HAW HAMBURG

Bachelorarbeit

Amadou Bah

Kollisionsfreie lokale Roboternavigation mit Deep Reinforcement Learning in statischen und dynamischen Umgebungen

Fakultät Technik und Informatik
Department Informatik

Faculty of Engineering and Computer Science
Department Computer Science

Amadou Bah

Kollisionsfreie lokale Roboternavigation mit Deep Reinforcement Learning in statischen und dynamischen Umgebungen

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung
im Studiengang *Bachelor of Science Informatik Technischer Systeme*
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Prüfer: Prof. Dr. Stephan Pareigis
Zweitgutachter: Prof. Dr. Andreas Meisel

Eingereicht am: 13. Februar 2023

Amadou Bah

Thema der Arbeit

Kollisionsfreie lokale Roboternavigation mit Deep Reinforcement Learning in statischen und dynamischen Umgebungen

Stichworte

Deep Reinforcement Learning, Roboternavigation, Stable-Baselines3, Husky-Roboter, Pybullet

Kurzzusammenfassung

Im Rahmen dieser Bachelorarbeit wird ein Prototyp für den Einsatz von Deep Reinforcement Learning auf dem Husky-Roboter entwickelt. Mithilfe von Deep Reinforcement Learning sollte der Roboter kollisionsfrei und zeiteffizient in seiner Umgebung navigieren. Zu diesem Zweck werden verschiedene Reinforcement Learning Agenten trainiert. Für die Erkennung von den Hindernissen wird ein zweidimensionaler Laserscanner verwendet und dabei werden verschiedene Darstellungsformen von den Laser-Daten untersucht. Die Agenten werden sowohl in statischen als auch in dynamischen Umgebungen trainiert.
In den Experimenten konnte gezeigt werden, dass die Agenten trotz einiger Limitationen kollisionsfrei und zeiteffizient in statischen Umgebungen navigieren konnten. Im dynamischen Kontext konnte zudem gezeigt werden, dass einer der trainierten Agenten dynamische Hindernisse in seiner Bewegungsplanung berücksichtigen konnte.

Amadou Bah

Title of Thesis

Collision-free local robot navigation with deep reinforcement learning in static and dynamic environments

Keywords

Deep Reinforcement Learning, Robot navigation, Stable-Baselines3, Husky Robot, Pybullet

Abstract

In this bachelor thesis, a prototype for the use of Deep Reinforcement Learning on the Husky robot is developed. With the help of Deep Reinforcement Learning the robot should navigate collision-free and time-efficiently in its environment. For this purpose different reinforcement learning agents are trained. For the detection of obstacles a two-dimensional laser scanner is used and different representations of the laser data are investigated. The agents are trained in both static and dynamic environments. In the experiments, it was shown that the agents could navigate collision-free and time-efficiently in static environments despite some limitations. In the dynamic context it could be shown that one of the trained agents was able to consider dynamic obstacles in its motion planning.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungen

DDPG Deep Deterministic Policy Gradient.

Deep-RL Deep Reinforcement Learning.

HAW Hochschule für Angewandte Wissenschaften.

LSTM Long Short-Term Memory.

MDP Markov Decision Process.

POMDP Partially-Observable Markov Decision Process.

PPO Proximal Policy Optimization.

RL Reinforcement Learning.

ROS Robot Operating System.

SAC Soft Actor-Critic.

SB3 Stable-Baselines3.

TD Temporal-Difference Learning.

Symbolverzeichnis

m Einheit von Länge.

m/s Einheit von linearer Geschwindigkeit.

rad Winkeleinheit.

rad/s Einheit von Winkelgeschwindigkeit.

s Einheit der Zeit.

1 Einleitung

In diesem Kapitel wird die Thematik dieser Arbeit eingeführt. Hierzu werden im Abschnitt 1.1 die Motivation und die Problemstellung der Arbeit und im Abschnitt 1.2 die Zielsetzung vorgestellt. Abschließend wird im letzten Abschnitt der aktuelle Forschungsstand dargestellt.

1.1 Motivation und Problemstellung

Unter Kollisionsfreier Navigation versteht man die Fähigkeit eines autonomen Systems, sicher durch eine Umgebung zu navigieren, ohne dabei mit anderen Objekten zu kollidieren. Dabei sollte das System zeiteffizient navigieren und dies bedeutet, die bestmögliche Route zu einem Zielort in minimaler Zeit zu finden und zu verfolgen. Auf dem Markt befinden sich eine Reihe von autonomen Robotern für unterschiedliche Einsatzbereiche: Lieferroboter [1] [2] [40], Reinigungsroboter wie der Neo 2 [3], autonome mobile Industrie-Roboter wie der MiR100 [8], Feldroboter [39], etc.

Die COVID-19 Pandemie hat den Bedarf an mobilen Robotern erhöht. Während der COVID-19 Pandemie wurden autonome Roboter zur Eindämmung der Ausbreitung vom Virus eingesetzt [7]. Untersuchungen, die die COVID-19 Pandemie berücksichtigen erwarten ein starkes Wachstum im Markt für mobile Roboter in den nächsten 10 Jahren [14]. Berichte gehen von einem jährlichen Wachstum von 24% aus: von 19 Milliarden in 2018 auf 54 Milliarden Dollar in 2023 [6].

Mehr Roboter in menchlicher Umgebung erfordert auch deren Akzeptanz in der Gesellschaft und ein natürliches Verhalten der Roboter während der Navigation könnte deren Akzeptanz in der Gesellschaft erhöhen.

Viele erfolgreich eingesetzte Roboter verwenden traditonelle Planer für die Navigation. Traditonelle Ansätze verhindern Kollisionen durch passive Reaktionen und betrachten Hindernisse meistens als statische Objekte [11],[12],[13],[23].

Einige Planer wie der Dynamic-Window-Approach, der zum Einsatz in vielen Roboteranwendungen vorkommt, haben die manuelle Parametrierung als zusätzliches Problem. Manuelle Parametrierung kann herausfordernd sein [18],[42]. In [30] wurde zum Beispiel der Einfluss von Parametern auf die Bewegungsplanung untersucht.

Ansätze wie in [9] sind kurzsichtig, denn sie betrachten die Geschwindigkeit von Objekten in der Umgebung nur zum Planen des nächsten Navigationsschrittes und können daher die Dynamik der Umgebung nicht antizipieren. Diese Probleme können in hochdynamischen überfüllten Umgebungen zu unnatürlichen Verhalten der Roboter führen und in solchen Umgebungen sollten mobile Objekte und Menschen in der Umgebung dem Roboter bewusst sein [28].

Zur Lösung dieser Probleme hat sich Deep Reinforcement Learning (Deep-RL) als Alternative ergeben. Deep-RL Frameworks wurden für das Training von effizienten Policies, die die Interaktion und Kooperation zwischen Agenten implizit mitberücksichtigen [15],[16],[21],[25],[29].

1.2 Vorgehensweise und Zielsetzung

Im Rahmen des TIQ-Projekts an der HAW Hamburg soll sich der Husky-Roboter mit Hilfe von künstlicher Intelligenz in statischen und dynamischen Umgebungen zurecht finden. Die Arbeit verfolgt das konkrete Ziel, einen Prototyp für den Einsatz von Deep-RL für die lokale Navigation auf dem Husky-Roboter zu entwickeln.

Zum Erreichen des Zieles wird ein Policy-Network für die lokale Navigation unter Berücksichtigung von mobilen und statischen Objekten trainiert. Für die Erkennung, von Hindernissen in der Umgebung werden 2D-Laser-Daten verwendet. Es wird angenommen, dass die Übertragung vom trainierten Modell von der Simulation zur realen Welt mit 2D-Laser-Daten im Vergleich zu bildbasierender Objekterkennung leichter wird.

Für das Training wird eine realitätsnahe Simulationsumgebung vorbereitet und der Husky dient als Roboterplatform.

1.3 Struktur der Arbeit

Die Bachelorarbeit besteht aus sechs Kapiteln.

Im ersten Kapitel wird eine Einleitung in die Problemstellung gegeben, dann werden das Ziel der Arbeit und die Vorgehensweise beschrieben und schließlich werden verwandte Arbeiten vorgestellt.

Im zweiten Kapitel wird der Husky-Roboter vorgestellt, dabei wird auf seine Hardware- und Software-Bestandteile eingegangen.

Das dritte Kapitel führt die nötigen Grundlagen für die Arbeit ein. Dabei werden die Grundlagen von Reinforcement Learning erläutert und der Soft Actor-Critic (SAC) Algorithmus wird präsentiert.

Im vierten Kapitel wird die Simulationsumgebung vorgestellt, in der das Training und die Evaluation durchgeführt werden. Dabei werden das Roboter-Modell, die statischen und die dynamischen Hindernisse präsentiert.

Das fünfte Kapitel beschäftigt sich mit dem gesamten Experiment. Dabei wird zuerst das Navigationsproblem im Reinforcement Learning Kontext formuliert. Dann werden die Trainingsumgebungen und die Trainingsergebnisse vorgestellt. Der letzte Teil vom Kapitel widmet sich der Evaluation der trainierten Agenten und der Auswertung der Ergebnisse.

Im sechsten Kapitel werden abschließend die Ergebnisse der Arbeit zusammengefasst und Ansätze für Weiterentwicklungen werden vorgestellt.

1.4 Verwandte Arbeiten

Viele Arbeiten haben sich dafür interessiert, Deep-RL für die Roboternavigation einzusetzen.

In [41] haben die Autoren eine mit Deep-RL gelernte Policy mit einem Ultra-Wideband für die lokale Indoor-Navigation kombiniert und haben in deren Ergebnissen gezeigt,

dass die in der Simulation gelernte end-to-end Policy, die Lidar-Informationen auf lineare Geschwindigkeit und Winkelgeschwindigkeit abbildet, eine robuste und kosteneffiziente Lösung für die Navigation in realen Indoor-Umgebungen bieten kann.

Zur Vermeidung von der manuellen Parametereinstellung in traditionellen lokalen Planern haben die Autoren in [35] den Proximal Policy Optimization (PPO) Algorithmus verwendet und einen Deep-RL-Agenten trainiert, um kollisionsfreies und zielgerichtetes Verhalten zu erreichen. Deren Methode kombiniert ein neuronales Netz mit Pfadplanung auf einem 2D Grid-Map. Das neuronales Netz verwendet einen Ausschnitt vom Grid-Map als Information über die Hindernisse und gibt basierend darauf und auf den Informationen über die aktuelle Geschwindigkeit und die Zielposition die lineare Geschwindigkeit und Winkelgeschwindigkeit aus. Die haben deren Ansatz für das Robot Operating System (ROS) implementiert und sind zu den Ergebnissen gekommen, dass der trainierte Agent in schwierigen Navigationsaufagen eingesetzt werden kann und der Vergleich mit dem Dynamic-Window-Approach hat gezeigt, dass der trainierte Agent schneller am Ziel ankommt.

In [17] haben die Autoren eine zeiteffiziente Deep-RL Navigation-Policy entwickelt, die allgemeine soziale Normen berücksichtigt. Die vorgeschlagene Methode soll dem Roboter es befähigen, autonom in Umgebungen mit vielen Fußgängern zu navigieren. Dabei haben sie ein Multiagent-Szenario simuliert, wobei jeder Agent eine Policy lernt, die soziale Normen wie „Fahren auf der rechten Seite“ und „Überholen auf der linken Seite“ berücksichtigt.

Zur kollisionsfreien Navigation in Umgebungen mit statischen und dynamischen Hindernissen haben die Autoren in [19] mithilfe vom Soft Actor-Critic Algorithmus eine Navigationspolicy trainiert und die in ROS implementiert. Damit der Agent es lernt, die Bewegung von dynamischen Objekten in der Umgebung vorherzusagen, haben sie die Lidar-Daten der drei letzten Schritte im Beobachtungsraum mitberücksichtigt. Bei der Verwendung der Laser-Daten, der Geschwindigkeit vom Roboter und der Distanz zum Ziel für die Modellierung der Beobachtung haben sie festgestellt, das der Roboter in einigen Umgebungen nicht navigieren kann. Zur Lösung von diesem Problem haben sie einen Pfadplaner integriert. Obwohl der Agent besser als der Dynamic-Window-Approach Algorithmus war, haben sie festgestellt, dass der Roboter nicht gradlinig fahren konnte und bei der Vermeidung von Hindernissen nahm er ineffiziente Wege. Zudem konnte der Roboter nicht beschleunigen oder bremsen außer bei der Vermeidung von Hindernissen.

Viele weitere Arbeiten, wie [29] verwenden die letzten aufeinanderfolgenden Laser-Daten, um dem Agenten die Geschwindigkeit und Bewegungsrichtung von Objekten in der Umgebung zu geben und bestrafen den Agenten, wenn er einen gewissen Abstand zu dynamischen Objekten verletzt, wodurch der Agent es lernen kann, die Komfortzone von Menschen nicht zu betreten.

Andere Arbeiten wie [20] setzen Recurrent-Neural-Networks ein, um temporale Informationen aus aufeinanderfolgenden Beobachtungen zu extrahieren.

Eine weitere Herausforderung bei der Vermeidung von dynamischen Objekten ist, dass die Anzahl von Objekten in der Umgebung variieren kann. Ansätze die ein neuronales Netz mit fester Größe von der Eingabe Schicht verwenden und Informationen von anderen Objekten explizit im Beobachtungsraum berücksichtigen, haben das Problem, dass die nur eine feste Anzahl von Objekten in der Umgebung behandeln können. Die Autoren von [21] haben zur Lösung dieses Problems die Idee von Natural Language Processing eingesetzt und mithilfe eines Long Short-Term Memory (LSTM) Netzes konnten sie die Änderung der Anzahl von dynamischen Objekten in einem Vektor fester Länge kodieren, was es derem Algorithmus ermöglicht, eine beliebige Anzahl an Objekten zu behandeln und deren Ergebnisse zeigen eine bessere Performanz, wenn die Anzahl von dynamischen Objekten in der Umgebung steigt.

2 Der Husky A200

Der Husky ist ein von Clearpath Robotics entwickeltes unbemanntes Landfahrzeug. Der Roboter hat eine Länge von 0.99 m und eine Breite von 0.67 m. Der wird im Rahmen des TIQ-Projekts an der HAW Hamburg eingesetzt und ist mit zwei Intel Realsense D435 Kameras, einem Robosense 3D-laser-scanner RD-LiDAR-16, einem UM7 Orientierungssensor, einem U-Blox 7 GPS-Sensor und einem UR5 Arm für die Interaktion mit der Umgebung ausgestattet. Das Robot Operating System wird in Kombination mit C++ und Python verwendet, um den Roboter zu betreiben [31],[39].

Abbildung 2.1 zeigt den Roboter auf dem Campus der HAW:

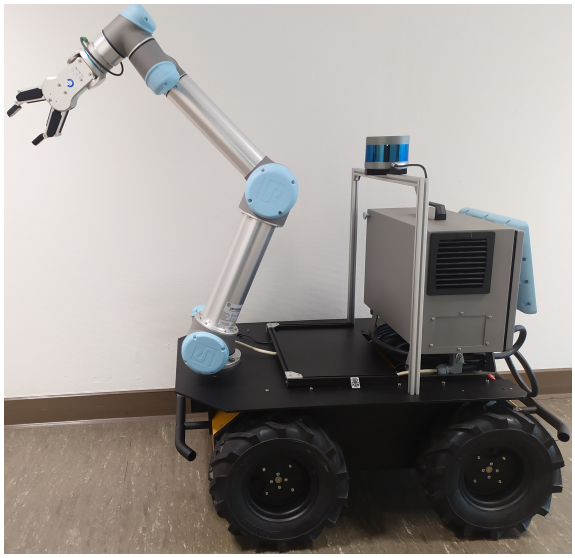


Abbildung 2.1: Der Husky-Roboter auf dem Campus der HAW [31].

3 Grundlagen

In diesem Kapitel werden die Grundlagen von Deep Reinforcement Learning erläutert und der Soft Actor-Critic Algorithmus wird präsentiert.

3.1 Reinforcement Learning

Die Grundidee in Reinforcement Learning ist Lernen durch Interaktion mit der Umgebung. Wie wir Menschen es lernen, zu laufen oder Auto zu fahren, lernt ein Reinforcement Learning Agent, indem er Handlungen in seiner Umgebung ausführt und die Konsequenzen dieser Handlungen beobachtet. Durch mehrere Interaktionen mit seiner Umgebung lernt der Agent selbständig zu jeder Situation die richtige Handlung. Der Agent wird beim Lernen nicht gesagt, welche Aktion in einer Situation auszuwählen ist, sondern findet er es durch ein Belohnungssystem selber heraus.

