2ms	1,60%



(a) $Q_\pi(s, a|\theta^Q)$ mit allen Zustands- und Aktionswerten gleich 0 bis auf $\dot{x}$ und $\dot{y}$

(b) $Q_\pi(s, a|\theta^Q)$ mit allen Zustands- und Aktionswerten gleich 0 bis auf ϕ und θ

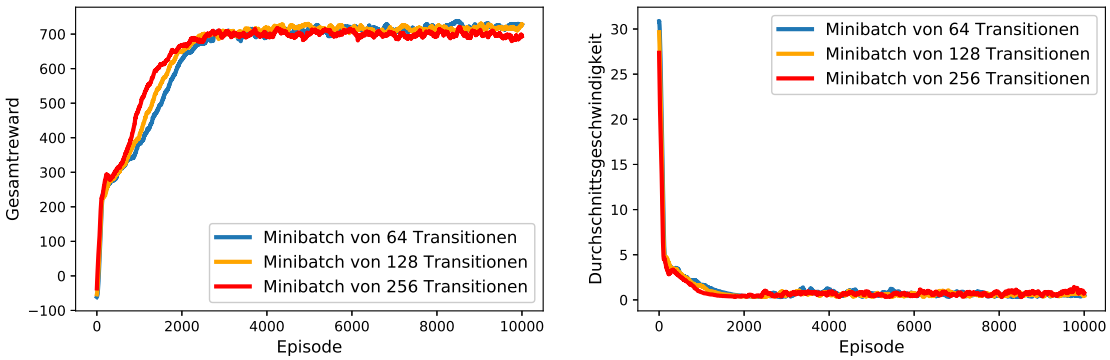
Abbildung 4.10: Q-Werte des Critics im Zentrum bei einer Netzgröße von $x_N = 7$

Minibatchgröße

Die Minibatchgröße N bestimmt die Anzahl an Transitionen in einem Minibatch. Eine Standardsimulation beinhaltet 128 Transitionen in einem Minibatch. In Algorithmus 1 wird festgelegt, dass in jeder Episode T-mal ein neues Minibatch erstellt und trainiert wird. Mit T, der Anzahl an Episodenschritten, gleich 200 Schritten ergeben sich 25.600 Transitionen, die pro Episode trainiert werden. Die Abbildung 4.11 zeigt, dass ein Halbieren oder Verdoppeln dieser Menge keine Auswirkung auf die Lernkurve hat. Allerdings steigt die benötigte Rechenzeit mit steigender Minibatchgröße.

4.2.3 Evaluierung der Dimensionalität

In Abschnitt 3.1 wurde der Zustands- und Aktionsraum definiert. Je nach Problem kann dieser unterschiedlich gestaltet werden. Aber auch für ein konkretes Problem gibt es Ge-



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.11: Vergleich von verschiedenen Minibatchgrößen

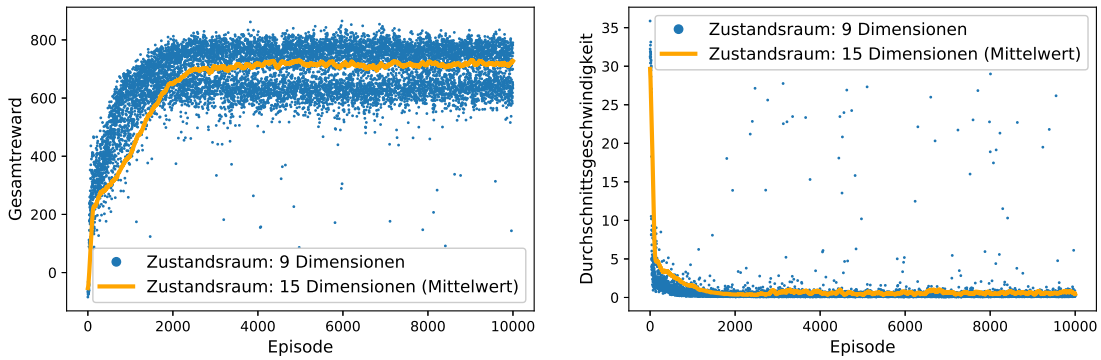
staltungsfreiraum. Die unterschiedlichen Varianten verändern dabei die Dimensionalität der Räume, was Auswirkungen auf das Training haben kann.

Dimensionen des Zustandsraumes

Durch Entfernen der Zustandselemente, die die Beschleunigung der Position ($\ddot{x}$, $\ddot{y}$ und $\ddot{z}$) und der Rotation ($\ddot{\phi}$, $\ddot{\theta}$ und $\ddot{\psi}$) beinhalten, wird der Zustandsraum auf neun Dimensionen reduziert. Durch den kleineren Zustandsraum ist der Verlauf Lernkurve zu Trainingsbeginn steiler (siehe Abbildung 4.12). Für das Training der Zustände im Zielbereich ist jedoch mehr Feinmotorik nötig. Deswegen ist es dort hilfreich die Beschleunigungswerte in das Training zu integrieren. In Abbildung 4.12a ist zudem ein Spalt in der Lernkurve (blau) zu erkennen. Grund dafür könnte ebenfalls die fehlende Information über die Beschleunigung sein.

Dimensionen des Aktionsraumes

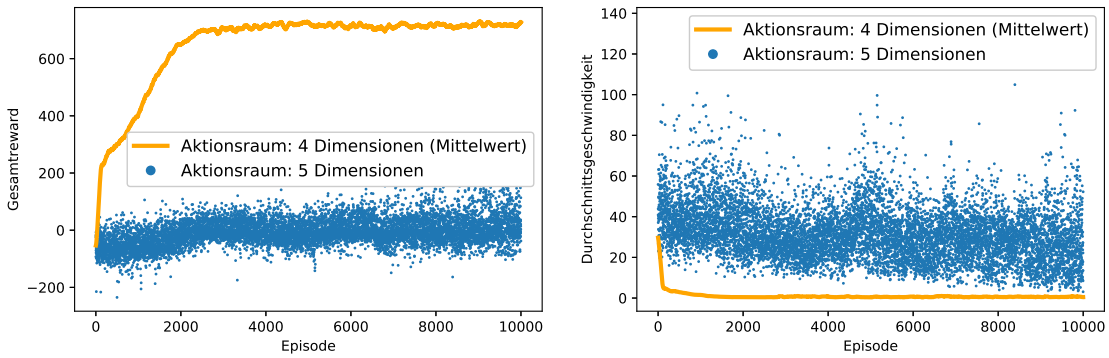
Die Definition für die Varianten mit vier und mit fünf Dimensionen befindet sich in Abschnitt 3.1.2. Die Variante mit fünf Dimensionen erzielt kein erfolgreiches Ergebnis (siehe Abbildung 4.13). Dabei ist nicht die Anzahl an Dimensionen die Ursache, sondern die Funktionsweise der Variante mit fünf Dimensionen. Diese erlaubt keine großen Unterschiede zwischen den einzelnen Rotorgeschwindigkeiten. Der Erkenntnisgewinn aus diesem Vergleich ist, dass bei der Definition eines Aktionsraumes dem Agenten genug



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.12: Vergleich von verschiedenen Zustandsräumen

Freiraum gelassen werden sollte, um die Werte selbst zu wählen und ihn nicht einzugrenzen.



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

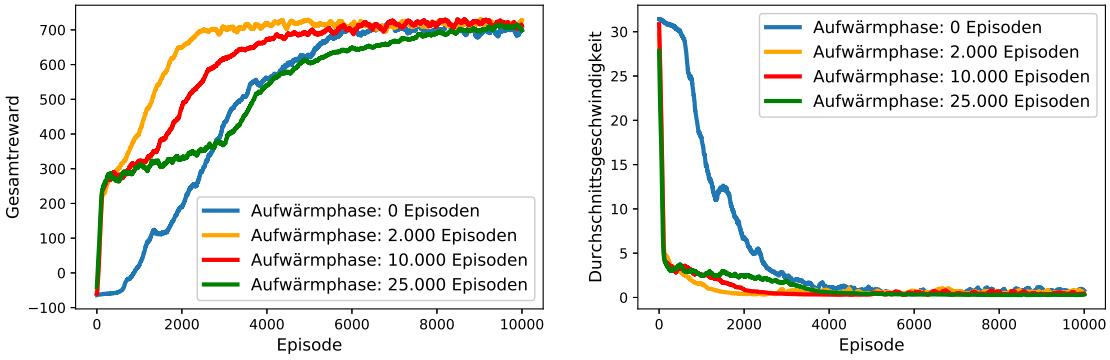
Abbildung 4.13: Vergleich von verschiedenen Aktionsräumen

4.2.4 Evaluierung des Experience Replay Speichers

Ohne den Speicher ist ein Training nicht möglich. Dies zeigt die Evaluierung der Simulation mit sehr kleiner Speichergröße. Die Trainingsdaten zu Simulationsstart stammen aus der Aufwärmphase. Die Evaluierung zeigt, wie hilfreich diese Aufwärmphase für das gesamte Training ist.

Aufwärmphase

Die Länge der Aufwärmphase bestimmt wie viele Transitionen bereits zum Simulationsstart in den Experience Replay Speicher geladen werden. Somit stehen gleich zu Trainingsbeginn genügend Transitionen zur Auswahl. Damit alle Transitionen, die durch die Aufwärmphase in den Speicher geladen werden, in den Speicher passen, ist die Speichergröße entsprechend der Aufwärmphase angepasst.
Abbildung 4.14 zeigt wie wertvoll diese Transitionen zu Beginn sind. Wird die Aufwärmphase weggelassen (blaue Lernkurve), so tritt ein Lernerfolg erst ein, sobald sich der Speicher durch die Simulation langsam füllt. Bei einer langen Aufwärmphase (grüne Lernkurve) ist der Lernerfolg zu Trainingsbeginn zwar stark, jedoch verlangsamt sich die Lerngeschwindigkeit in der mittleren Trainingsphase. Der Grund dafür ist, dass die Transitionen aus der Aufwärmphase mit zufälligen Aktionswerten erzeugt werden und somit wenig Transitionen aus dem Zielbereich beinhalten. Bei Betrachtung der finalen Ausreißerquote lohnt sich jedoch eine längere Aufwärmphase, denn die Quote von der Aufwärmphase mit 25.000 Episoden liegt bei 0,02%. Durch die Vielzahl an Zuständen, die durch die lange Aufwärmphase abgedeckt werden, werden potenzielle Ausreißer mit in das Training aufgenommen und insgesamt ist die *Exploration* dadurch höher.



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.14: Vergleich von verschiedenen Längen der Aufwärmphase

Die Variable σ_G, welche beim *Parameter Space Noise* für die Rauschbestimmung genutzt wird, steuert die *Exploration* (siehe Abschnitt 3.2.4). Der Vergleich der Werte von σ_G einer Simulation mit einer langen und einer ohne Aufwärmphase ist in Abbildung 4.15 gezeigt. Ohne Aufwärmphase sinkt σ_G zunächst stark, wodurch die *Exploitation* steigt. Dies sorgt dafür, dass zunächst eine Strategie entsteht, die es dem Agenten ermöglicht