Zusätzlich zu dem Agenten und der Umgebung sind weitere Begriffe in einem Reinforcement Learning (RL) System zu unterscheiden:

- **Policy**: die Policy ist der Kern eines RL Agenten. Die bildet die vom Agenten beobachteten Zustände der Umgebung auf Aktionen ab und steuert damit das Verhalten vom Agenten.
- **Reward**: der Reward ist ein skalarer Wert und definiert das Ziel eines RL Problems. Der Agent bekommt von der Umgebung in jedem Schritt einen Wert, den Reward, der es bestimmt, wie gut oder schlecht er sich verhalten hat und das Ziel vom Agenten ist es, den gesamten Reward langfristig zu maximieren. Der Reward bildet zudem die Basis für die Veränderung der Policy: Wenn eine von der Policy ausgewählte Aktion in einem Zustand einen kleinen Reward erhält, kann sich die Policy so verändern, dass in die Zukunft eine andere Aktion in diesem Zustand ausgewählt wird, die einen größeren Reward liefert [37].

3.1.1 Markov Decision Process

Interaktion zwischen Agent und Umgebung

Zum Lernen durch Interaktionen werden Reinforcement Learning Systeme als Markov-Entscheidungsprozesse (MDP) modelliert. Der Agent interagiert kontinuierlich mit der Umgebung und bei jeder ausgeführten Aktion führt die Aktion den Agenten zu einem neuen Zustand der Umgebung. Abbildung 3.1 illustriert die Interaktion zwischen Agent und Umgebung.

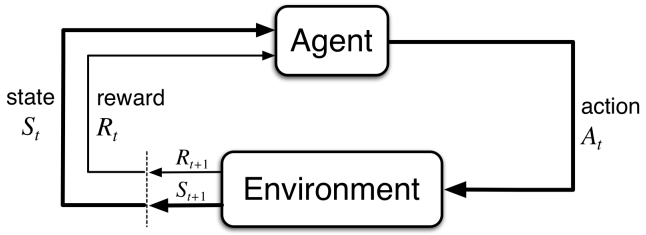


Abbildung 3.1: Der Agent und die Umgebung interagieren in diskreten Zeitschritten $t = 0, 1, 2, 3, \ldots$ und in jedem Schritt erhält der Agent einen Zustand $S_t \in \mathcal{S}$ von der Umgebung (Environment) und auf Basis von diesem Zustand S_t wählt er eine Aktion $A_t \in \mathcal{A}$ aus. Ein Zeitschritt später befindet sich der Agent im Zustand $S_{t+1} \in \mathcal{S}$ und er erhält einen Reward $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$. Der MDP und der Agent führen damit zu einer Sequenz von Zuständen, Aktionen und Rewards : $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \ldots$ [37].

Zu einem endlichen MDP gehören ein endlicher diskreter Zustandsraum $\mathcal{S}$ sowie eine Wahrscheinlichkeitsfunktion p, die die Übergangswahrscheinlichkeiten eines Zustands in einen anderen Zustand beschreibt. Bei der Interaktion zwischen dem Agenten und der Umgebung kann die Wahrscheinlichkeitsverteilung der zufälligen Variablen R_t und S_t durch die Formel 3.1 definiert werden: für $s' \in \mathcal{S}$ und $r \in \mathcal{R}$ definiert p die Wahrscheinlichkeit im Zustand s' zu sein und den Reward r zu erhalten, wenn sich der Agent im Zustand s befindet und die Aktion a ausführt.

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\}, \tag{3.1}$$

für alle $s', s \in \mathcal{S}, r \in \mathcal{R}$ und $a \in \mathcal{A}(s)$.

In einem MDP muss das System die **Markov-Eigenschaft** erfüllen. Das bedeutet, die zukünftigen Veränderungen an dem System in einem bestimmten Zustand nur von diesem

Zustand und nicht von der Vorgeschichte im Prozess abhängen darf [27][37]. Da p die Wahrscheinlichkeitsverteilung über jede Auswahl von s und a definiert, gilt zudem die Gleichung 3.2.

$$\sum_{s' \in \mathcal{S}} \sum_{r \in \mathcal{R}} p(s', r|s, a) = 1, \tag{3.2}$$

für alle $s \in \mathcal{S}, a \in \mathcal{A}(s)$.

Mithilfe von den vier Parametern von p lassen sich weitere Informationen von der Umgebung berechnen, wie die Zustandsübergangswahrscheinlichkeit (siehe Formel 3.3) oder der erwartete Reward für ein Zustand-Aktion Paar (siehe Formel 3.4) [37].

$$p(s'|s, a) \doteq \Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a). \tag{3.3}$$

$$r(s, a) \doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a). \tag{3.4}$$

Returns und Episoden

Wie schon erwähnt wurde ist das Ziel von einem RL Agenten den akummulierten Reward langfristig zu maximieren. Der Reward muss also so definiert werden, dass er es beschreibt, was der Agent erreichen soll und dass zukünftige Rewards auch mitberücksichtigt werden. Für eine Sequenz von Rewards $R_{t+1}, R_{t+2}, R_{t+3}, ...$ wird daher statt den unverzüglichen Reward der sogenannte *Return* beim Lernen verwendet. Der Return G_t zum Zeitpunkt t ist durch die Formel 3.5 definiert:

$$G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T, \tag{3.5}$$

wobei T der letzte Zeitschritt in der Interaktion ist.

Diese Formulierung vom Return in 3.5 macht jedoch nur Sinn, wenn es einen letzten Zeitschritt gibt und die Interaktion zwischen Agent und Umgebung irgendwann in einem Endzustand endet. Das ist der Fall zum Beispiel in einem Schachspiel. Das Spiel endet irgendwann und unabhängig vom Endzustand beginnt die nächste Episode vom Anfang an. Aufgaben, die sich in Episoden unterteilen lassen, werden *Episodische Aufgaben* genannt.

In *Kontinuierlichen Aufgaben* hingegen, wie zum Beispiel ein mobiler Roboter, der kontinuierlich mit der Umgebung interagiert, würde der letzte Zeitschritt unendlich sein und der Return könnte damit einfacher gegen unendlich liegen. Um dieses Problem zu umgehen, wird ein Diskontierungsfaktor γ in die Formel 3.5 eingeführt:

$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} = R_{t+1} + \gamma G_{t+1}, \tag{3.6}$$

mit $0 \leq \gamma \leq 1$.
Der Diskontierungsfaktor steht für die Voraussicht eines Agenten. Wenn gamma gleich 1 ist, dann entspricht es einem Agenten, der alle nachfolgenden Rewards exakt kennt. Ist gamma gleich 0, dann entspricht es einem kurzsichtigen Agenten und der Return G_t ist in diesem Fall der sofortige Reward. Üblicherweise liegt der Wert von gamma zwischen den Extremwerten, etwa bei 0.9 oder 0.99. Das Ziel vom Agenten ist nun den erwarteten diskontierten Return zu maximieren [27],[37].

State- und Action-Value

State- und *Action-Values* sind zwei Konzepte, die auf dem Return basieren. Sie bestimmen, wie gut es für den Agenten ist, sich in einem Zustand zu befinden beziehungsweise eine Aktion in einem Zustand auszuführen. Das bedeutet, welchen Return der Agent in einem Zustand oder bei einer Aktion erwarten kann. State- und Action-Values werden in Bezug auf eine Policy definiert, die das Verhalten vom Agenten steuert. Wenn ein Agent einer Policy π zum Zeitpunkt t folgt, dann definiert $\pi(a|s)$ die Wahrscheinlichkeit die Aktion a im Zustand s auszuwählen.
Die Formeln 3.7 und 3.8 definieren den State-Value vom Zustand s beziehungsweise den Action-Value von der Aktion q im Zustand s unter der Policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s \right], \forall s \in \mathcal{S}. \tag{3.7}$$

$$q_\pi(s,a) \doteq \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right] = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \middle| S_t = s, A_t = a \right], \forall s \in \mathcal{S}, a \in \mathcal{A}(s). \tag{3.8}$$

Der *State-Value* eines Endzustandes ist immer 0, wenn es einen gibt.

Die Formeln 3.7 und 3.8 lassen sich zudem in rekursiver Form ausdrücken:

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s\right] \\ &= \sum_a \pi(a \mid s) \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_a \pi(a \mid s) \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma v_\pi(s')\right], \forall s \in S \end{aligned} \tag{3.9}$$

$$\begin{aligned} q_\pi(s, a) &\doteq \mathbb{E}_\pi\left[G_t \mid S_t = s, A_t = a\right] \\ &= \mathbb{E}_\pi\left[R_{t+1} + \gamma G_{t+1} \mid S_t = s, A_t = a\right] \\ &= \sum_{s'} \sum_r p(s', r \mid s, a)\left[r + \gamma \mathbb{E}_\pi\left[G_{t+1} \mid S_{t+1} = s'\right]\right] \\ &= \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma \sum_a \pi(a \mid s') \mathbb{E}_\pi\left[G_{t+1} \mid A_{t+1} = a', S_{t+1} = s'\right]\right] \\ &= \sum_{s', r} p(s', r \mid s, a)\left[r + \gamma \sum_a \pi(a \mid s').q_\pi(s', a')\right] \end{aligned} \tag{3.10}$$

Diese rekursive Formen 3.9 und 3.10 sind die *Bellman-Gleichungen* für *State-* und *Action-Values*. Die Gleichungen drücken die Beziehung zwischen dem Wert eines Zustandes $v_\pi(s)$ oder eines Zustand-Aktion Paares $q_\pi(s, a)$ und dem vom nächsten Zustand $v_\pi(s')$ oder Zustand-Aktion Paar $q_\pi(s', a')$ aus.
Das Lösen von RL Problemen besteht nun darin, eine optimale Policy durch Lösen von den Bellman-Gleichungen zu finden. Wenn man jedoch die Bellman-Gleichung 3.9 für State-Values zum Beispiel betrachtet, bildet die ein System von Gleichungen mit einer Gleichung für jeden Zustand. Für n Zustände hat man dann n Gleichungen mit n Unbekannten. Bei einer großen Menge von Zuständen ist eine effiziente Lösung notwendig und die Dynamik der Umgebung p ist zudem nicht immer bekannt[37].

3.1.2 Das Exploration-Exploitation Problem

Bei der Interaktion zwischen dem Agenten und der Umgebung kommt eine weitere Schwierigkeit ins Spiel. Am Anfang vom Training ist die Näherung von den Action-Values Q nicht perfekt und das kann dazu führen, dass sich der Agent für manche Zustände festfährt und für diese Zustände kein anderes Verhalten ausprobiert. Eine zufällige Auswahl von Aktionen am Anfang vom Training ist daher sinnvoll. Andererseits wird zufälliges Verhalten ineffizient, wenn der Agent Fortschritte gemacht hat und in diesem Fall ist es sinnvoll bei der Entscheidung über die Handlung vom Agenten auf die Q-Näherung zurückzugreifen. Dieses Dilema ist in RL als *Exploration-Exploitation trade-off* bekannt.

Ein Verfahren, dass eine Mischung aus beiden Verhaltensweisen nutzt ist als *Epsilon-Greedy-Verfahren* bekannt. Dabei wird mithilfe eines Wahrscheinlichkeitsparameter ϵ zwischen Zufall und Anwendung der Action-Values gewechselt. Durch verschiedene Werte von ϵ kann der Anteil zufälliger Aktionen ausgewählt werden. Üblicherweise beginnt man mit $\epsilon = 1.0$ (100% zufällige Aktionen) und verringert den Wert allmählich auf 5 oder 2%. Das Verfahren ermöglicht es, am Anfang des Trainings die Umgebung zu erkunden und am Ende des Trainings bei einer guten Policy zu bleiben [27].

3.1.3 Monte Carlo Methode

Die Monte Carlo Methode ist eine Methode zur Schätzung von State- oder Action-Values und damit zur Bestimmung von optimalen Policies. Mit der Methode ist es nicht vorausgesetzt, die Umgebung vollständig zu kennen. Anders als bei den Bellman-Gleichungen (siehe Formel 3.9 und 3.10) ist die vollständige Wahrscheinlichkeitsverteilung für alle möglichen Transitionen nicht notwendig. Die Monte Carlo Methode benötigt nur eine Interaktion mit der Umgebung, Folgen von Zustand, Aktion und Reward. Die Methode basiert darauf, für jeden Zustand oder jedes Zustand-Aktion Paar den Mittelwert von über alle Episoden erhalteten Return-Werten zu ermitteln. Damit alle Return-Werte für die Berechenung vom diskontierten Return-Wert in einem Zustand zur Verfügung stehen wird die Methode für episodische Aufgaben definiert. Es wird angenommen, dass die Intreaktionen mit der Umgebung in Episoden unterteilt sind und dass alle Episoden unabhängig von den ausgewählten Aktionen irgendwann enden. State-Values werden erst am Ende der Episode berechnet und nicht bei jeder Interaktion, was auch bedeutet, dass die Policy erst am Ende von der Episode aktualisiert wird. Die Anzahl an Berechnungen für die Aktualisierung vom Wert eines Zustandes hängt nun nicht von der Größe vom

MDP, sondern von der Länge einer Episode ab. Die Formel 3.11 beschreibt eine effiziente Berechenung vom State-Value V duch Bilden des Mittelwerts über mehrere Episoden. Mit dieser Formel braucht man nicht alle vergangenen Return-Werte zu speichern, um den Mittelwert zu bilden [37]:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[G_t - V(S_t) \right], \tag{3.11}$$

wobei G_t der erhaltene Return nach dem Schritt t und $\alpha \in (0, 1]$ die Schrittgröße sind. S_t ist der Zustand, in dem sich der Agent zum Zeitpunkt t befindet. Die Herleitung von der Formel kann in [37] gefunden werden.

Approximierung von Policies und Action-Values mit Monte Carlo

Die optimale Policies lassen sich sowohl aus den State-Values als auch den Action-Values ableiten. Mit den State-Values kann dies erfolgen, indem die Policy in jedem Zustand die Aktion auswählt, die zum nächsten Zustand mit dem größten State-Value führt. Diese Ableitung aus den State-Values setzt aber voraus, dass das Modell der Umgebung bekannt ist. Mit einem Modell weiß man, zu welchem Zustand eine Aktion führt. Wenn das Modell hingegen unbekannt ist, ist es sinnvoller die Action-Values zu bestimmen und basierend darauf die optimale Policy zu herleiten. Das Ziel bei Monte-Carlo Methoden ist daher, die Action-Values einzuschätzen.

Die Approximierung erfolgt in mehreren Iterationen durch Abwechlung zwischen Evaluierung der Policy und ihre Aktualisierung. Abbildung 3.2 stellt den Approximierungsprozess dar:

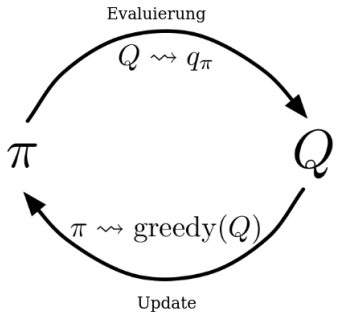


Abbildung 3.2: Approximierungsprozess von Policy und Action-Values. In jeder Iteration werden in der Evaluierungsphase die Action-Values für die Policy berechnet. Die Policy wird wiederum gemäß den Action-Values aus der Evaluierung aktualisiert und der Prozess setzt sich fort. Die zwei Schritte im Prozess arbeiten gegeneinander, aber bewegen Policy und Action-Values zu den optimalen Werten [37].

Für jede Action-Value Funktion Q lässt sich bei der Aktualisierung die neue Policy (die *greedy* Policy) berechnen, indem in jedem Zustand $s \in \mathcal{S}$ die Aktion mit dem größten Action-Value gewählt wird:

$$\pi(s) \doteq \arg\max_a Q(s, a) \tag{3.12}$$

Für eine Episode $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$ unter der Policy π lässt sich der Approximierungsprozess nach dem folgenden Algorithmus durchgeführen:

Algorithm 1 Evaluierung und Aktualisierung von Policies mit Monte-Carlo[37].