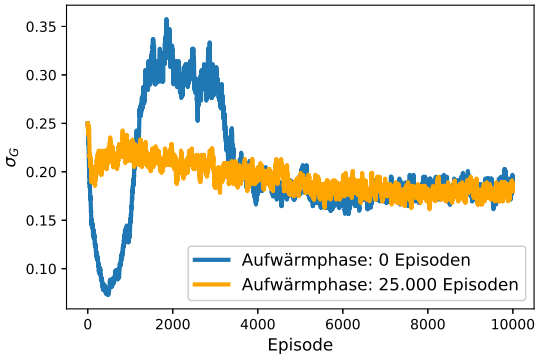
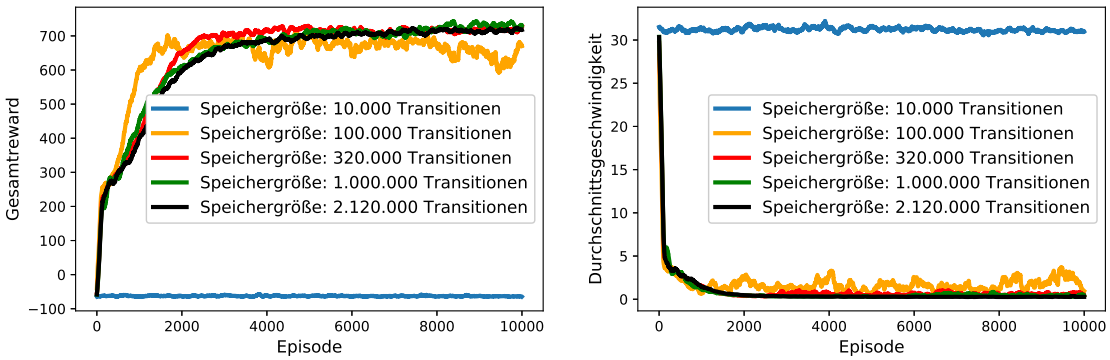


Abbildung 4.15: Verlauf von σ_G des *Parameter Space Noise* während verschiedener Simulationen

den Zielbereich anzusteuern. Sobald die Strategie dies gewährleisten kann, steigt die *Exploration*, um weitere Zustände zu erkunden. Anschließend stabilisiert sich σ_G bei einem Wert, der ein geeignetes Verhältnis zwischen *Exploration* und *Exploitation* herstellt. Im Vergleich dazu sind die Schwankungen der Verlaufskurve von σ_G bei einer langen Aufwärmphase wesentlich geringer. Die große Menge an bestehende Transitionen aus der Aufwärmphase beinhaltet viele verschiedene Zustände. Daher braucht der Agent durch ein *Parameter Space Noise* nicht mehr dazu geleitet werden weitere unbekannte Zustände in dem gleichen Maße zu erkunden.

Speichergröße

Die Größe des Experience Replay Speicher legt fest wie viele Transitionen zur Auswahl für einen Minibatch stehen. Für eine Simulation mit 10.000 Episoden und einer Aufwärmphase von 2.000 Episoden ist die maximale Anzahl von möglichen Transitionen 2.120.000. In der Standardsimulation wurde eine Speichergröße von 320.000 Transitionen gewählt. Dies bietet genug Speicher für alle Transitionen aus der Aufwärmphase (120.000) und sorgt dafür, dass der Speicher die Transitionen aus der Simulation alle 1.000 Episoden im Speicher überschreibt. Die Abbildung 4.16 zeigt, dass jede Speichergröße zu einem erfolgreichen Training führt. Jedoch ist ohne den Speicher kein Training möglich. Ebenfalls eine geringe Speichergröße von 10.000 Transitionen (blaue Lernkurve) bringt keine guten Ergebnisse.
Es empfiehlt sich eine große Speichergröße zu wählen, da mit steigender Größe die finale Ausreißerquote sinkt. Bei der maximalen Speichergröße beträgt diese nur noch 0,03%.



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.16: Vergleich von verschiedenen Speichergrößen

4.3 Evaluierung von HER und PER

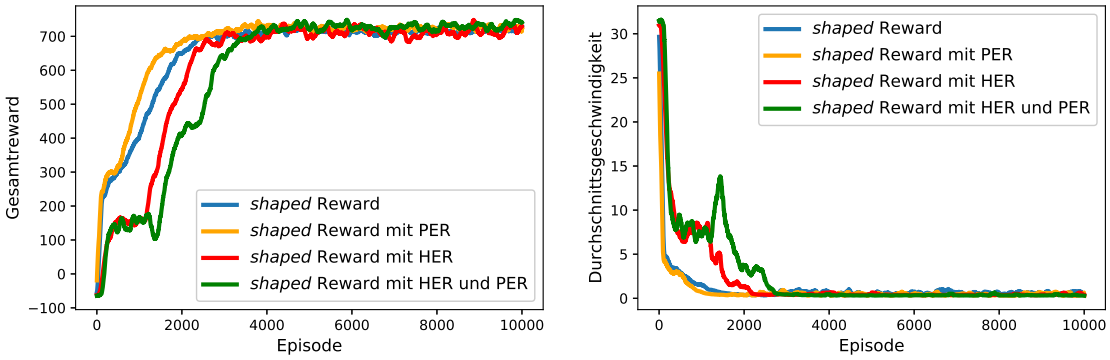
Die beiden Techniken *Hindsight Experience Replay* (HER) und *Priorized Experience Replay* (PER) sind mit jeder vorgestellten Methode kombinierbar, da diese ausschließlich die Transitionen im Speicher und deren Selektierung beeinflussen. Insbesondere eine Kombination der beiden Techniken führt in der Simulation zu den besten Ergebnissen.
Es werden jeweils die Techniken einzeln und in Kombination verglichen. Bei der Evaluierung wird zwischen Simulationen mit einer *sparse* und einer *shaped* Reward-Funktion unterschieden. Insgesamt werden 16 verschiedene Simulationen mit verschiedenen Schwierigkeitsgraden (einfach, mittel und *Curriculum Learning*) verglichen. Die Tabelle 4.3 zeigt alle Kombinationen und deren jeweilige finale Ausreißerquote.

Tabelle 4.3: Finale Ausreißerquote von Simulationen mit kombinierten Techniken

	shaped	*sparse*		
	mittel	einfach	mittel	*Curriculum*
Standard	0,97%	0,18%	100,0%	4,45%
Standard + PER	0,62%	0,0%	100,0%	3,13%
Standard + HER	0,18%	99,1%	100,0%	100,0%
Standard + HER + PER	0,08%	0,0%	100,0%	0,73%

4.3.1 Kombinationen mit *shaped* Reward-Funktion

Obwohl die HER Technik insbesondere für Simulationen mit *sparse* Reward-Funktion gedacht ist, kann sie auch mit einer *shaped* Reward-Funktion eingesetzt werden. Die Abbildung 4.17 zeigt zwar, dass die Simulationen in Kombination mit HER zu Trainingsbeginn einen Einbruch in der Lernkurve vorweisen, jedoch sind die finalen Ausreißerquoten sehr gut.



(a) Gesamtreward pro Episode

(b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.17: Vergleich von verschiedenen Kombinationen von PER und HER bei Simulationen mit *shaped* Reward-Funktion und normalen Startzuständen

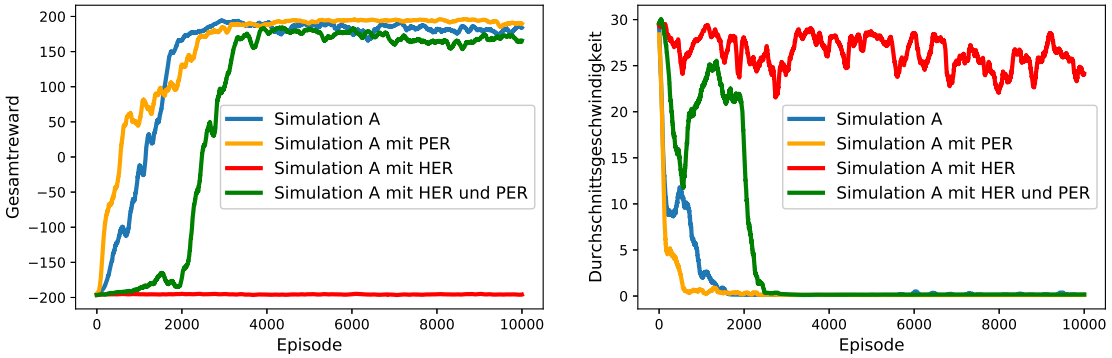
4.3.2 Kombinationen mit *sparse* Reward-Funktion

In Abschnitt 4.2.1 wurde bereits gezeigt, dass das Training durch eine Umgebung mit *sparse* Reward erschwert wird. Die zum Vergleich stehenden Simulationen besitzen die folgenden Eigenschaften:

- Simulation A: *sparse* Reward-Funktion und einfacher Startzustand
- Simulation B: *sparse* Reward-Funktion und mittlerer Startzustand
- Simulation C: *sparse* Reward-Funktion und *Curriculum Learning*

Simulation A

In diesem Vergleich (siehe Abbildung 4.18) unter vereinfachten Umständen erzielt bis auf HER jede Methode gute Ergebnisse. Die Kombinationen mit PER schaffen sogar eine finale Ausreißerquote von 0,0%.



(a) Gesamtreward pro Episode

(b) Durchschnittsgeschwindigkeit pro Episode

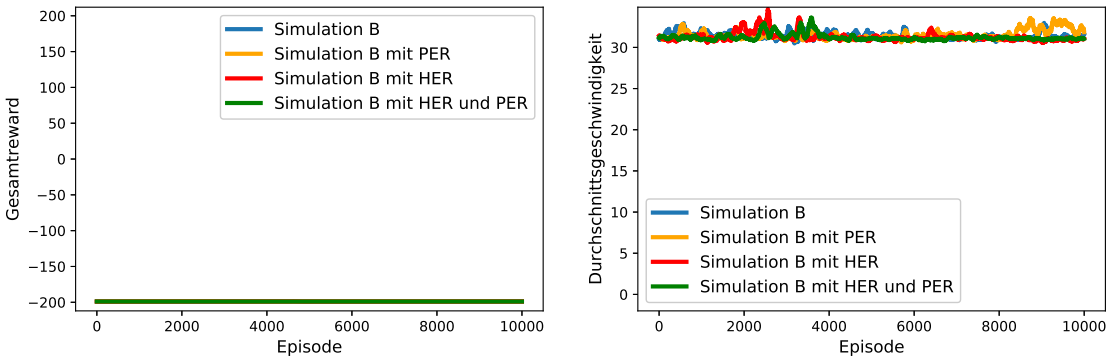
Abbildung 4.18: Vergleich von verschiedenen Kombinationen von PER und HER bei Simulationen mit *sparse* Reward-Funktion und einfachen Startzuständen

Simulation B

Unter normalen Umständen gelingt es keiner der Techniken den Zielbereich zu erkunden (siehe Abbildung 4.19). Die fehlenden Informationen über den Zielbereich sollten durch die HER Technik mit verschiedenen Zielen ausgeglichen werden. Jedoch war dies selbst in Kombination mit PER nicht erfolgreich.