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s, a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$
$\alpha \in (0, 1]$
for each episode **do**
  Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
  Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  $G \leftarrow 0$
  for each step of episode, $t = T-1, T-2, \ldots, 0$ **do**
    $G \leftarrow \gamma G + R_{t+1}$
    if the pair S_t, A_t does not appear in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$ **then**
      Update $Q(S_t, A_t)$ using $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[G - Q(S_t, A_t)]$
      $\pi(S_t) \leftarrow \text{argmax}_a Q(S_t, a)$
    end if
  end for
end for

Wichtig zu bemerken ist die fehlende Exploration bei der Berechnung von der *greedy* Policy im Algorithmus 1 . Wenn man die Formel 3.12 betrachtet, könnte sie dazu führen, dass immer nur eine Aktion in einem gegebenen Zustand gewählt wird und dass weitere Aktionen nicht ausprobiert werden, die möglicherweise einen besseren Action-Value hätten. Statt das im Abschnitt 3.1.2 erwähnte *Epsilon-Greedy-Verfahren* wurde im Algorithmus 1 eine sogenannte *Exploring starts* Methode verwendet. Die Methode besteht darin, am Anfang von jeder Episode den Startzustand und die Aktion zufällig auszuwählen[37].

3.1.4 Temporal-Difference Learning

Ein Nachteil von der Schätzung von Action-Values mit der Monte-Carlo Methode besteht darin, dass der Agent während der Episode nicht lernt, denn in der Formel 3.11 liegt G_t erst am Ende der Episode vor. TD ermöglicht es dem Agenten, auch während der Episode zu lernen, indem frühere Schätzungen verwendet werden. TD konvergiert zu den optimalen Werten daher schneller als die Monte-Carlo Methode [37].

Policy-Evaluierung mit TD

Mit TD wird die Formel 3.11 zu:

$$V(S_t) \leftarrow V(S_t) + \alpha \left[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\right], \tag{3.13}$$

Mit der Formel 3.13 kann der Agent nach einer Transition vom Zustand S_t zu S_{t+1} sofort $V(S_t)$ mithilfe von der aktuellen Schätzung von $V(S_{t+1})$ aktualisieren. Die Verwendung von der aktuellen Schätzung wird *Boostrapping* genannt. Diese TD Methode wird TD(0) oder *one-step* TD gennant. Der Teil $[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)]$ aus der Formel wird TD-Error genannt und durch δ notiert. Der TD-Error berechnet den Fehler bei der Schätzung von $V(S_t)$.

Der Algorithmus 2 beschreibt den Prozess bei der Schätzung von State-Values mit TD(0):

Algorithm 2 Algorithmus zur Evaluierung von Policies mit TD(0)[37].

Input: the policy π to be evaluated
Algorithm parameter: step size $\alpha \in (0, 1]$
Initialize: $V(s)$ for all $s \in \mathcal{S}$ arbitrarily except that $V(terminal) = 0$
for each episode **do**
  Initialize S
  for each non terminal step of the episode **do**
    $A \leftarrow$ action given by π for S
    Take action A, observe R, S'
    $V(S) \leftarrow V(S) + \alpha[R + \gamma V(S') - V(S)]$
    $S \leftarrow S'$
  end for
end for

Q-Learning

Q-Learning ist eine Methode zur Evaluierung von Policies basierend auf Action-Values. Ihre Entwicklung war einer der ersten Durchbrüche in Reinforcement Learning. Q-Learning verwendet TD und die Berechnung von Action-Values erfolgt mit der folgenden Formel:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \right] \tag{3.14}$$

Für ein Zustand-Aktion Paar zum Schritt t berechnet die Formel 3.14 den durchschnittlichen Return, den der Agent für dieses Paar erhalten hat. Beim *Boostrapping* verwendet Q-Learning $\max_a Q(S_{t+1}, a)$ und approximiert damit die optimale Action-Value Funktion. Mit Q-Learning ist die gelernte Action-Value Funktion Q unabhängig von der Policy, die das Verhalten vom Agenten bestimmt. Q-Learning wird daher in dem Sinne *off-policy* genannt, dass die gelernte Policy und die den Agenten steuernde Policy unterschiedlich sein können [37].

3.1.5 Policy-Evaluierung mit neuronalen Netzen

Eine Möglichkeit die State- oder Action-Values mit den Bellman-Gleichungen oder mit Q-Learning zu berechnen ist eine Tabelle zu nutzen, wo die Zustände beziehungsweise die Zustand-Aktion Paare gespeichert werden. Diese Methode ist als *Tabular-Learning* bekannt. Es ist jedoch schwierig alle Zustände in einer Tabelle zu speichern, wenn die Anzahl der beobachtbaren Zustände zu groß ist. Außerdem, bei der Modellierung von

Zuständen durch Bilder, wie bei den Atari-Spielen, macht es keinen großen Unterschied, wenn sich 2 Bilder nur durch ein Pixel unterscheiden und in dem Fall ist es effizienter die Bilder als ein einziger Zustand zu betrachten. Zur Lösung von diesen Problemen können tiefe neuronale Netze zur Approximierung von der Action-Value Funktion q beziehungsweise State-Value Funktion v eingesetzt werden, die zu einem Zustand-Aktion Paar oder einem Zustand einen Wert zuordnet [27].

Die Funktionen $v_\pi(s)$ und $q_\pi(s,a)$ können mithilfe einer parametrisierten Funktion $\hat{v}(s,\mathbf{w})$ bzw. $\hat{q}(s,a,\mathbf{w})$ approximiert werden, wobei $\mathbf{w} \in \mathbb{R}^d$ die Gewichte des neuronalen Netzes representiert.

Bei der Approximierung von State-Values können die Gewichte nach einem Zeitpunkt t mit Stochastic-Gradient-Descend wie folgt verändert werden:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] . \nabla \hat{v}(s_t, \mathbf{w}_t) \tag{3.15}$$

Analog zur Gleichung 3.15 können Action-Values auf derselben Weise bestimmt werden:

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \left[q_\pi(s_t, a_t) - \hat{q}(s_t, a_t, \mathbf{w}_t) \right] . \nabla \hat{q}(s_t, a_t, \mathbf{w}_t) \tag{3.16}$$

Die Herleitung von den Formeln 3.15 und 3.16 sind in [37] zu finden. Der Algorithmus 3 präsentiert den Gradienten Algorithmus von TD für die Approximierung von $\hat{v} \approx v_\pi$.

Algorithm 3 Semi Gradient TD(0) for Estimating $\hat{v} \approx v_\pi$ [37]

Require: the policy π to be evaluated
Require: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$; $\hat{v}(S_{terminal}, .) = 0$
  Choose step-size α: $\alpha > 0$
  Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily
  for *episode* $= 1, 2, \ldots$ **do**
    Initialize S
    for *episode_step* $= 0, 1, \ldots, T-1$ **do**
      Choose A $\sim \pi(.|S)$
      Take action A, observe R, S'
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w}) \right] \nabla \hat{v}(S, \mathbf{w})$
      $S \leftarrow S'$
    end for
  end for

3.2 Deep-Reinforcement Learning

3.2.1 Policy-Gradient Methode

Bei allen bisher erwähnten Methoden ist die Bestimmung von der Action-Value Funktion die Zentrale Aufgabe und von den Action-Values wurde die *greedy*-Policy abgeleitet. Ohne die Action-Values würde die Policy nicht existieren. Bei großen oder kontinuierlichen Aktionsräumen, wie der linearen Geschwindigkeit eines Roboters, die im Intervall [-1;1] m/s liegt, ist die Berechnung von Action-Values schwierig.

Mit der Policy-Gradient Methode lernt der Agent eine parametrisierte Policy, die Wahrscheinlichkeit $\pi(a \mid s, \theta)$ zum Zeitpunkt t eine Aktion a im Zustand s unter dem Parameter $\theta \in \mathbb{R}^{d'}$ auszuwählen:

$$\pi(a \mid s, \theta) = \Pr\{A_t = a | S_t = s, \theta_t = \theta\} \tag{3.17}$$

Das Ziel mit der Policy-Gradient Methode ist die Performanz vom Agenten $J(\theta)$ mithilfe von Stochastic-Gradient-Ascend zu maximieren:

$$\theta_{t+1} \doteq \theta_t + \alpha \widehat{\nabla J(\theta_t)}, \tag{3.18}$$

wobei α die Schrittweite festlegt und $\widehat{\nabla J(\theta_t)}$ eine stochastische Schätzung ist, deren Erwartungswert den Gradienten von $J(\theta)$ approximiert. Nach dem *Policy-Gradient-Theorem* ist es bewiesen, dass sich $\nabla J(\theta)$ wie folgt ausdrücken lässt:

$$\nabla J(\theta) = \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t \mid S_t, \theta)}{\pi(A_t \mid S_t, \theta)} \right] \tag{3.19}$$

$\nabla J(\theta)$ is der Gradient der Log-Wahrscheinlichkeit der ausgeführten Aktion und die Skalierung des gradienten ist proportional zum Wert der ausgeführten Aktion. Das hat zur Folge, dass die Wahrscheinlichkeit von Aktionen erhöht wird, die den größten Return eingebracht haben und die Wahrscheinlichkeit von Aktionen mit kleinerem Return verringert wird [27],[37].

Die Gewichte der parametrisierten Policy können somit wie folgt aktualisiert werden:

$$\begin{aligned}\theta_{t+1} &\doteq \theta_t + \alpha G_t \frac{\nabla \pi(A_t \mid S_t, \theta_t)}{\pi(A_t \mid S_t, \theta_t)} \\ &= \theta_t + \alpha G_t \nabla \ln \pi(A_t \mid S_t, \theta_t)\end{aligned} \tag{3.20}$$

Bei der Aktualisierung der Gewichte in 3.20 wird der Return zum Zeitpunkt t verwendet, der alle zukünftigen Rewards bis Ende der Episode enthält. Es handelt sich daher um eine Monte-Carlo Methode und ein Algorithmus, der diese Methode verwendet ist in Reinforcement Learning als REINFORCE Algorithmus bekannt.

Die Herleitung von $\nabla J(\theta)$ kann in [37] gefunden werden.

3.2.2 Actor-Critic Methoden

Anders als REINFORCE Methoden sind Actor-Critic Methoden TD Methoden. Bei Actor-Critic Methoden lernt der Agent eine Approximation von sowohl Policy als auch von Value Funktion (meistens Action-Value Funktion). Der *Actor* bezeichnet die gelernte policy und der *Critic* bezeichnet die gelernte Value Funktion, denn er kritisiert die vom Actor ausgewählten Aktionen. Der Actor und Critic Teil können beide neuronale Netze sein. Zum Trainieren des Critic-Teils kann jeder Algorithmus zum Lernen von State- oder Action-Values verwendet werden, Semi-Gradient TD(0) (siehe Algorithmus 3) zum Beispiel. Die parametrisierte Policy kann mit der eben beschriebenen Policy-Gradient Methode aktualisiert werden.

Abbildung 3.3 stellt die Actor-Critic Architektur dar und der Algorithmus 4 beschreibt den *One-Step Actor-Critic* Algorithmus für episodische Aufgaben.

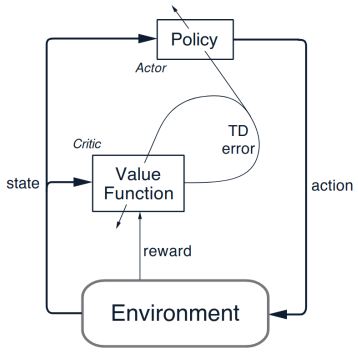


Abbildung 3.3: Die Actor-Critic Architektur [36]. Der TD-Error steuert den Lernprozess für sowohl Actor als auch Critic.

Algorithm 4 One-step Actor-Critic (episodic) for Estimating $\pi_\theta \approx \pi_*$ [37]

Require: a differentiable policy parametrization $\pi(a \mid s, \theta)$
Require: a differentiable state-value function parametrization $\hat{v}(s, w)$
  Choose $\alpha^\theta > 0$, $\alpha^w > 0$
  Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ and state-value weights $w \in \mathbb{R}^d$
  for each episode **do**
    Initialize S (first state of episode)
    $I \leftarrow 1$
    while S is not terminal (for each time step t) **do**
      $A_t \sim \pi(. \mid S, \theta_t)$
      Take action A_t, observe S_{t+1}, R_{t+1}
      $\delta \leftarrow R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$ (if S_{t+1} is terminal, then $\hat{v}(S_{t+1}, w_t) \doteq 0$)
      $w_{t+1} \leftarrow w_t + \alpha^w \delta \nabla \hat{v}(S_t, w_t)$
      $\theta_{t+1} \leftarrow \theta_t + \alpha^\theta I \delta \nabla \ln \pi(A_t, \mid S_t, \theta_t)$
      $I \leftarrow \gamma I$
      $S \leftarrow S_{t+1}$
    end while
  end for

3.2.3 Policy-Parametrisierung

Bei kleinen diskreten Aktionsräumen lässt sich die Wahrscheinlichkeit $\pi(a \mid s, \theta)$ mit der Softmax-Funktion berechnen. Die Softmax-Funktion ist eine Aktivierungsfunktion und wird bei einer Mehrklassen-Klassifikation auf der letzten Schicht eines neuronalen Netzes verwendet, wobei die Schicht ein Ausgangsneuron für jede Klasse hat. Für k Ausgangsneuronen mit den Werten $a_1, \ldots, a_k$ für die jeweiligen k Aktionen ist die Softmax definiert durch:

$$\pi(a_i \mid s, \theta) \doteq \frac{e^{a_i}}{\sum_{j=1}^{k} e^{a_j}}, \tag{3.21}$$

wobei $\pi(a_i \mid s, \theta)$ die Wahrscheinlichkeit gibt, dass eine Aktion i im Zustand s unter den Parametern θ ausgewählt wird. Der Nenner in der Funktion führt dazu, dass die Summe der Wahrscheinlichkeiten von allen Aktionen 1 ergibt [37].

Für kontinuierliche Aktionsräume mit unendlichen Aktionen ist es nicht praktikabel die Wahrscheinlichkeit für jede Aktion zu berechnen. Daher lernt der Agent stochastische Parameter einer Wahrscheinlichkeitsverteilung statt die Wahrscheinlichkeit für jede Aktion. Der Agent kann die Parameter einer Gauß-Verteilung mithilfe eines neuronalen Netzes zum Beispiel lernen und für einen gegebenen Zustand kann der Agent mithilfe dieser

Parameter eine Aktion aus der Verteilung auswählen. Die folgende Formel definiert die Wahrscheinlichkeitsdichtefunktion einer Gauß-Verteilung:

$$p(x) \doteq \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right), \tag{3.22}$$

wobei μ der Erwartungswert, σ die Standardabweichung und $p(x)$ die Wahrscheinlichkeitsdichte bei x sind. Mit der Gauß-Verteilung lässt sich durch Berechnung der Fläche unter der Funktion in einem Intervall X die Wahrscheinlichkeit berechnen, das ein Parameter x in diesem Intervall X fällt.

Die Parametrisierte Policy kann dann wie folgt definiert werden:

$$\pi\left(a \mid s, \theta\right) \doteq \frac{1}{\sigma\left(s,\theta\right)\sqrt{2\pi}} \exp\left(-\frac{\left(a-\mu\left(s,\theta\right)\right)^2}{2\sigma\left(s,\theta\right)^2}\right), \tag{3.23}$$

mit: $\mu : \mathcal{S} \times \mathbb{R}^{d'} \to \mathbb{R}$ und $\sigma : \mathcal{S} \times \mathbb{R}^{d'} \to \mathbb{R}^+$

Die Policy Parameter θ können nun in 2 geteilt werden: $\theta = [\theta_\mu, \theta_\sigma]^T$. θ_μ und θ_σ werden für die Approximierung vom Erwartungswert und von der Standardabweichung verwendet. Am Beispiel eines neuronalen Netzes können sich zum Beispiel μ und σ bis auf die letzte Schicht die selben Gewichte teilen.

3.2.4 Soft Actor-Critic

Traditionelle Actor-Critic Methoden haben das Problem, dass der Agent auch bei kleinen Aufgaben lange mit der Umgebung interagieren muss, damit er etwas lernt. Die Hyperparameter haben außerdem einen großen Einfluss auf das Trainingsergebnis und müssen sorgfältig ausgewählt werden, um gute Ergebnisse zu erreichen. Diese Probleme limitieren ihre Anwendbarkeit in der realen Welt. Bei *On-Policy* Learning-Algorithmen ist das Sammeln von Trainingsdaten ineffizient. PPO zum Beispiel braucht für jeden Gradienten-Schritt eine neue Sammlung von Trainingsdaten. Bei komplexen Aufgaben, in denen der Agent viele Gradienten-Schritte zum lernen braucht und viele Daten per Gradienten-Schritt zu sammeln sind, erhöht dies die Komplexität drastisch. *Off-Policy* Algorithmen hingegen ermöglichen alte Interaktionen mit der Umgebung wieder zu verwenden, indem die Interaktionen in einem sogenannten *Replay-Buffer* gespeichert werden. Die Verwendung von Replay-Buffer mit der Policy-Gradient Methode wie der Deep

Deterministic Policy Gradient (DDPG) Algorithmus es implementiert, bereitet jedoch eine Herausforderung bei der Konvergenz. Der Algorithmus ist sehr sensibel zu seinen Hyperparametern, was seine Verwendung schwierig macht [24].