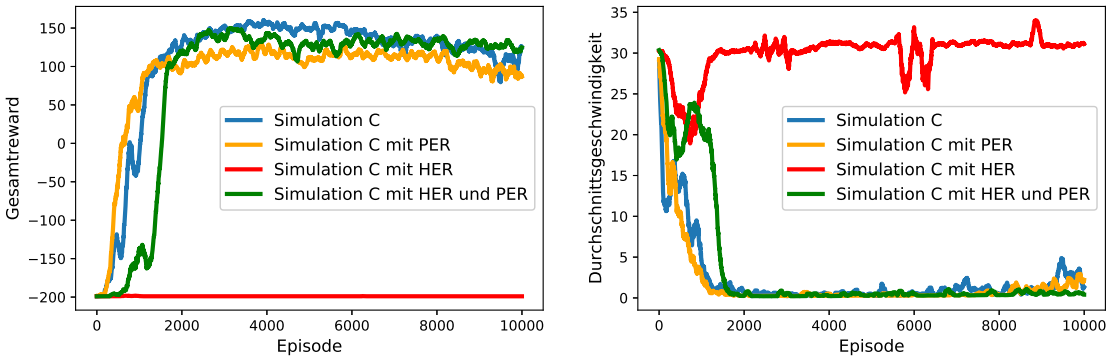
Simulation C

In Abbildung 4.20 zeigt sich die Stärke von *Curriculum Learning*. Bei der Verwendung von *Curriculum Learning* startet der Quadcopter episodenweise aus schwierigeren Startzuständen. Die HER Technik zeigt zu Trainingsbeginn zwar eine richtige Tendenz, doch mit steigendem Schwierigkeitsgrad durch das *Curriculum Learning* verliert HER die Verbindung zum Zielbereich. Eventuell könnte dies durch ein langsameres Erhöhen des Schwierigkeitsgrades verbessert werden. Jedoch erreicht die Kombination von HER und PER eine viermal bessere finale Ausreißerquote als die anderen beiden Techniken.



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.19: Vergleich von verschiedenen Kombinationen von PER und HER bei Simulationen mit *sparse* Reward-Funktion und normalen Startzuständen



(a) Gesamtreward pro Episode (b) Durchschnittsgeschwindigkeit pro Episode

Abbildung 4.20: Vergleich von verschiedenen Kombinationen von PER und HER bei Simulationen mit *sparse* Reward-Funktion und *Curriculum Learning*

4.3.3 Betrachtung des Speichers

Die Speichergröße der Simulationen variiert je nach angewendeter Kombination. Während die Standardsimulationen einen Speicher mit 312.000 Transitionen besitzt, sind es bei PER aufgrund der Baumstruktur 524.288 Transitionen. Bei HER werden pro Episode neunmal so viele Transitionen durch die verschiedenen Zieldefinitionen erstellt. Somit entstehen alleine in der Aufwärmphase mit HER 1.080.000 Transitionen und in der gesamten Simulation werden 18 Millionen weitere erzeugt. Daher wird der Speicher auf eine Größe von 2 Millionen Transitionen begrenzt. Für die Kombination HER und PER ergibt sich eine Speichergröße von 2.097.152 Transitionen.

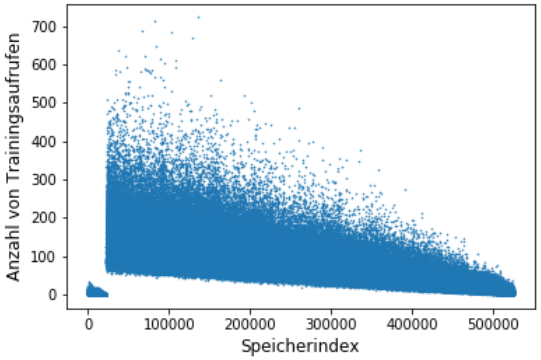
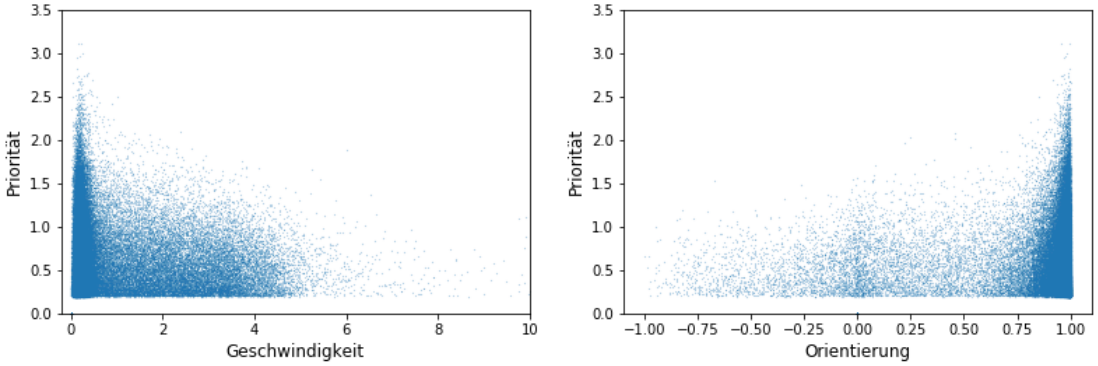


Abbildung 4.21: Anzahl von Trainingsaufrufen einer Transition im Speicher

Im folgenden wird der Speicher nach Beendung von Simulation C mit PER betrachtet. Der Speicher ist vollständig mit 524.288 Transitionen gefüllt, wobei jede Transition zählt, wie oft sie bereits trainiert wurde. Die Abbildung 4.21 zeigt, dass jede Transition im Laufe ihrer Existenz mindestens 50 mal trainiert wird. Zudem ist zu erkennen, wie der FIFO-Buffer alte Transitionen löscht und mit neuen ersetzt. Die Anzahl der neuen Transitionen ist dementsprechend gering.



(a) Prioritätsverteilung in Abhängigkeit von der Geschwindigkeit

(b) Prioritätsverteilung in Abhängigkeit von der Orientierung

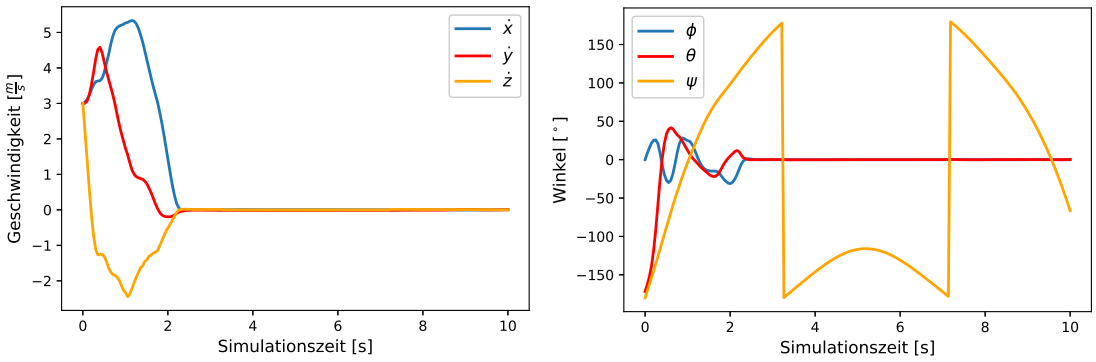
Abbildung 4.22: Prioritätsverteilung

Die Abbildung 4.22 zeigt die Verteilung der Prioritäten in Abhängigkeit von der Geschwindigkeit und der Orientierung. Jeder Punkt entspricht dabei einer Transition, dessen Prioritätswert im Graphen eingetragen ist. Die Geschwindigkeit entspricht dabei dem Geschwindigkeitsvektor aus Gleichung 4.1. Die Orientierung ist in Gleichung 3.20 definiert und beschreibt die Ausrichtung des Quadcopters nach oben (+1) oder unten (−1). Zum einem ist bei den beiden Verteilungen zu erkennen, dass der größte Teil der Tran-

sitionen aus der Nähe des Zielbereiches stammt. Zum anderen ist zu erkennen, dass die höchste Priorität Transitionen aus dem Zielbereich haben. Dies unterstützt den Agenten in einer Umgebung mit *sparse* Reward das Ziel zu erkunden. Die meisten Transitionen haben jedoch eine geringe Priorität und werden kaum mehr für das Training selektiert. So haben circa 80% einen Prioritätswert unter 0,4.

4.4 Verlauf der Flugbahn

Nach dem Training kann der Agent die beiden trainierten Netze mit einer *greedy*-Strategie ausnutzen. Dazu wird eine weitere Episode gestartet, in der immer die beste Aktion gewählt wird und diese auch mit keinem Rauschen behaftet ist. Der Startzustand für diese Episode entspricht dem schwersten Schwierigkeitsgrad und ist ein unkontrollierter Flugzustand. Somit beträgt die Geschwindigkeit $\dot{x} = \dot{y} = \dot{z} = 3\frac{m}{s}$. Die Winkel werden mit $\phi = 0°$, $\theta = -180°$ und $\psi = -180°$ so gesetzt, dass der Quadcopter nach unten gerichtet ist. Die Flugbahn des Quadcopters ist in Abbildung 4.23 gezeigt. In der ersten Sekunde wird eine aufrechte Orientierung hergestellt. Während dieser Phase kann noch keine Kontrolle über die Geschwindigkeit des Quadcopters gewonnen werden. Nachdem eine aufrechte Orientierung gegeben ist, wird der Quadcopter stabilisiert. Die gesamte Stabilisierung dauert circa drei Sekunden. Im stabilisierten Zustand rotiert der Quadcopter um seine z'-Achse, jedoch wird dabei die Position nicht verlassen. Dieses Verhalten könnte durch eine veränderte Reward-Funktion bestraft werden. Jedoch hat sich der Agent trotz dem gegebenen Entscheidungsfreiraum für dieses Verhalten entschieden.



(a) Geschwindigkeit $\dot{\xi}$ (b) Orientierung der Winkel η

Abbildung 4.23: Flugbahn

5 Fazit

In diesem Kapitel wird eine Zusammenfassung dieser Arbeit gegeben. Zudem wird ein Ausblick für weitere Arbeiten diskutiert.

5.1 Zusammenfassung

Der in dieser Arbeit implementierte Algorithmus beinhaltet einen selbstlernenden Agenten, der mit RL einen unkontrollierten Flugzustand stabilisiert. Der Algorithmus baut auf dem DDPG-Algorithmus mit seiner Actor-Critic Architektur auf. Während der Simulation werden alle Flugbahnen in Form von Transitionen in einen Experience Replay Speicher geschrieben. Aus diesem werden die Transitionen in Minibatches selektiert und zum Training der Netze verwendet. Durch das Abspeichern der Transitionen können diese beliebig oft für das Training verwendet werden. Zur *Exploration* ist die Aktion mit einem Rauschen behaftet. Dies ermöglicht dem Agenten noch unbekannte Zustände zu entdecken, um potenzielle Flugbahnen kennen zu lernen. Den Vorteil einer guten *Exploration* zeigt sich auch in den Evaluierungsergebnissen der Aufwärmphase. Die Transitionen der Aufwärmphase ermöglichen schon zu Trainingsbeginn Lernerfolge.
Das Ziel dieser Arbeit war es insbesondere ein Training in Umgebungen mit *sparse* Reward zu ermöglichen. Da das Training von *sparse* Reward-Funktionen in der Standardsimulation nicht erfolgreich war, wurden dafür die Techniken PER und HER eingesetzt. Durch PER lassen sich gezielt Transitionen selektieren, die besonders wertvoll für das Training sind. Dieser Vorteil zeigt sich besonders bei Umgebungen mit *sparse* Reward. Da Transitionen mit positiven Reward selten sind, können diese durch PER bevorzugt in das Training integriert werden. Der Einsatz von HER hat keinen Vorteil beim Trainieren von *sparse* Reward-Funktionen erbracht. Jedoch war ein Mehrwert durch diese Technik bei der Betracht der finalen Ausreißerquote der Standardsimulation gegeben. Durch den Einsatz von HER wird somit das Training von *shaped* Reward-Funktionen verbessert. Die Evaluierung hat weiterhin gezeigt, dass *Curriculum Learning* in allen Umgebungen

einen großen Vorteil bringt. Durch das schrittweise Erhöhen der Aufgabenschwierigkeit, kann der Agent in jeder Umgebung bessere Ergebnisse erzielen.