Der Soft Actor-Critic ist ein auf dem *Maximum Entropy Reinforcement Learning Framework* basierender *off-Policy Actor-Critic* Deep-RL Algorithmus. Die *Maximum Entropy* Formulierung fügt zum Reward einen auf einem Entropy-Term $\mathcal{H}$ basierenden Bonus hinzu.

Die Entropy $\mathcal{H}$ is ein Maß, dass die Stochastizität einer Policy π reflektiert und steigt mit der Unsicherheit von der Policy. Die Entropy-Funktion ist definiert durch:

$$\mathcal{H}(\pi(. \mid s_t)) = \mathbb{E}_{a_t \sim \pi} \left[-\log \pi(a_t \mid s_t) \right] \tag{3.24}$$

Die Objective-Function $J(\pi)$ im Soft Actor-Critic ist wie folgt definiert und der Agent versucht $J(\pi)$ zu maximieren:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{\pi} \left[r(s_t, a_t) + \alpha H(\pi(. \mid s_t)) \right], \tag{3.25}$$

mit T, der letzte Zeitschritt und α ist ein Gewichtungsfaktor, der die Wichtigkeit von der Stochastizität im Vergleich zum originalen reward bestimmt [24].

Der Einsatz von diesem Entropy-Term in die Objective-Function bringt durch die Stochastizität einige Vorteile mit sich [22]:

- **Robustheit**: bei einer deterministischen Policy, einer Policy, die in einem Zustand immer die selbe Aktion ausführt, kann es zu Problemen kommen, wenn die Daten aus der Umgebung verauscht sind. Der Agent wird aus den verauschten Daten lernen, die die Umgebung nicht wirklich räpresentieren, für jeden Zustand die selbe Aktion auszuführen. Da besteht es die Gefahr, dass der Agent nicht die optimalste Lösung lernt. Wenn die Aktion aber aus einer Distribution ausgewählt wird, ist die Wahrscheinlichkeit größer, dass der Agent die optimalste Lösung findet und besser generalisiert, auch wenn die Umgebungsdaten verauscht sind.

- **Bessere Exploration**: mit einer stochastischen Policy ist keine explizite Explorationsstrategie wie die Epsilon-greedy Strategie notwendig. Durch die stochastizität hat man eine automatische Exploration.
- **Transfer in die reale Welt**: durch die Robustheit generalisiert die policy besser und die kann leichter in die reale Welt transferiert werden.

Der Algorithmus 5 beschreibt den Soft Actor-Critic Algorithmus und die verwendeten mathematischen Funktionen werden im folgenden erläutert [24].

Algorithm 5 Soft Actor-Critic

Initialize parameter vectors $\psi, \bar{\psi}, \theta, \phi$.
for each iteration **do**
  for each environment step **do**
    $a_t \sim \pi_\phi(a_t \mid s_t)$ (sample an action according to π_ϕ)
    $s_{t+1} \sim p(s_{t+1}|s_t, a_t)$ (observe next state)
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(s_t, a_t, r(s_t, a_t), s_{t+1})\}$ (add transition to the replay buffer)
  end for
  for each gradient step **do**
    $\psi \leftarrow \psi - \lambda_V \hat{\nabla}_\psi J_V(\psi)$ (update weights of soft V-function)
    $\theta_i \leftarrow \theta_i - \lambda_Q \hat{\nabla}_{\theta_i} J_Q(\theta_i)$ for $i \in \{1, 2\}$ (update weights of soft Q-functions)
    $\phi \leftarrow \phi - \lambda_\pi \hat{\nabla}_\phi J_\pi(\phi)$ (update policy weights)
    $\bar{\psi} \leftarrow \tau\psi + (1 - \tau)\bar{\psi}$ (update weights of target soft V-function)
  end for
end for

Der Algorithmus verwendet Stochastic-Gradient-Descend zur Aktualisierung der Gewichte der Soft State-Value Funktion (V_ψ), der Soft Action-Value Funktion(Q_θ) und der Soft Policy (π_ϕ).

Parametrisierte Soft State-Value Funktion (Soft V-Funktion)

Die Soft V-Funktion 3.26 ist eine Erweiterung von der bellmansche v Funktion auf das Entropy-Term und der Agent wird trainiert, um $J_V(\psi)$ zu minimieren:

$$V(s_t) = \mathbb{E}_{a_t \sim \pi} \left[Q(s_t, a_t) - \log \pi(a_t \mid s_t) \right] \tag{3.26}$$

$$J_V(\psi) = \mathbb{E}_{s_t \sim \mathcal{D}} \left[\frac{1}{2} \left(V_\psi(s_t) - \mathbb{E}_{a_t \sim \pi_\phi} \left[Q_\theta(s_t, a_t) - \log \pi_\phi(a_t \mid s_t) \right] \right)^2 \right], \tag{3.27}$$

wobei $\mathcal{D}$ ein Replay-Buffer ist, der Informationen über alte Intraktionen zwischen dem Agenten und der Umgebung speichert. Die Aktionen werden gemäß der aktuellen Policy und nicht aus dem Replay-Buffer abgerufen.

Parametrisierte Soft Action-Value Funktion (Soft Q-Funktion)

Die Parameter der Soft Q-Funktion können durch Minimierung von $J_Q(\theta)$ gelernt werden. Im vorgestellten Algorithmus, werden 2 unabhängige parametisierte Soft Q-Funktionen eingesetzt. Das Minimum zwischen den zwei Soft Q-Funktionen wird in 3.27 zur Vermeidung von der Überschätzung der Policy verwendet.

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t) + \hat{Q}_(s_t,a_t)\right)^2\right], \tag{3.28}$$

mit $\hat{Q}_(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}\left[V_{\bar{\psi}}(s_{t+1})\right]$

$\hat{Q}_(s_t,a_t)$ verwendet bei seiner Berechnung ein sogenanntes *target value network* $V_{\bar{\psi}}$, wobei $\bar{\psi}$ mithilfe des exponentiell gleitenden Durchschnitts von den Gewichten ψ berechnet wird. Alternativ, können die Gewichte $\bar{\psi}$ so aktualisiert werden, dass die ψ periodisch entsprechen.

Parametrisierte Soft Policy

In traditionellen Actor-Critic Methoden (siehe Abschnitt 3.2.2) kann kein Replay-Buffer verwendet werden, denn die Aktualisierung der Policy-Parameter von der aktuellen Policy abhängt.

Beim Soft Actor-Critic Algorithmus hängt die Aktualisierung der Policy-Parameter nicht von der aktuellen Policy sondern vom Softmax der aktuellen Soft Q-Funktion. Die Verwendung vom Replay-Buffer ist daher ohne Einsatz von einem *Target Policy-Network* (wie bei DDPG) möglich.

Bei der Aktualisierung der Policy-Parameter wird $J_\pi(\phi)$ minimiert:

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}}\left[\mathrm{D}_{\mathrm{KL}}\left(\pi_\phi(.\mid s_t)\middle\|\frac{\exp\left(Q_\theta(s_t,.)\right)}{Z_\theta(s_t)}\right)\right] \tag{3.29}$$

$J_\pi(\phi)$ ist die Kullback-Leibler-Divergenz (KL-Divergenz) zwischen der gesuchten Policy $\pi_\phi(. \mid s_t)$ und dem Softmax der Soft Q-Funktion $\frac{\exp(Q_\theta(s_t,.))}{Z_\theta(s_t)}$. $Z_\theta(s_t)$ ist der Normalisierungsfaktor in der Softmax-Funktion (siehe Formel 3.21). Die Idee bei der Minimierung der KL-Divergenz besteht darin, die Policy-Parameter so zu aktualisieren, dass die gelernte Policy zum Softmax der Soft Q-Funktion konvergiert. Es ist bewiesen, dass die resultierende Policy eine optimale Policy ist und der Beweis ist in [24] zu finden.

4 Simulation

In diesem Kapitel wird die Simulationsumgebung vorgestellt, in der das Training und die Evaluation stattfinden. Zuerst wird das Robter-Modell präsentiert und danach werden die statischen und dynamischen Hindernisse vorgestellt.

Für die Simulation wurde zuerst Gazebo in Verbindung mit ROS gewählt, aber beim Zurücksetzen der Simulation am Ende einer Episode im Training konnten einige Probleme erkannt werden. Einige ROS-Nodes mussten neu gestartet werden, was das Training verlangsamt. Statt Gazebo in Verbindung mit ROS wurde Pybullet für die Simulation verwendet. Pybullet ist ein einfaches und gut dokumentiertes Python-Modul für die Simulation von Robotern und für Machine-Learning Aufgaben mit dem Fokus auf *sim-to-real transfer*. Pybullet unterstützt außerdem URDF, SDF und weitere Dateiformate [5].

4.1 Roboter-Modell

Der Husky wird in Pybullet basierend auf seinem URDF-Modell aus dem Husky Github-Repository modelliert [38]. Der Roboter wird mit einem zweidimensionalen Laserscanner ausgestattet, der sich in seiner Mitte befindet. Abbildung 4.1 präsentiert das Husky-Modell in einer Pybullet-Umgebung.

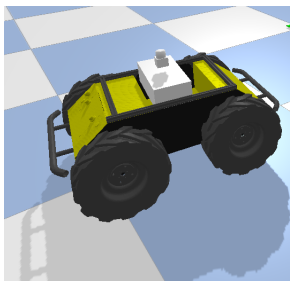


Abbildung 4.1: Das Roboter-Modell in Pybullet

Für die Steuerung vom Roboter in Pybullet wird die *Differential-Drive* Formel 4.1 verwendet, wobei v und ω die gewünschten lineare Geschwindigkeit und Winkelgeschwindigkeit vom Roboter sind [4]. ω_L und ω_R repräsentieren die Winkelgeschwindigkeit, mit der das linke und das rechte Rad kontrolliert wird. r ist der Radius von den Rädern und b ist die Breite vom Roboter.

$$\begin{cases} \omega_L = \frac{v - \omega.b/2}{r} \\ \omega_R = \frac{v + \omega.b/2}{r} \end{cases} \tag{4.1}$$

Die Formel 4.1 ist jedoch für *Differential-Drive* Systeme aufgestellt, wobei 2 Räder kontrolliert werden, um den Roboter zu steuern. Da aber der Husky 4 Räder hat, muss $b/2$ skaliert werden, damit eine vorgegebene Winkelgeschwindigkeit ω vom Roboter erreicht wird. Dafür wurde matematisch der Skalierungsfaktor s ermittelt und die endgültige Formel für die Steuerung der Räder ist wie folgt definiert:

$$\begin{cases} \omega_L = \frac{v - \omega.(s.b/2)}{r} \\ \omega_R = \frac{v + \omega.(s.b/2)}{r} \end{cases} \tag{4.2}$$

Der Wert vom Skalierungsfaktor s ist in Tabelle A.1 zu finden.

4.2 Statische Hindernisse

Die statischen Hindernisse bestehen aus 4 Wänden, die das Trainingsfeld umgeben und den drei Arten von Objekten in Abbildung 4.2. Die Hindernisse wurden in Blender modelliert.

(a) (b) (c)

Abbildung 4.2: 2D-Darstellung der verwendeten Objekte bei der Modellierung von Hindernissen. 4.2a ist ein Cylinder mit 0.3 m Durchmesser und einer Höhe von 1 m. 4.2b hat eine Länge von 1 m, eine Breite von 0.3 m und eine Höhe von 1 m. 4.2c hat eine Länge von 0.3 m, eine Breite von 0.3 m und eine Höhe von 1 m.

4.3 Dynamische Hindernisse

Die Dynamik in der Umgebung wird modelliert, indem Fußgänger simuliert werden, die sich zwischen einem Startpunkt und einem Zielpunkt hin und her bewegen. Für die Simulation von Fußgängern wird *pedsim_ros* verwendet. Pedsim_ros enthält ROS-Packages für die 2D-Simulation von Fußgängern basierend auf dem Social-Force-Model [26]. Die Pakete werden für Experimente in der Roboternavigation in überfüllten Umgebungen verwendet und wurden zum Teil im Rahmen des SPENCER Projekts entwickelt [32][10]. Für die Integration mit Pybullet wird auf das ROS-Topic abonniert, auf dem Pedsim_ros die Position der Fußgänger publiziert. Die Position der assozierten Fußgänger in der Pybullet-Umgebung werden entsprechend den publizierten Positionen in jedem Simulationsschritt aktualisiert. Pedsim_ros erwartet die Position vom Roboter in Form von *ROS-Transform-Message*. Daher wird die Position vom Roboter in der Pybullet-Umgebung nach jedem Simulationsschritt als ROS-Transform-Message publiziert. Der Cylinder in Abbildung 4.2a wird für die Darstellung von Fußgängern in der Pybullet-Umgebung verwendet.

5 Experiment

In diesem Kapitel wird das gesamte Experiment vorgestellt. Zuerst wird das Navigationsproblem im Reinforcement Learning Kontext formuliert, dann werden die verwendeten neuronalen Netze, die Trainingsumgebungen, die Ergebnisse vom Training und von der Evaluation präsentiert.

5.1 Formulierung vom Navigationsproblem

Der Agent ist dafür zuständig, von einem Startpunkt zu einem Zielpunkt innerhalb eines 7 m $\times$ 7 m Feldes zu navigieren. Dabei sollte er sowohl statische als auch dynamische Hindernisse vermeiden. Bei dynamischen Umgebungen sollte der Agent das Bewegungsmodell von den dynamischen Hindernissen (Fußgängern) mitberücksichtigen. Da er dieses Bewegungsmodell nicht direkt beobachten kann wird das Navigationsproblem als Partially-Observable Markov Decision Process (POMDP) formuliert. Ein POMDP ist ein MDP, in dem der Agent die zugrunde liegenden Zustände im Modell nicht direkt beobachten kann.

Das Problem kann daher als Tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \Omega, \mathcal{O})$ beschrieben werden. $s \in \mathcal{S}$ repräsentiert hier den Zustand von der Umgebung. Dieser Zustand verändert sich abhängig von der vom Agenten ausgeführten Aktion $a \in \mathcal{A}$ und vom Zustandsübergangsmodell $\mathcal{P}(s^{'} \mid s, a)$. Ω ist der Beobachtungsraum ($o \in \Omega$) und für einen gegebenen Zustand s gibt $\mathcal{O}$ die Beobachtungswahrscheinlichkeit: $\mathcal{O}(o \mid s, a)$.
Anders als in einem MDP (siehe Abschnitt 3.1.1) basiert die Auswahl von Aktionen in einem POMDP auf der Beobachtung und nicht direkt auf dem Zustand. $\mathcal{R}$ ist die Reward-Funktion und der Agent erhält nach jeder Transition den Reward $\mathcal{R}(s, a)$.

5.1.1 Beobachtungsraum

Die Beobachtung ($o^t \in \Omega$) zu einem Zeitpunkt t besteht aus der Position vom Zielpunkt $o_g^t = \left[d_r^t, \theta_r^t\right]$ im Polarform im Roboter-Koordinatensystem (siehe Abbildung 5.1), seiner aktuellen Geschwindigkeit $o_{vel}^t = [v_t, \omega_t]$ (lineare Geschwindigkeit und Winkelgeschwindigkeit) und den drei letzten 2D-Laser-Daten $o_l = [o_l^{t-2}, o_l^{t-1}, o_l^t]$. Die Laser-Daten repräsentieren die Beziehung zwischen dem Agenten und den Hindernissen in seiner Umgebung. Die letzten 2D-Laser-Daten wurden mitberücksichtigt, um dem Agenten es zu ermöglichen, die Bewegungsrichtung und die Geschwindigkeit von Objekten in der Umgebung vorherzusagen.

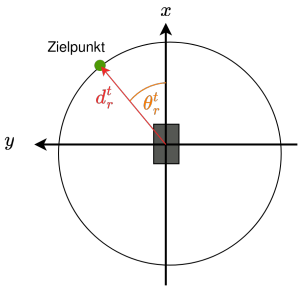
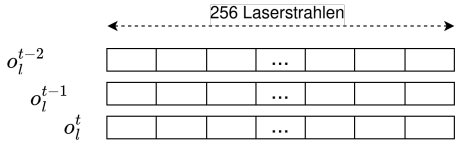
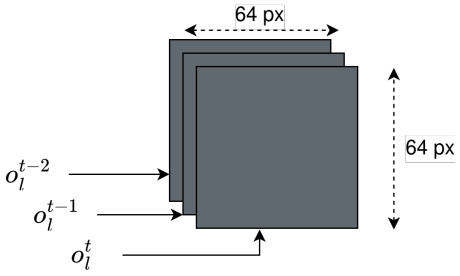


Abbildung 5.1: Zielpunkt im Polarform. Der grüne Punkt repräsentiert den Zielpunkt. Der graue Rechteck ist der Roboter und der ist in die x-Richtung orientiert. Der Winkel θ_r^t wird normalisiert und liegt im Intervall $[-\pi; \pi]$ rad.