5.2 Ausblick

Der vorgestellte Algorithmus löst das gegebene Problem zwar sehr gut, dennoch gibt es noch weitere Möglichkeiten die Implementierung zu erweitern und zu verbessern. Diese Möglichkeiten werden hier in Aussicht gestellt. Des Weiteren kann das beschriebene Problem noch erweitert werden, wodurch die Komplexität der Aufgabe steigt.

5.2.1 Aerodynamische Effekte

In der Beschreibung der Umgebung (siehe Abschnitt 2.1.1) wurde festgelegt, dass einige Sonderfälle nicht betrachtet werden. Daher kann das Flugmodell für die erweiterte Flugdynamik angepasst werden. Dazu gehören aerodynamische Effekte, die zum Beispiel in Boden- und Deckennähe auftreten.
In der vorgestellten Simulation sind die Rotorwinkelgeschwindigkeiten direkt veränderbar. Dies kann durch die Modellierung eines Rotormotors und dessen Spannungswerte erweitert werden.
Eine weitere Erweiterung wäre die Implementierung von Luftströmungen. Wenn diese zufällig auftreten, verliert die Übergangsfunktion δ von einem Zustand zum Folgezustand ihren Determinismus. Diese Erweiterung würde neue Herausforderungen mit sich bringen.

5.2.2 Vorführung

Eine Vorführung (*demonstration*) ist in diesem Fall eine Flugbahn, die von einem professionellen Modellflieger geflogen wurde. Die Flugdaten werden vor Trainingsbeginn anstatt der Aufwärmphase in den Experience Replay Speicher geladen [16]. Dadurch soll ein schnelleres Training ermöglicht werden. Die Problematik ist die Erstellung solcher Flugdaten. Für viele Szenarien ist es nicht möglich einen Experten auf dem Gebiet zu finden. Eine Alternative ist es, Flugdaten zu nehmen, die mit einer bestehenden Strategie erstellt worden sind. Dies setzt allerdings voraus, dass ein Agent das Problem bereits

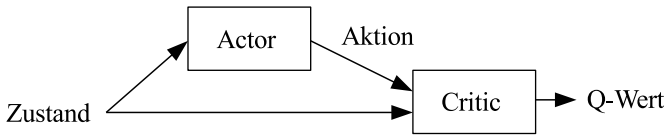
einmal gelöst hat. Somit ist diese Technik nicht immer eine Option. Insbesondere für neue, unerforschte Probleme kann eine Vorführung nicht eingesetzt werden.

5.2.3 Parallelisierung

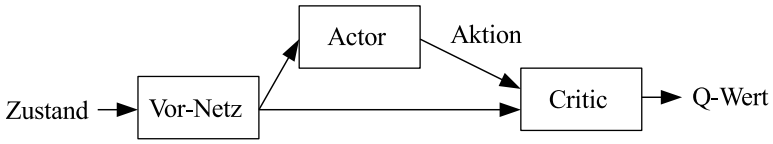
Mit der Parallelisierung von Programmabläufen lässt sich die Rechenzeit der Simulation verkürzen. Jedoch bietet der Algorithmus aufgrund seines sequentiellen Ablaufs nur wenig Möglichkeiten dazu. In dieser Implementierung sind Funktionen, wie das Erstellen eines Minibatches und das Netztraining, parallelisiert. Die Parallelisierung des Netztrainings übernimmt dabei *TensorFlow* entweder auf der CPU oder GPU.
Durch eine Umstrukturierung des Algorithmus lässt sich dieser massiv parallelisieren [17]. Dafür muss das Training von der Simulation entkoppelt werden. Dies ist durch den Experience Replay Speicher gut realisierbar. Der Algorithmus wird in zwei Klassen mit getrennten Ressourcen geteilt. Die eine Klasse simuliert die Flugbahn und benutzt dafür eine Kopie der Strategie, welche in regelmäßigen Abständen aktualisiert wird. Die erstellten Flugbahnen werden nach jeder Episode in Form von Transitionen in den Experience Replay Speicher geladen. Die zweite Klasse erstellt aus dem Experience Replay Speicher Minibatches und trainiert damit die Netze. Von jeder Klasse lassen sich beliebig viele Instanzen erstellen, die alle parallel arbeiten können.

5.2.4 Modifizierung der Netzarchitektur

In der vorgestellten Implementierung wird die Netzarchitektur aus Abschnitt 3.2.2 verwendet, die in Abbildung 5.1a zusammengefasst ist. Beide Netze erhalten die Zustandselemente als Eingabeschicht. Der Critic verarbeitet diese in seiner ersten verdeckten Schicht und fügt die Aktionswerte erst in der zweiten Schicht hinzu. Es besteht die Vermutung, dass die Neuronen aus der ersten Schicht des Actors und des Critics dieselben Eigenschaften trainieren. Daher ist eine mögliche Erweiterung durch Abbildung 5.1b gegeben. Vor das Netz des Actors und des Critics wird ein gemeinsames Vor-Netz geschaltet, dass diese Schicht ersetzen soll. Dieses Vor-Netz erhält als Eingabe den Zustand und reicht seine Ausgangswerte an die beiden anderen Netze weiter.



(a) Actor und Critic erhalten den Zustand als Eingabe.



(b) Vor dem Actor und Critic wird ein Vor-Netz geschaltet, welches den Zustand verarbeitet.

Abbildung 5.1: Mögliche Modifizierung der Netzarchitektur

5.2.5 Feinstellung der Parameter

Wie in der Evaluierung gezeigt, bietet der Algorithmus sehr viele Möglichkeiten zum Anpassen von Parameter. Die Feinstellung dieser ist für jedes Problem unterschiedlich. Für dieses Problem wurden gute Einstellungen gefunden. Jedoch lässt sich durch weitere Vergleiche von Parametereinstellungen sicherlich eine bessere Lösung finden. Da unter den Parametern Abhängigkeiten bestehen, die nicht offensichtlich sind, bieten die Feinstellung viel Platz zum Testen von Einstellungen. Insbesondere die Netzarchitektur bietet viele Möglichkeiten, wie zum Beispiel verschiedene Aktivierungsfunktionen, zur ausführlichen Evaluierung.