Bei der Repräsentation vom 2D-Laser Komponent aus der Beobachtung werden zwei Darstellungsformen verwendet, die in Abbildung 5.2 dargestellt sind:

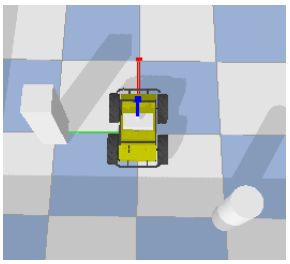


(a) Rohe Darstellung von den Laser-Daten

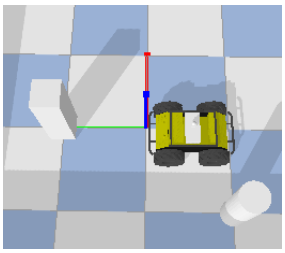


(b) Bildliche Darstellung von den Laser-Daten

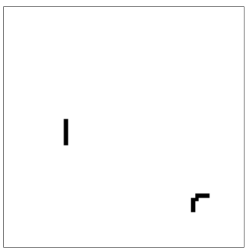
Abbildung 5.2: Die erste Form ist in Abbildung 5.2a dargestellt. Zu jedem der drei Zeitpunkten besteht der Laserscan in Abbildung 5.2a aus 256 Laserstrahlen und der Laserscanner deckt eine Entfernung von 0.03 m bis 3 m im Winkelbereich $[-\frac{\pi}{2}; \frac{\pi}{2}]$ rad ab. In Abbildung 5.2b ist die bildliche Darstellung zu sehen. Die bildliche Darstellung besteht aus einem Stapel von den drei letzten Karten von der Umgebung. Die Karten sind in Abbildung 5.3 dargestellt.



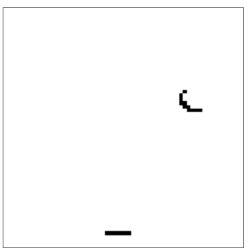
(a) Initiale Position vom Roboter



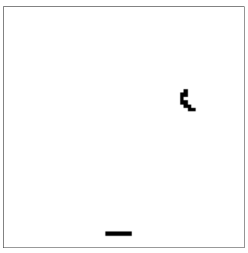
(b) Position nach Drehung um $-\frac{\pi}{2}$ rad



(c) Karte der Umgebung aus der initialen Position (Abbildung 5.3a) ohne Transformation



(d) Abbildung 5.3c nach Transformation



(e) Karte der Umgebung nach der Drehung (Abbildung 5.3b)

Abbildung 5.3: Die Karte der Umgebung aus dem Laserscan. Die Karte ist aus 720 Laserstrahlen im Winkelbereich $[-\pi; \pi]$ rad konstruiert. Die hat eine Dimension von 64 $\times$ 64 und eine Auflösung von 0.05 m. Die schwarzen Punkte markieren erkannte Hindernisse und alle Bereiche ab 1.5 m Entfernung werden als frei betrachtet. Die Mitte des Bildes ist die Position vom Roboter und er ist nach oben orientiert. Um den Einfluss von der Bewegung vom Roboter auf der Karte auszugleichen, werden die letzten Karten immer auf dem aktuellen Koordinaten-Frame vom Laserscanner transformiert. In Abbildung 5.3c ist die Karte von der Umgebung aus der initialen Position vom Roboter (Abbildung 5.3a) zu sehen. Abbildung 5.3d stellt die Transformation der Karte 5.3c nach der Drehung vom Roboter um $-\frac{\pi}{2}$ rad dar. In Abbildung 5.3e sieht man die aktuelle Karte von der Umgebung nach der Drehung vom Roboter um $-\frac{\pi}{2}$ rad (Abbildung 5.3b). Wenn man Abbildung 5.3d und 5.3e betrachtet, stellt man fest, dass die statischen Hindernisse in der selben Position auf den Karten sind und durch die Transformation wurde die Bewegung vom Roboter kompensiert.

5.1.2 Aktionsraum

Eine Aktion a besteht aus der linearen Geschwindigkeit $v \in [0; 1]$ in m/s und der Winkelgeschwindigkeit $\omega \in [-1; 1]$ in rad/s vom Roboter. Kontinuierliche Aktionsräume wurden ausgewählt, denn sie unterstützen flüssige Bewegung und Rückwärtsbewegung wurde zur Vereinfachung des Problems vermieden.

5.1.3 Reward

Wie im Abschnitt 3.1 erwähnt wurde, definiert der Reward das Ziel vom Reinforcement Learning Problem. Bei der Modellierung von der Reward-Funktion werden zwei verschie-

dene Funktionen verwendet. Die erste Funktion $\mathcal{R}_1^t$ wird im statischen Kontext und die zweite $\mathcal{R}_2^t$ im dynamischen Kontext verwendet. Das Ziel vom Agenten im statischen Kontext ist es, zeiteffizient zum Zielpunkt zu navigieren und dabei Kollisionen zu vermeiden. Zusätzlich zur zeiteffizienten und kollisionsfreien Navigation ist ein weiteres Ziel vom Agenten im dynamischen Kontext, eine Geschwindigkeitsgrenze in der Komfortzone von den Fußgängern einzuhalten.

Reward-Funktion im statischen Kontext

Um den Agenten beim Lernen zum Zielpunkt zu führen, während Hindernisse dabei vermieden werden, wird zum Zeitpunkt t der Reward $\mathcal{R}_1^t$ wie folgt definiert:

$$\mathcal{R}_1^t = \begin{cases} \mathcal{R}_{1g}^t + R_s & \text{wenn} \quad l_{min}^t \geq l_{critical} \\ \mathcal{R}_{1c}^t + R_s & \text{sonst} \end{cases} \tag{5.1}$$

$\mathcal{R}_s$ bestraft den Agenten für jeden Schritt, damit er es lernt, mit wenigen Schritten am Ziel zu kommen.

$\mathcal{R}_{1g}^t$ belohnt die Änderung des Abstands zum Zielpunkt und die Ankunft am Zielpunkt.

$\mathcal{R}_{1c}^t$ sollte dem Agenten dazu ermutigen, Kollisionen zu vermeiden, indem er bei einer Kollision bestraft wird. Die Funktionen $\mathcal{R}_{1g}^t$ und $\mathcal{R}_{1c}^t$ sind in 5.2 und 5.3 definiert.

l_{min}^t ist die minimale Laser-Distanz zum Zeitpunkt t und $l_{critical}$ definiert einen Schwellwert, unter dem die Kollisionsvermeidung über die Navigation zum Ziel privilegiert wird. Wenn der Abstand zum nächsten Hinderniss kleiner als $l_{critical}$ ist, dann befindet sich der Agent in einem kritischen Bereich, in dem die Navigation zum Ziel nicht belohnt wird. Nach einigen Versuchen wurde festgestellt, dass die Belohnung der Navigation zum Ziel zu Kollisionen führte, wenn das Ziel hinter einem Hindernis ist und die Wege schmal sind. Dies hat dazu motiviert, die Kollisionsvermeidung in kritischen Bereichen zu privilegieren. Im kritischen Bereich wurden weitere Ansätze ausprobiert: zum Beispiel die negative Belohnung vom Agenten für jeden Schritt näher zum nächsten Hindernis und eine positive Belohnung, wenn der Schritt ihn vom Hindernis entfernt. Aber die Deaktivierung der Belohnung im kritischen Bereich außer im Falle einer Kollision hat die besten Ergebnisse geliefert.

$$\mathcal{R}_{1g}^{t} = \begin{cases} r_{goal} & \text{wenn} \quad d_r^t < goal\_radius \\ w\left(d_r^{t-1} - d_r^t\right) & \text{sonst} \end{cases} \tag{5.2}$$

$$\mathcal{R}_{1c}^{t} = \begin{cases} r_{collision} & \text{wenn} \quad l_{min}^t \leq col_{dist} \\ 0 & \text{sonst} \end{cases} \tag{5.3}$$

In 5.2 sind d_r^t und d_r^{t-1} die relative euklidische Distanz zum Zielpunkt zu den Zeitpunkten t und $t-1$. $w > 0$ ist ein Gewichtungsfaktor. Die Differenz $d_r^{t-1} - d_r^t$ führt dazu, dass der Agent positiv belohnt wird, wenn er zum Zielpunkt fährt und negativ ansonsten.

Reward-Funktion im dynamischen Kontext

Im dynamischen Kontext wird $\mathcal{R}_2^t$ für die Belohnung verwendet. $\mathcal{R}_2^t$ ist eine Erweiterung von $\mathcal{R}_1^t$ (siehe 5.1). Zustäzlich zur Bestrafung bei jedem Schritt und bei der Kollision mit Hindernissen wird der Agent im dynamischen Kontext bestraft, wenn der Roboter die in der Komfortzone von Fußgängern maximale erlaubte Geschwindigkeit überschreitet. Dies ist der Fall, wenn folgende drei Bedingungen gelten:

- der Abstand zwischen dem Roboter und dem nächsten Fußgänger ist kleiner als ped_{radius},
- der Fußgänger befindet sich im Winkelbereich $[-\frac{\pi}{4}; \frac{\pi}{4}]$ rad relativ zum Koordinatensystem vom Roboter,
- der Roboter fährt mit einer Geschwindigkeit, die größer als 0.1 m/s ist.

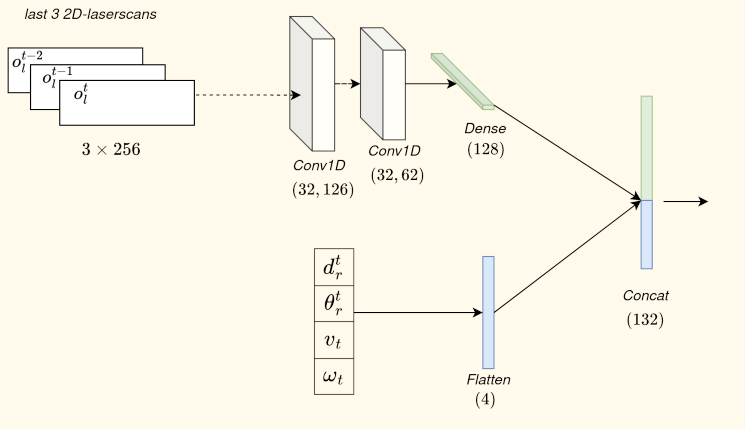
$\mathcal{R}_1^t$ und $\mathcal{R}_2^t$ unterscheiden sich daher nur in der Definition von $\mathcal{R}_{2c}^t$: $\mathcal{R}_{2g}^t = \mathcal{R}_{1g}^t$.
In $\mathcal{R}_2^t$ wird $\mathcal{R}_{2c}^t$ wie folgt definiert:

$$\mathcal{R}_{2c}^{t} = \begin{cases} r_{collision} & \text{wenn} \quad (l_{min}^t \leq col_{dist}) \vee (\text{max vel in Komfortzone} > 0.1 \text{ m/s}) \\ 0 & \text{sonst} \end{cases} \tag{5.4}$$

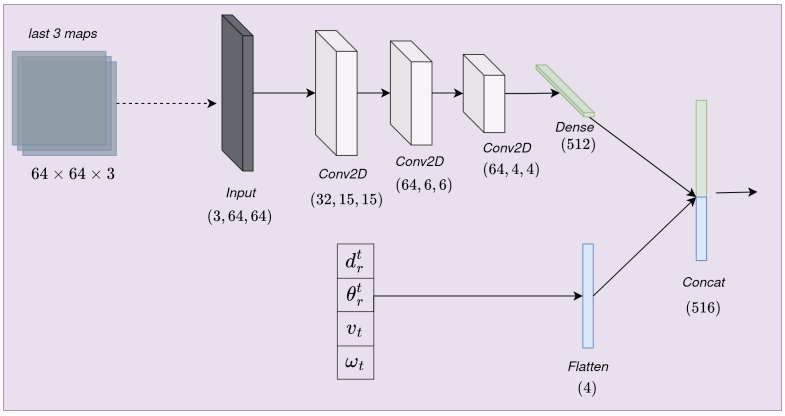
5.1.4 Struktur der neuronalen Netze

Zur Lösung vom Navigationsproblem wird eine Soft Actor-Critic Policy (siehe Abschnitt 3.2.4) trainiert. Dabei wird die Implementierung von Stable-Baselines3 vom SAC Algorithmus verwendet. SB3 bietet eine Open-Source Implementierung von Deep-RL Algorithmen in Python [33]. Wie im Abschnitt 5.1.1 erwähnt wurde, besteht die Beobachtung aus den Laser-Daten, der Geschwindigkeit vom Roboter und der Position vom Zielpunkt. Da die Laser-Daten komplexer sind werden die durch einen Feature-Extractor zuerst vorverarbeitet. Hierbei wird für die rohe Darstellung von den Laser-Daten (siehe Abbildung 5.2a) das 1D-Convolutional-Neural-Network in Abbildung 5.4a verwendet. Für die bildliche Darstellung (siehe Abbildung 5.2b) wird das 2D-Convolutional-Neural-Network (2D-CNN) in Abbildung 5.4b verwendet.

Die Feature-Extractors sind Teil der Soft Actor-Critic Netze und werden mittrainiert. SB3 bietet die Möglichkeit, dass sich der Actor und der Critic den Feature-Extractor teilen, aber in den Versuchen wurde festgestellt, dass der Agent besser lernt, wenn sowohl Actor als auch Critic sein eigenes Feature-Extractor Netz hat.



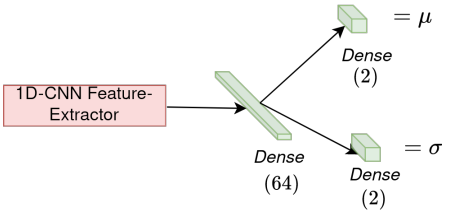
(a) Feature-Extractor für rohe Laser-Daten. Die Eingabe vom Netz hat die Dimension 3×256. Die ReLU-Funktion wurde als Aktivierungsfunktion verwendet und die wurde auf die Ausgabe von den Convolutional- und Dense-Layers angewendet. Die Ausgabe vom Dense-Layer wird auf die zwei Komponente von o_{vel}^t und die zwei Komponente von o_g^t erweitert und bildet damit die Ausgabe vom Feature-Extractor Netz. Die Tupeln unter den jeweiligen Layers beschreiben die Dimension deren Ausgaben. Das Netz wurde von [29] inspiriert.



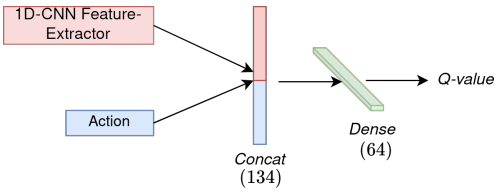
(b) Feature-Extractor für Bilddaten. Die Eingabe vom Netz hat die Dimension $64 \times 64 \times 3$. Die ReLU-Funktion wurde als Aktivierungsfunktion verwendet und die wurde auf die Ausgabe von den Convolutional- und Dense-Layers angewendet. Die Ausgabe vom Dense-Layer wird auf die zwei Komponente von o_{vel}^t und die zwei Komponente von o_g^t erweitert und bildet damit die Ausgabe vom Feature-Extractor Netz. Die Tupeln unter den jeweiligen Layers beschreiben die Dimension deren Ausgaben.

Abbildung 5.4: Feature-Extractor Netze

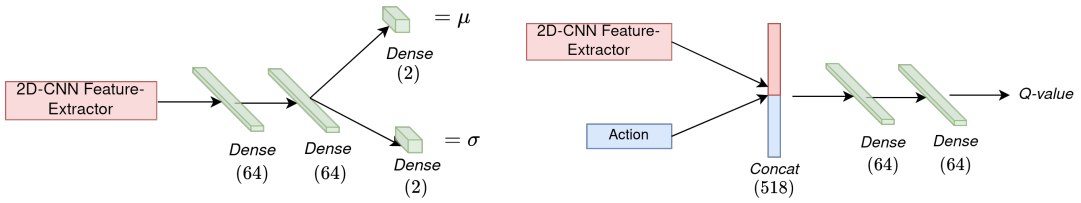
Aus dem originalen Paper vom Soft Actor-Critic (siehe Formel 3.28) wurde bei der Approximierung der Soft Q-Funktion ein *State-value network* ($V_{\bar{\psi}}$) als Target Netz verwendet. SB3 verwendet stattdessen ein *Action-value network*. Die endgültige Architektur vom Actor- und vom Critic-Netz ist in Abbildung 5.5 dargestellt.



(a) Actor-Network für rohe Laserdaten



(b) Critic-Network für rohe Laserdaten



(c) Actor-Network für Bilddaten (d) Critic-Network für Bilddaten

Abbildung 5.5: Actor- und Critic Netze. Die beiden Actor Netze 5.5a und 5.5c geben den Erwartungswert und die Standardabweichung für die jeweiligen Komponenten einer Aktion aus (siehe Abschnitt 3.2.3). Aus diesen jeweiligen Verteilungen kann eine Aktion ausgewählt werden. Die Critic Netze geben einen skalaren Wert aus, der den Action-value definiert. Um eine Überschätzung von den Aktionen im Training zu vermeiden, verwendet SB3 für das Critic Netz und das Target Critic Netz jeweils 2 Netze und der minimale Action-Value aus den zwei Netzen wird verwendet.

5.2 Training

Das Training wird auf einem Lenovo LEGION 5 Pro Laptop durchgeführt. Die Hardware und Software Spezifikationen vom Computer sind in der Tabelle A.3 zu finden.

Es werden zwei Typen von Agenten trainiert: Agenten, die nur gegenüber statischen Hindernissen trainiert werden und Agenten, die während des Trainings sowohl statischen als auch dynamischen Hindernissen ausgesetzt sind. Bei der Evaluation wird gegenüber dynamischen Hindernissen das Verhalten von den zwei Typen von Agenten verglichen, um den Einfluss der dynamischen Hindernisse auf die Bewegungsplanung von den Agenten zu untersuchen.

Im statischen Kontext werden Static-Agent-1, Static-Agent-2, Static-Agent-3 und im dynamischen Kontext Dynamic-Agent-1 und Dynamic-Agent-2 trainiert.

Die maximale Länge einer Episode im Training ist auf 600 Schritte gesetzt, wobei der Agent jede Aktion 0.1 s lang wiederholt, bevor der nächste Schritt beginnt. Diese Wiederholung dient dazu, eine signifikante Änderung der Distanz für die Berechnung vom Reward zu haben.

Static-Agent-1, Static-Agent-2 und Dynamic-Agent-1 verwenden die rohen Laser-Daten und werden mit dem dafür vorgesehenen Netz trainiert (siehe Abbildung 5.5). Bei Static-Agent-2 werden außerdem die vom Actor Netz ausgegebenen Aktionen so angepasst, dass

die Änderung in zwei aufeinanderfolgenden Aktionen nicht 0.1 m/s bzw. 0.1 rad/s überschreitet.
Mit dieser Anpassung werden große Beschleunigungen vermieden. Obwohl kontinuierliche Aktionsräume eine flüssige Bewegung unterstützen, guarantieren sie alleine keine Vermeidung von großen Sprüngen bei der Aktionsauswahl. Das Ziel dieser Einschränkung ist es daher, zu untersuchen, ob man dadurch vermeiden kann, dass der Roboter in seiner Bewegung stark wackelt und ob eine flüssigere Bewegung als mit Static-Agent-1 dadurch erreicht werden kann.
Static-Agent-3 und Dynamic-Agent-2 verwenden die bildliche Darstellung von den Laser-Daten und beim Training wird das entsprechende Netz verwendet (siehe Abbildung 5.5).

5.2.1 Statischer Kontext

Die statische Umgebung fürs Training besteht aus einem 7 m $\times$ 7 m Feld, in dem am Anfang jeder Episode die Hindernisse neu eingefügt werden. Die Umgebung wird so konfiguriert, dass der Agent mit vielen vielfältigen Situationen konfrontiert wird.

Zu Beginn einer Episode werden zuerst aus einer Gleichverteilung der Start- und Zielpunkt und die Orientierung vom Roboter am Startpunkt zuffällig im Feld ausgewählt. Die euklidische Distanz zwischen den beiden Punkten liegt dabei im Intervall $[2; 3]$ m. Die Orientierung wird aus dem Intervall $[-\pi; \pi]$ rad ausgewählt.

Die Anzahl der Hindernisse $n \in N = \{6, 7, 8\}$ im Feld und deren Art werden zudem am Anfang jeder Episode zuffällig ausgewählt. Die Menge der Hindernisse besteht aus drei Objekten von 4.2a, vier Objekten von 4.2b und vier Objekten von 4.2c (siehe Abbildung 4.2).

Die Hindernisse werden so positioniert, dass sie mindestens 1.5 m voneinander entfernt sind und dass sie mindestens 1 m vom Startpunkt und 1 m vom Zielpunkt entfernt sind.

Abbildung 5.6 stellt das 7 m $\times$ 7 m Feld mit sechs zuffällig ausgewählten Hindernissen dar:

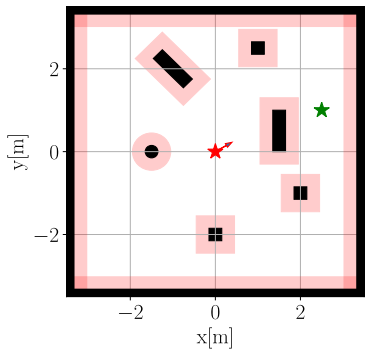
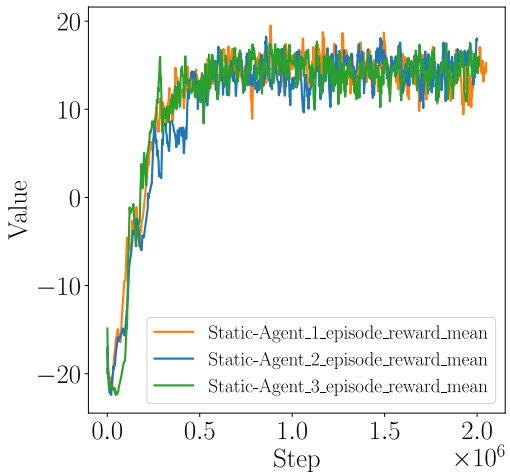


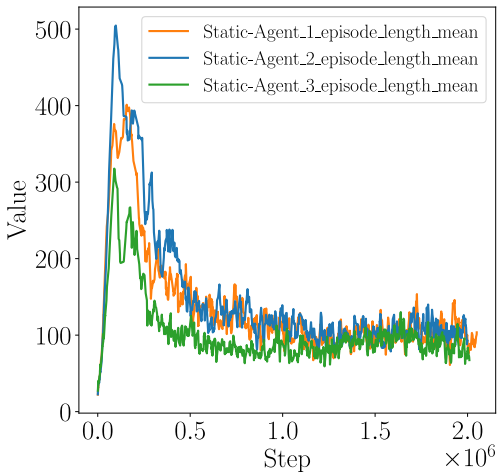
Abbildung 5.6: Statische Umgebung mit zuffälligen Hindernissen. Der rote Punkt und Pfeil beschreiben die Startposition und die Orientierung vom Roboter im Raum. Der grüne Punkt ist der Zielpunkt. Der schwarze Bereich am Rand repräsentiert Wände und die schwarzen Objekte im Feld sind statische Hindernisse aus Abbildung 4.2. Der rote Bereich um die Hindernisse markiert die kritische Zone, in der der Roboter die Vermeidung der Hindernisse privilegiert.

Im statischen Kontext werden Static-Agent-1, Static-Agent-2 und Static-Agent-3 jeweils in zehn parallelen statischen Umgebungen mit der Reward-Funktion R_1^t (siehe Formel 5.1) trainiert. Jede Episode wird beendet, sobald eine Kollision stattfindet oder der Agent den Zielpunkt erreicht hat. Die Agenten werden alle 50 000 Schritte mit der eben beschriebenen Konfiguration von Hindernissen für 50 Episoden evaluiert.

Die Ergebnisse vom Training für 2 000 000 Schritte in ca. 18 Stunden pro Agent und die von der Evaluation während des Trainings sind in Abbildung 5.7 und 5.8 dargestellt:

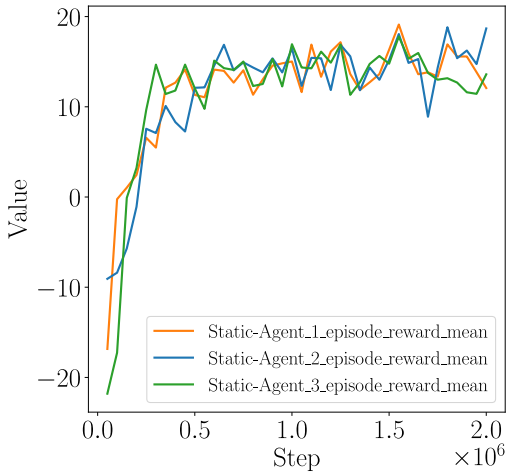


(a) Durchschnittlicher Reward pro Episode

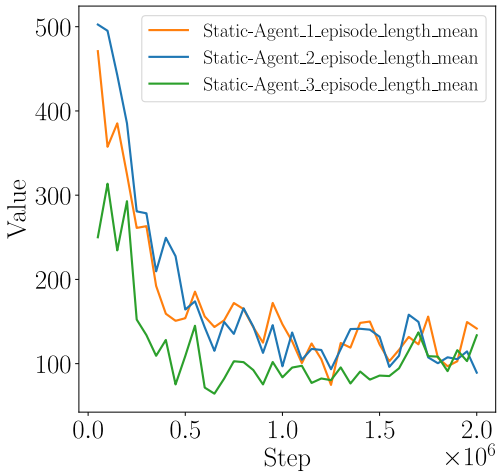


(b) Durchschnittliche Länge einer Episode

Abbildung 5.7: Die Abbildung präsentiert den durchschnittlichen Reward pro Episode (episode_reward_mean) und die durchschnittliche Episodenlänge (episode_length_mean) für je 100 Episoden im Training.



(a) Durchschnittlicher Reward pro Episode



(b) Durchschnittliche Länge einer Episode

Abbildung 5.8: Die Abbildung präsentiert den durchschnittlichen Reward pro Episode und die durchschnittliche Episodenlänge für je 50 Episoden bei der Evaluation.

Bei allen Agenten ist der durchschnittliche Reward in den ersten 1 000 000 Schritten stark gestiegen. Neben dem durchschnittlichen Reward wurde ebenfalls die durchschnittliche Episodenlänge dargestellt, denn der durchschnittliche Reward alleine reicht nicht aus, um eine Aussage über die Performanz von einem Agenten zu treffen. Auch wenn der Reward groß ist, wäre die Performanz nicht gut, wenn die Anzahl der Schritte zum Ziel mit der Zeit nicht abnehmen würde.

5.2.2 Dynamischer Kontext

Im dynamischen Kontext werden Dynamic-Agent-1 und Dynamic-Agent-2 trainiert.
Die Agenten werden in acht parallelen Umgebungen trainiert, wobei vier Umgebungen statisch sind (siehe Abschnitt 5.2.1). In den vier weiteren werden jeweils acht Fußgänger eingefügt, die sich jeweils zwischen einem Start- und Zielpunkt hin und her bewegen. Die statischen und dynamischen Hindernisse wurden nicht in einer selben Umgebung gemischt, um überfüllte Umgebungen zu vermeiden und um dabei eine ausreichende Repräsentation von sowohl dynamischen als auch statischen Hindernissen zu erreichen. Auch ohne die Mischung sollten die Agenten in der lage sein, dynamische von statischen Hindernissen zu unterscheiden, denn die Trainingsdaten werden parallel in beiden Umgebungsarten gesammelt.

Die dynamische Umgebung ist in Abbildung 5.9 dargestellt:

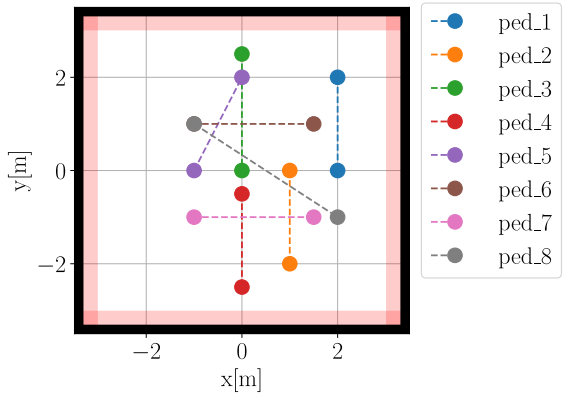
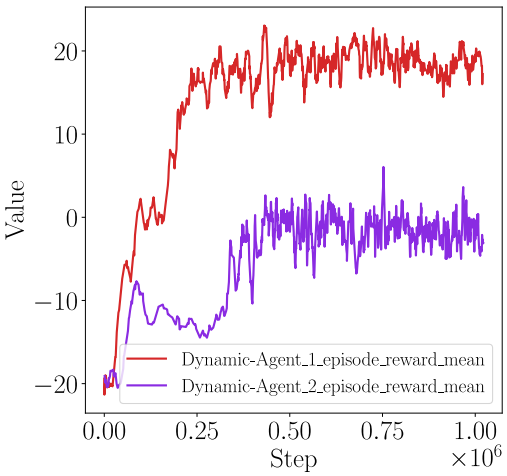


Abbildung 5.9: Dynamische Umgebung. Für die jeweiligen Fußgänger (ped_1 bis ped_8) in der Abbildung markieren die jeweiligen zwei Punkte den Start und Zielpunkt, zwischen denen die Agenten hin und her laufen. Der direkte Weg zwischen den Punkten ist durch die gestrichelten Linien dargestellt. Da der Roboter, die Wände oder die anderen Fußgänger nach dem Social-Force-Model die Bewegung von einem Fußgänger beeinflussen, nehmen die Fußgänger nicht immer den direkten oder den selben Weg zwischen Start- und Zielpunkt.

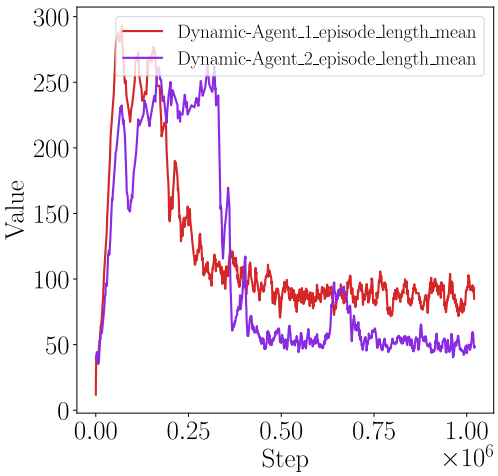
Die Koordinaten (x_s,y_s) vom Startpunkt, (x_g,y_g) vom Zielpunkt und die Orientierung vom Roboter am Start werden zu Beginn jeder Episode aus einer uniformen Verteilung ausgewählt: $x_s \in [-2.5; -2]$ m , $y_s \in [-2.5; 2.5]$ m, $x_g \in [1; 2.5]$ m und $y_g \in [-2.5; 2.5]$ m. Die Orientierung wird aus dem Intervall $[-\frac{\pi}{2}; \frac{\pi}{2}]$ rad ausgewählt.

Pedsim_ros wählt die Geschwindigkeit in m/s von den jeweiligen Fußgängern aus einer normalen Verteilung aus. Dabei wurde für den Erwartungswert 1 und für die Standardabweichung 0.2 gewählt.