5.2.6 Reale Welt

Nach dem Training ist der Agent durch seine Strategie in der Lage einen Quadcopter zu stabilisieren. Diese Strategie kann anschließend auf einen echten Quadcopter in der realen Welt übertragen werden. Da das mathematische Modell der Flugdynamik auf realen Quadcoptern basiert, können gute Flugergebnisse erwartet werden. Jedoch muss das Training des Agenten zuvor in der Weise angepasst werden, dass ein begrenzter Flugraum definiert wird. Des Weiteren müssen die Maße, Größe und weitere Konstanten des Modells mit dem echten Quadcopter übereinstimmen. Da der Agent zur Strategieverfolgung die Zustandsinformationen benötigt, muss diese beim Versuchsaufbau durch Kameras und andere technische Hilfsmittel ermittelt werden.

Literaturverzeichnis

[1] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[2] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[3] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[4] Marcin Andrychowicz, Filip Wolski, Alex Ray, Jonas Schneider, Rachel Fong, Peter Welinder, Bob McGrew, Josh Tobin, Pieter Abbeel, and Wojciech Zaremba. Hindsight experience replay. *CoRR*, abs/1707.01495, 2017.

[5] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 566–574, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[6] J. Hwangbo, I. Sa, R. Siegwart, and M. Hutter. Control of a quadrotor with reinforcement learning. *IEEE Robotics and Automation Letters*, 2(4):2096–2103, Oct 2017.

[7] Anežka Chovancová, Tomáš Fico, Ľuboš Chovanec, and Peter Hubinsk. Mathematical modelling and parameter identification of quadrotor (a survey). *Procedia Engineering*, 96:172 – 181, 2014. Modelling of Mechanical and Mechatronic Systems.

[8] P. Wang, Z. Man, Z. Cao, J. Zheng, and Y. Zhao. Dynamics modelling and linear control of quadcopter. In *2016 International Conference on Advanced Mechatronic Systems (ICAMechS)*, pages 498–503, Nov 2016.

[9] Hakim Bouadi and M Tadjine. Nonlinear observer design and sliding mode control of four rotor helicopter. *Int Journal of Eng. and Applied Science*, 3, 01 2007.

[10] M Islam, M Okasha, and M M Idres. Dynamics and control of quadcopter using linear model predictive control approach. *IOP Conference Series: Materials Science and Engineering*, 270:012007, 12 2017.

[11] Matthew J. Hausknecht and Peter Stone. Deep reinforcement learning in parameterized action space. *CoRR*, abs/1511.04143, 2015.

[12] Peter J. Huber. Robust estimation of a location parameter. *Ann. Math. Statist.*, 35(1):73–101, 03 1964.

[13] Laëtitia Matignon, Guillaume J. Laurent, and Nadine Le Fort-Piat. Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning. In *Proceedings of the 16th International Conference on Artificial Neural Networks - Volume Part I*, ICANN'06, pages 840–849, Berlin, Heidelberg, 2006. Springer-Verlag.

[14] Matthias Plappert, Rein Houthooft, Prafulla Dhariwal, Szymon Sidor, Richard Y. Chen, Xi Chen, Tamim Asfour, Pieter Abbeel, and Marcin Andrychowicz. Parameter space noise for exploration. *CoRR*, abs/1706.01905, 2017.

[15] Shangtong Zhang and Richard S. Sutton. A deeper look at experience replay. *CoRR*, abs/1712.01275, 2017.

[16] Todd Hester, Matej Vecerik, Olivier Pietquin, Marc Lanctot, Tom Schaul, Bilal Piot, Andrew Sendonaris, Gabriel Dulac-Arnold, Ian Osband, John Agapiou, Joel Z. Leibo, and Audrunas Gruslys. Learning from demonstrations for real world reinforcement learning. *CoRR*, abs/1704.03732, 2017.

[17] Dan Horgan, John Quan, David Budden, Gabriel Barth-Maron, Matteo Hessel, Hado van Hasselt, and David Silver. Distributed prioritized experience replay. *CoRR*, abs/1803.00933, 2018.

Glossar

CPU central processing unit.

DDPG Deep Deterministic Policy Gradient.

DQN Deep Q-Network.

GPU graphics processing unit.

HER Hindsight Experience Replay.

MAE mean absolute error.

MDP Markov decision process - Markow-Entscheidungsproblem.

MSE mean squared error.

PD proportional-derivative.

PER Prioritized Experience Replay.

RL Reinforcement Learning.

TD Temporal Difference.

UAV unmanned aerial vehicle - unbemanntes Luftfahrzeug.

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

Gemäß der Allgemeinen Prüfungs- und Studienordnung ist zusammen mit der Abschlussarbeit eine schriftliche Erklärung abzugeben, in der der Studierende bestätigt, dass die Abschlussarbeit „– bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit [(§ 18 Abs. 1 APSO-TI-BM bzw. § 21 Abs. 1 APSO-INGI)] – ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt wurden. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich zu machen."

Quelle: § 16 Abs. 5 APSO-TI-BM bzw. § 15 Abs. 6 APSO-INGI

Erklärung zur selbstständigen Bearbeitung der Arbeit

Hiermit versichere ich,

Name: _____

Vorname: _____

dass ich die vorliegende Masterarbeit – bzw. bei einer Gruppenarbeit die entsprechend gekennzeichneten Teile der Arbeit – mit dem Thema:

Stabilisierung unkontrollierter Flugzustände mit Reinforcement Learning

ohne fremde Hilfe selbständig verfasst und nur die angegebenen Quellen und Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

_____	_____	_____
Ort	Datum	Unterschrift im Original