Diese Konfiguration von Start- und Zielposition führt dazu, dass der Roboter den Weg der Fußgänger traversieren muss und dies erhöht die Wahrscheinlichkeit, dass er die Fußgänger begegnet. Jede Episode wird bei einer Kollision, beim Ankunft am Zielpunkt oder bei einer Überschreitung der Geschwindigkeitsgrenze in der Komfortzone von den Fußgängern beendet. Die Agenten werden während des Trainings alle 80 000 Schritte in einer dynamischen Umgebung (siehe Abbildung 5.9) für 50 Episoden evaluiert. Die Ergebnisse vom Training für 1 000 000 Schritte und von der Evaluation während des Trainings sind in Abbildung 5.10 und 5.11 dargestellt:

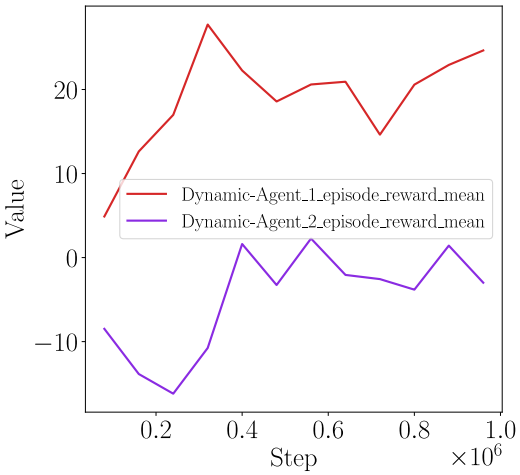


(a) Durchschnittlicher Reward pro Episode

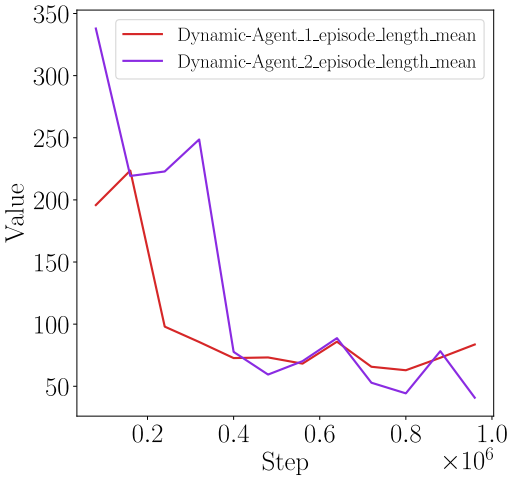


(b) Durchschnittliche Länge einer Episode

Abbildung 5.10: Die Abbildung präsentiert den durchschnittlichen Reward pro Episode und die durchschnittliche Episodenlänge für je 100 Episoden im Training.



(a) Durchschnittlicher Reward pro Episode



(b) Durchschnittliche Länge einer Episode

Abbildung 5.11: Die Abbildung präsentiert den durchschnittlichen Reward pro Episode und die durchschnittliche Episodenlänge für je 50 Episoden bei der Evaluation.

Wenn man sich die Graphen in Abbildung 5.10 und 5.11 anschaut, fällt der niedrige Reward von Dynamic-Agent-2 sowohl im Training als auch in der Evaluation sofort auf. Während der Reward ab Schritt 500 000 um die 0 liegt und sich kaum verbessert hat, liegt die Episodenlänge unter 100. Dies ist ein Zeichen, dafür dass der Roboter öfter mit den Fußgängern kollidiert, oder dass er die maximale erlaubte Geschwindigkeit in der Komfortzone von den Fußgängern überschreitet. Im Abschnitt 5.3.2 wird dieser Fall genauer betrachtet, indem die Bewegung vom Roboter, die von den Fußgängern und die Geschwindigkeit vom Roboter beobachtet wird. Im Gegensatz zu Dynamic-Agent-2 zeigt Dynamic-Agent-1 sowohl im Training als auch in der Evaluation eine gute Performanz.

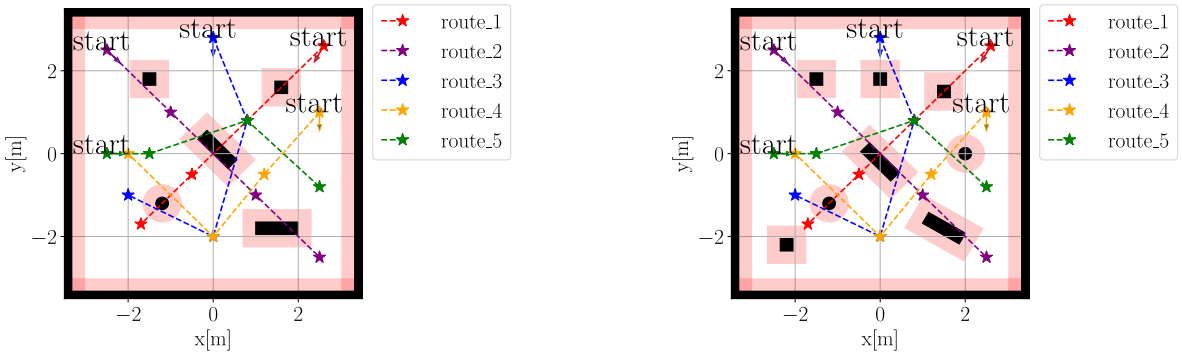
5.3 Evaluation

Die Evaluation findet auf dem selben Computer wie beim Training statt. Die Inferenzzeit ist bei allen Agenten etwa gleich. Für Static-Agent-1, Static-Agent-2 und Dynamic-Agent-1 wurde 0.0023 s gemessen. Für Static-Agent-3 und Dynamic-Agent-2 liegt die bei 0.0026 s.

Zusätzlich zur Evaluation während des Trainings ist es wichtig, weitere Evaluationen durchzuführen und zum Beispiel dabei die Navigationszeit, den gefolgten Weg von den Agenten zu betrachten oder auch Situation zu identifizieren, mit denen die Agenten nicht klar kommen. Dies ist für ein weiteres Training wichtig, denn man kann dadurch in einer weiteren Modellierung von der Trainingsumgebung Situationen berücksichtigen, in denen die Agenten Probleme haben.

5.3.1 Statische Umgebung

Im statischen Kontext wird das Verhalten von den Agenten auf fünf ausgewählte Routen evaluiert. Dabei werden zwei verschiedene Konfigurationen von Hindernissen verwendet. Eine Route besteht aus mehreren Zielpunkten, die der Agent der Reihe nach durchfahren sollte. Die Routen und die verschiedenen Konfigurationen von Hindernissen sind in Abbildung 5.12 dargestellt:



(a) Hindernis-Konfiguration-1

(b) Hindernis-Konfiguration-2

Abbildung 5.12: Statische Umgebungen zur Evaluierung des Verhaltens von den Agenten. Die gestrichelten Linien verbinden die Zielpunkte auf der Route, die der Roboter beginnend vom Startpunkt der Reihe nach fahren sollte.

Bei der Evaluation werden für jede Route 10 Versuche durchgeführt. Bei jedem Versuch werden über den gesamten Weg folgende vier Metriken berechnet: die Erfolgsrate S_{rate}, die Navigationszeit t_{nav} in s, die maximale Änderung der linearen Geschwindigkeit $\max(\Delta v)$ in m/s, die maximale lineare Geschwindigkeit $\max(v)$ in m/s, die Kollisionsrate C_{rate} und die Anzahl von Zeitüberschreitungen N_{timeout}.

Die Erfolgsrate S_{rate} per Versuch wird berechnet, indem die Anzahl der vom Agenten erreichten Zielpunkte durch die gesamte Anzahl von Zielpunkten auf der Route dividiert wird.
Die Kollisionsrate C_{rate} ist die totale Anzahl von Kollisionen in den Versuchen geteilt durch die gesamte Anzahl von Versuchen.
Eine Zeitüberschreitung liegt vor, wenn der Agent nach 1200 Schritten nicht am letzten Zielpunkt angekommen ist.

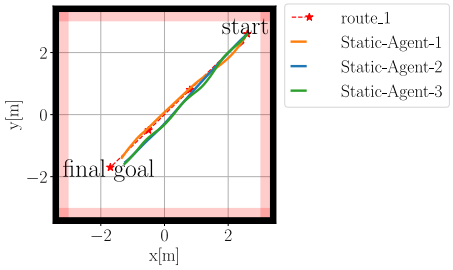
Für t_{nav}, S_{rate}, $\max(\Delta v)$ und $\max(v)$ wird der Durchschnitt aus den 10 Versuchen gebildet.

Zusätzlich zu den Metriken wird der gefolgte Weg auf jeder Route dargestellt, um seine Qualität bewerten zu können. Um zu untersuchen, welchen Einfluss die Hindernisse auf der Bewegungsplanung von den Agenten haben, wird die Evaluation auf den fünf Routen erstmal ohne Hindernisse im Feld und erst danach mit Hindernissen durchgeführt.

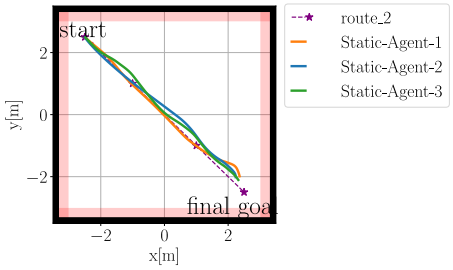
Evaluation in der leeren Umgebung

In der Evaluation im leeren Feld sind alle Agenten erfolgreich am finalen Ziel angekommen (siehe Abbildung 5.13 und Tabelle A.4). Der Einfluss von der Einschränkung in der Beschleunigung ist jedoch deutlich zu erkennen. Static-Agent-1 kommt in allen Routen etwa schneller am finalen Ziel als Static-Agent-2 an. Static-Agent-1 braucht durchschnittlich 8.6 s für eine Route im Vergleich zu 10.3 s bei Static-Agent-2.
Über alle Routen liegt die durchschnittliche maximale Geschwindigkeit von Static-Agent-1 bei 0.94 m/s, während der Wert bei 0.89 m/s für Static-Agent-2 liegt. Static-Agent-1 ist etwa schneller, denn er kann schneller beschleunigen bzw. entschleunigen und seine maximale lineare Beschleunigung über alle Routen liegt durchschnittlich bei 3.7 m/s^2. Bei Static-Agent-2 beträgt der Wert wegen der limitierten Beschleunigung 1.2 m/s^2.

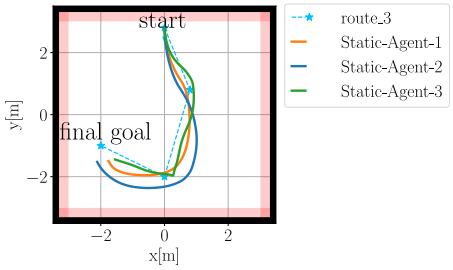
Diese Einschränkung wiederspiegelt sich zudem auf dem gefahrenen Weg. Wenn man Abbildung 5.13d zum Beispiel betrachtet, erkennt man, dass Static-Agent-2 im Vergleich zu den anderen Agenten einen etwa größeren Bogen zwischen dem Start- und dem ersten Zielpunkt und zwischen dem zweiten Zielpunkt und dem finalen Zielpunkt zurücklegt. Dies kann man außerdem auf der grafischen Darstellung der Geschwindigkeiten in Abbildung 5.14 ablesen. Das selbe Verhalten kann man in Abbildung 5.13c zwischen dem ersten Zielpunkt nach dem Startpunkt und dem finalen Zielpunkt beobachten. Das passiert immer, wenn der Roboter nach dem Start nicht zum Zielpunkt orientiert ist.



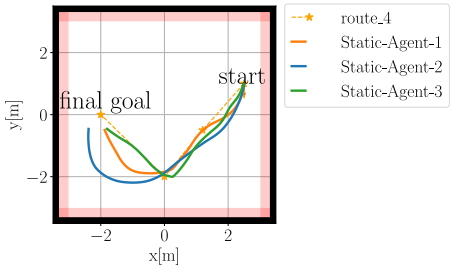
(a) Route 1



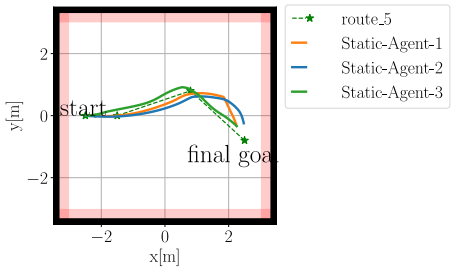
(b) Route 2



(c) Route 3

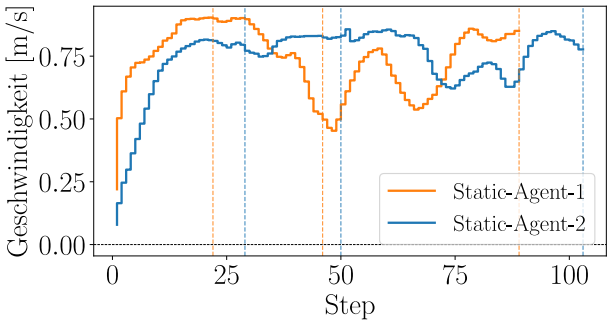


(d) Route 4

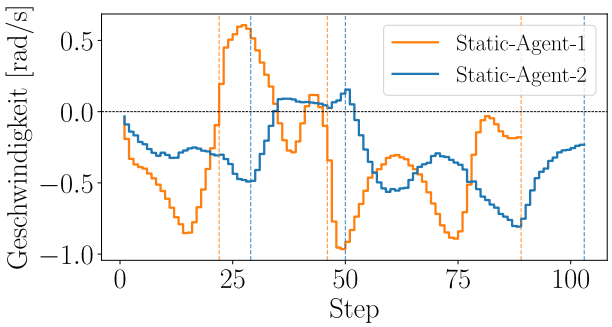


(e) Route 5

Abbildung 5.13: Die gefolgten Wege von den drei Agenten im leeren Feld.



(a) Lineare Geschwindigkeit



(b) Winkelgeschwindigkeit

Abbildung 5.14: Lineare und Winkelgeschwindigkeit von Static-Agent-1 und Static-Agent-2 auf Route_4. Die gestrichelten senkrechten Linien markieren den Zeitschritt, an dem die Agenten die verschiedenen Zielpunkte erreicht haben und die zeigen, dass Static-Agent-1 immer zuerst am nächsten Zielpunkt ankommt. Da der Roboter am Startpunkt nicht zum nächsten Ziel orientiert ist, ist Static-Agent-2 in den ersten 20 Schritten nicht in der Lage eine große Winkelgeschwindigkeit wie Static-Agent-1 zu erreichen und er legt daher einen größeren Bogen zurück. Der selbe Fall lässt sich zwischen dem zweiten Zielpunkt und dem finalen Ziel betrachten. Direkt nach Ankunft am zweiten Zielpunkt konnte Static-Agent-1 eine Winkelgeschwindigkeit von ca -0.9 rad/s erreichen.

Evaluation in Hindernis-Konfiguration-1

Bei gegebenen Hindernissen in der Umgebung haben es alle Agenten gelernt, die Hindernisse in deren Bewegungsplanung zu berücksichtigen. Die Ergebnisse der Evaluation sind in Abbildung 5.15 und Tabelle A.5 dokumentiert.

In dieser ersten Konfiguration der Umgebung mit Hindernissen kommen alle Agenten auch am finalen Zielpunkt an, aber wegen der Hindernisse brauchen sie diesmal alle eine längere Zeit. Durchschnittlich über alle Routen brauchen Static-Agent-1, Static-Agent-2 und Static-Agent-3 18.3 s, 17.76 s und 22.74 s bis zum finalen Ziel.

Auf Route_1, Route_3 und Route_5 haben die Agenten in den 10 Versuchen den selben Weg zum finalen Ziel genommen. Auf Route_2 hat Static-Agent-3 einen schnelleren Weg gelernt und ist 10 s früher am finalen Ziel als Static-Agent-1 (siehe Abbildung 5.15b). Auf Route_4 hat er 40% der Versuche den Weg in Abbildung 5.15d genommen.

Wenn man die Navigationszeit (Siehe Tabelle A.5) betrachtet, stellt man fest, dass Static-Agent-3 vor allem auf Route_1 und Route_4 langsamer als Static-Agent-1 ist. Seine kleinere Sichtweite könnte der Grund dafür sein, warum er langsamer fährt. Auf Route_1 (Abbildung 5.15a) zum Beispiel hat Static-Agent-3 durchschnittlich 40.87 s verbracht im Vergleich zu 25.10 s bei Static-Agent-1.

Auf Route_2 (Abbildung5.15b) ist Static-Agent-2 trotz der Limitierung in der Beschleunigung etwa 10 s früher am finalen Ziel als Static-Agent-1. Der Zeitunterschied liegt daran, dass Static-Agent-1 zwischen dem zweiten und dem finalen Zielpunkt etwa 10 s länger braucht. Insgesamt sind alle Agenten langsam in engen Bereichen aber hier könnte es sein, dass Static-Agent-1 weniger mit einer ähnlichen Situation im Training konfrontiert war.

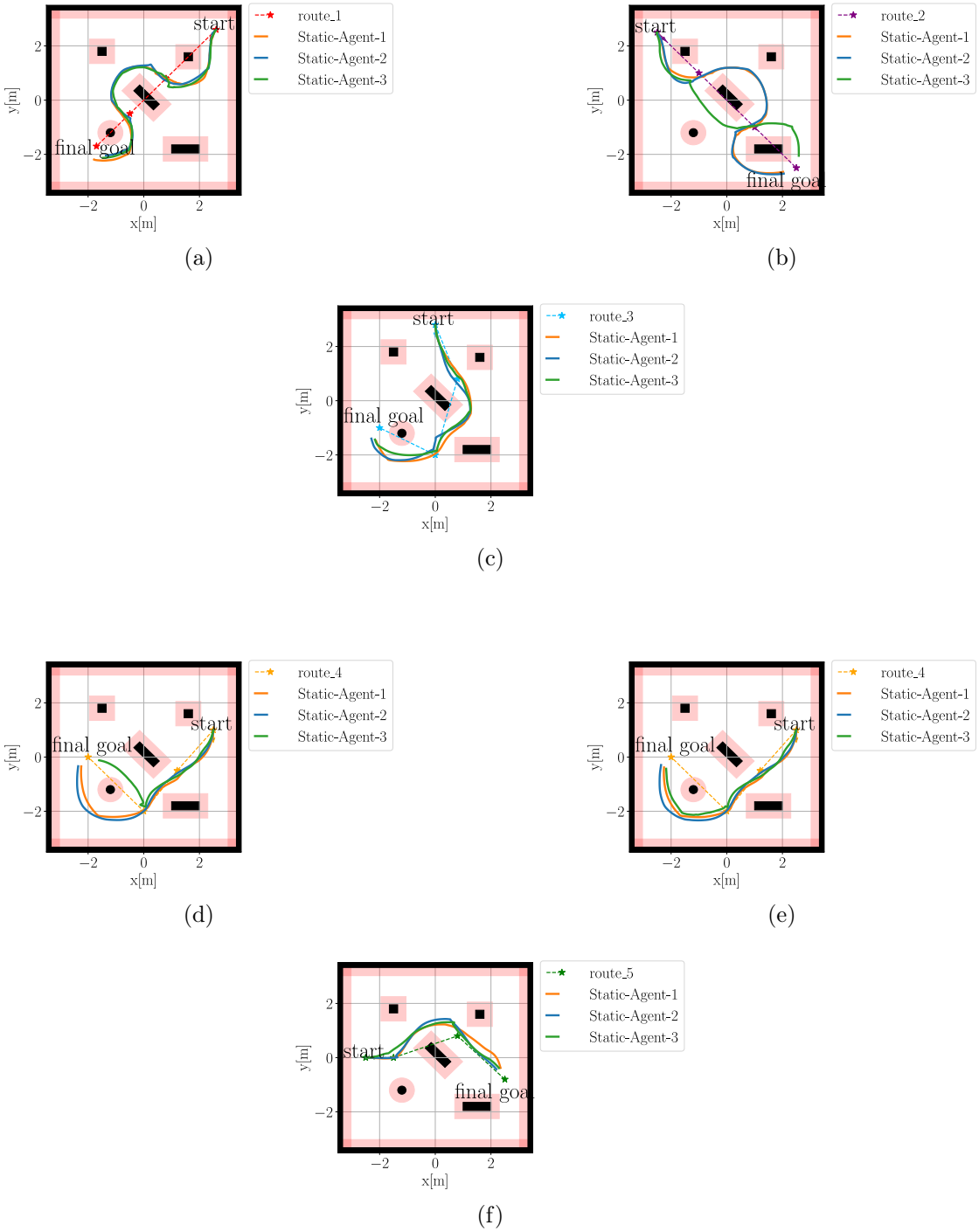


Abbildung 5.15: Die gefolgten Wege von den drei Agenten in Hindernis-Konfiguration-1.

Evaluation in Hindernis-Konfiguration-2

Beim Erhöhen der Anzahl von Hindernissen ist es zu einigen Kollisionen gekommen (siehe Tabelle A.6). In einigen Situationen konnten die Agenten zudem alle Zielpunkte in der vorgegebenen Zeit nicht erreichen. Das ist zu erwarten, wenn die Zielpunkte in engen Räumen sind und die Agenten die Vermeidung von Hindernissen gegenüber der Navigation zum Ziel bevorzugen.

Static-Agent-1 hat in dieser Konfiguration eine Erfolgsrate von 0.67 auf Route_1 und Route_3, 0.9 auf Route_2, 1 auf Route_4 und 0.77 auf Route_5 erreicht.
Auf Route_2 ist er einmal in den 10 Versuchen mit einem Hinderniss Kollidiert und auf Route_3 ist er bei jedem Versuch an der selben Stelle Kollidiert. Auf Route_5 an etwa der selben Stelle wie bei Route_3 wäre es in Abbildung 5.16k zu einer Kollision gekommen, wenn die Zeit nicht überschritten wäre. Die Kollisionsstellen sind in den Abbildungen 5.16d und 5.16g zu sehen.

Auf Route_1 und Route_5 hat Static-Agent-1 70% der Versuche die maximale Zeit überschritten, denn im Gegensatz zur ersten Konfiguration der Hindernisse sind die Hindernisse dichter nebeneinander und der Agent bevorzugt die Vermeidung von Hindernissen, sobald das nächste Hindernis näher als $l_{critical}$ ist. In Abbildung 5.16c zum Beispiel konnte der Agent den zweiten Zielpunkt nicht erreichen und ist nach mehreren Versuchen zurück zum ersten Zielpunkt zurückgefahren, um es zu versuchen, auf der anderen Seite einen Weg zu finden. Wenn man außerdem Abbildung 5.16i betrachtet sieht man, dass sich der Agent nach dem zweiten Zielpunkt 70% der Versuche für einen ineffizienteren Weg entschieden hat und er ist etwa weiter gerade gefahren statt direkt nach Rechts abzubiegen und zum letzten Zielpunkt zu fahren. Auch wenn er in Abbildung 5.16j zum letzen Zielpunkt direkt gefahren ist, nachdem er zum zweiten Zielpunkt zurückgefahren ist, hat er in 90% der Versuche entweder die Zeit überschritten oder einen längeren Weg genommen (siehe Abbildung 5.16i und 5.16l).

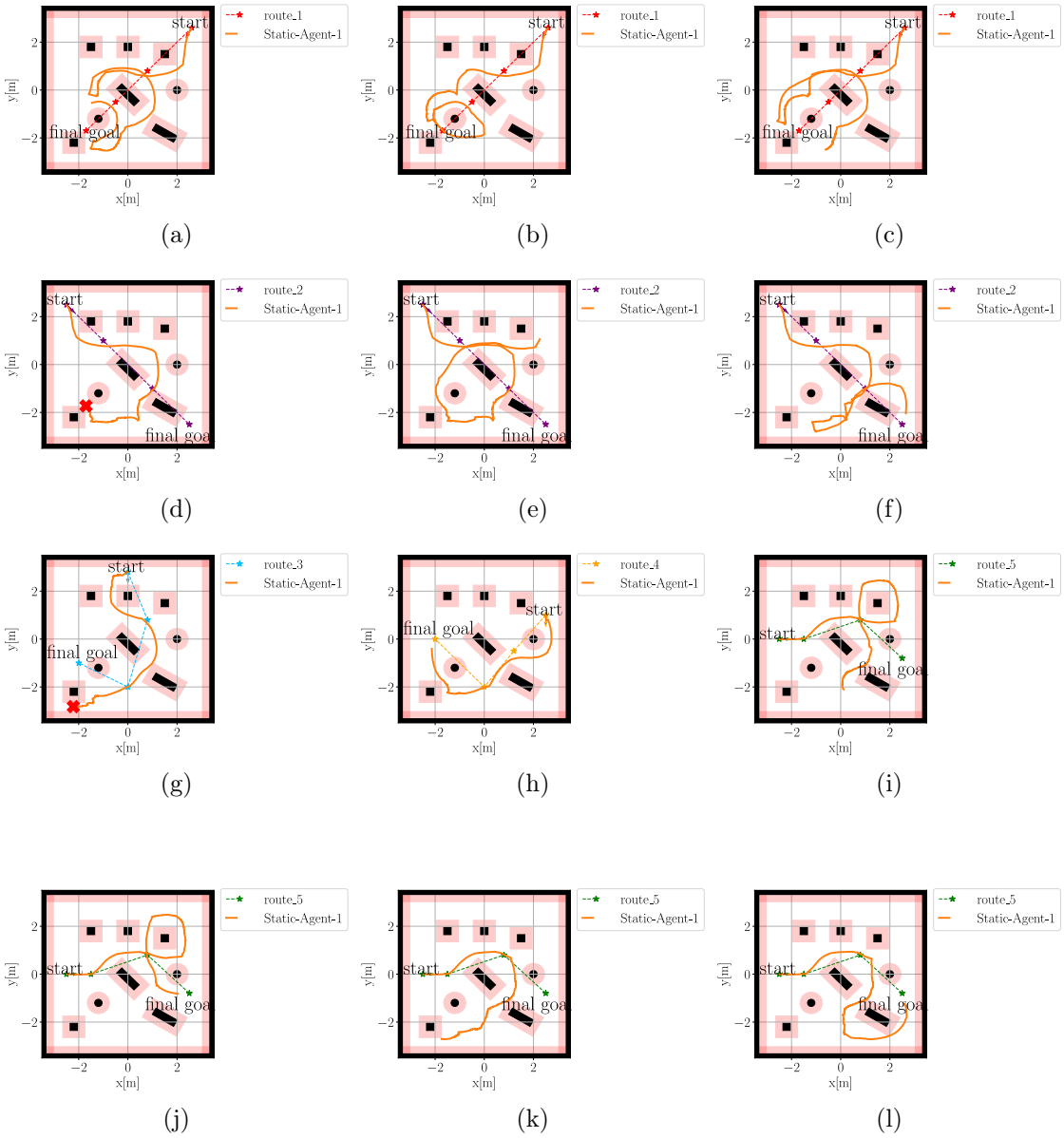


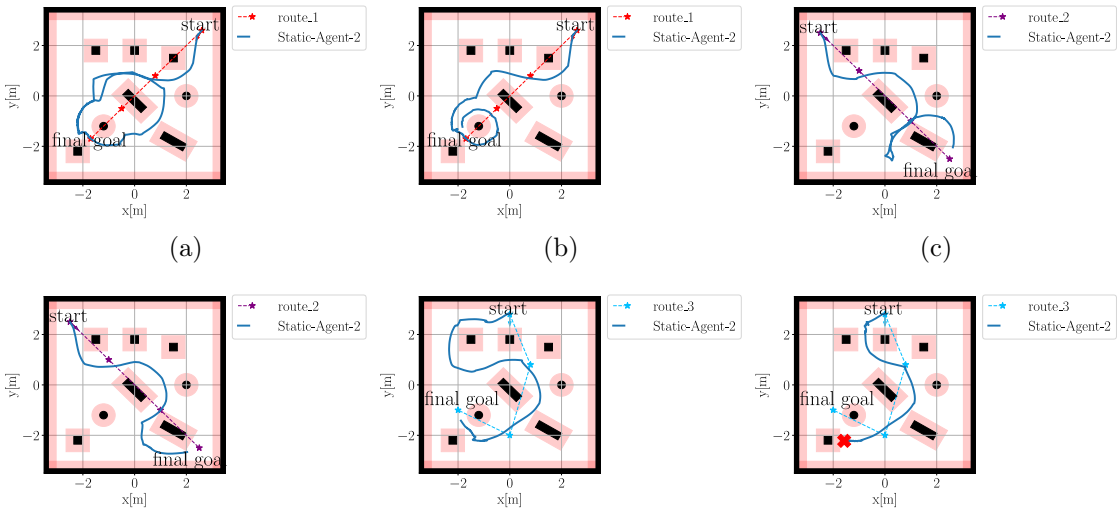
Abbildung 5.16: Die gefolgten Wege von Static-Agent-1 in Hindernis-Konfiguration-2. Das rote Kreuz am Ende eines Weges markiert eine Kollisionsstelle.

Von den drei Agenten hat Static-Agent-2 die niedrigste Erfolgsrate auf Route_1 (siehe Abbildung 5.17a) erreicht, nämlich 0.4. Dies liegt daran, dass der Agent den zweiten Zielpunkt im Gegensatz zu Hindernis-konfiguration-1 nur einmal erreicht hat (siehe Abbildung 5.17b). Wichtig zu erwähnen hier ist, dass der Weg durch den zweiten Zielpunkt

schmaller als in Hindernis-konfiguration-1 ist. In 90% der Versuche auf Route_1 hat der Agent den Weg in Abbildung 5.17a genommen.

Auf Route_2 und Route_5 hat er alle Zielpunkte erreicht. Trotzdem hat er nicht immer den schnelleren Weg genommen. In 90% der Versuche auf Route_2 hat er den Weg in Abbildung 5.17c genommen. Hier hat er, nachdem er das zweite Ziel erreicht hat, zuerst versucht, das nächste Hindernis zu vermeiden. Dann hat er sich dafür entschieden, zurückzufahren und über die andere Seite zum finalen Ziel zu fahren, obwohl der Weg in Abbildung 5.17d nach der Hindernisvermeidung effizienter wäre. Der selbe Fall ist auf Route_5 in Abbildung 5.17k zu erkennen. Auf Route_5 hat er aber trotzdem in 80% der Versuche einen sehr effizienten Weg zwischen den zwei letzten Zielpunkten genommen (siehe Abbildung 5.17j).

Auf Route_3 und Route_4 weist der Agent eine Erfolgsrate von 0.93 und 0.86 auf und alle Kollisionen sind auf diesen beiden Routen auf der selben Stelle aufgetreten (siehe Abbildung 5.17f und 5.17g). Auf Route_3 liegt die Kollisionsrate bei 0.2 und auf Route_4 liegt die bei 0.4.



(a) (b) (c)

(d) (e) (f)

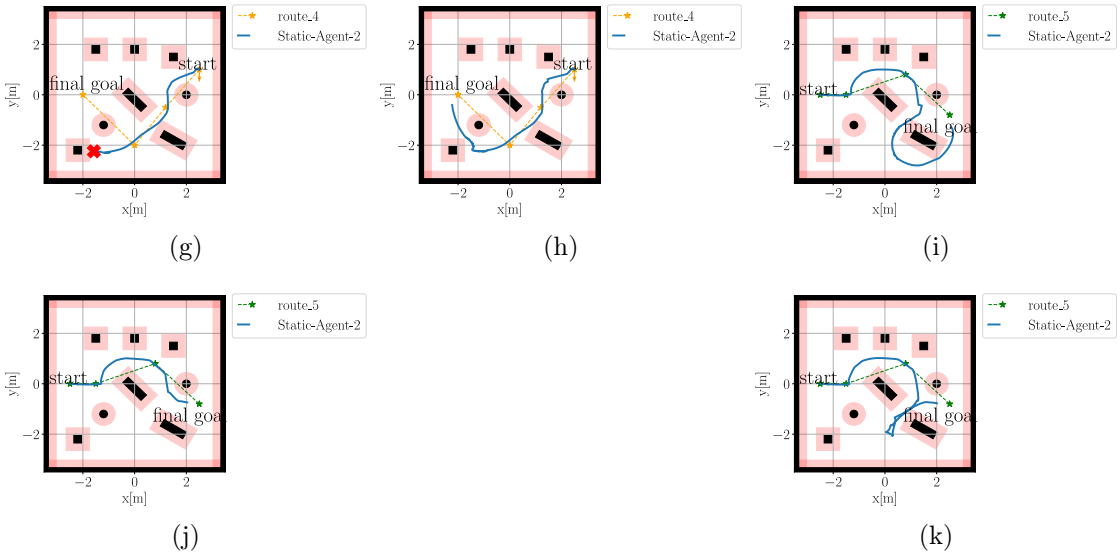


Abbildung 5.17: Die gefolgten Wege von Static-Agent-2 in Hindernis-Konfiguration-2. Das rote Kreuz am Ende eines Weges markiert eine Kollisionsstelle

Von den drei Agenten hat Static-Agent-3 in dieser Konfiguration die besten Ergebnisse gezeigt. Auf Route_2, Route_3 und Route_5 liegt seine Erfolgsrate bei 1. Zudem hat er eine deutlich kleinere Navigationszeit (siehe Tabelle A.6), denn er nimmt insgesamt im Vergleich zu den anderen Agenten immer einen effizienteren Weg zum Ziel. Die einzigen Kollisionen fanden auf Route_4 auf der selben Stelle statt, wobei der Agent 40% der Versuche Kollidiert ist. Die Kollisionsstelle ist in Abbildung 5.18h zu sehen.
Auf Route_1 ist es bei 30% der Versuche zu einer Zeitüberschreitung gekommen. Wenn man jedoch Abbildung 5.18b betrachtet, stellt man fest, dass da auch eine Kollision zu erwarten ist, denn der Agent hat sich in einer Situation begeben, wo er keinen Ausweg hat.

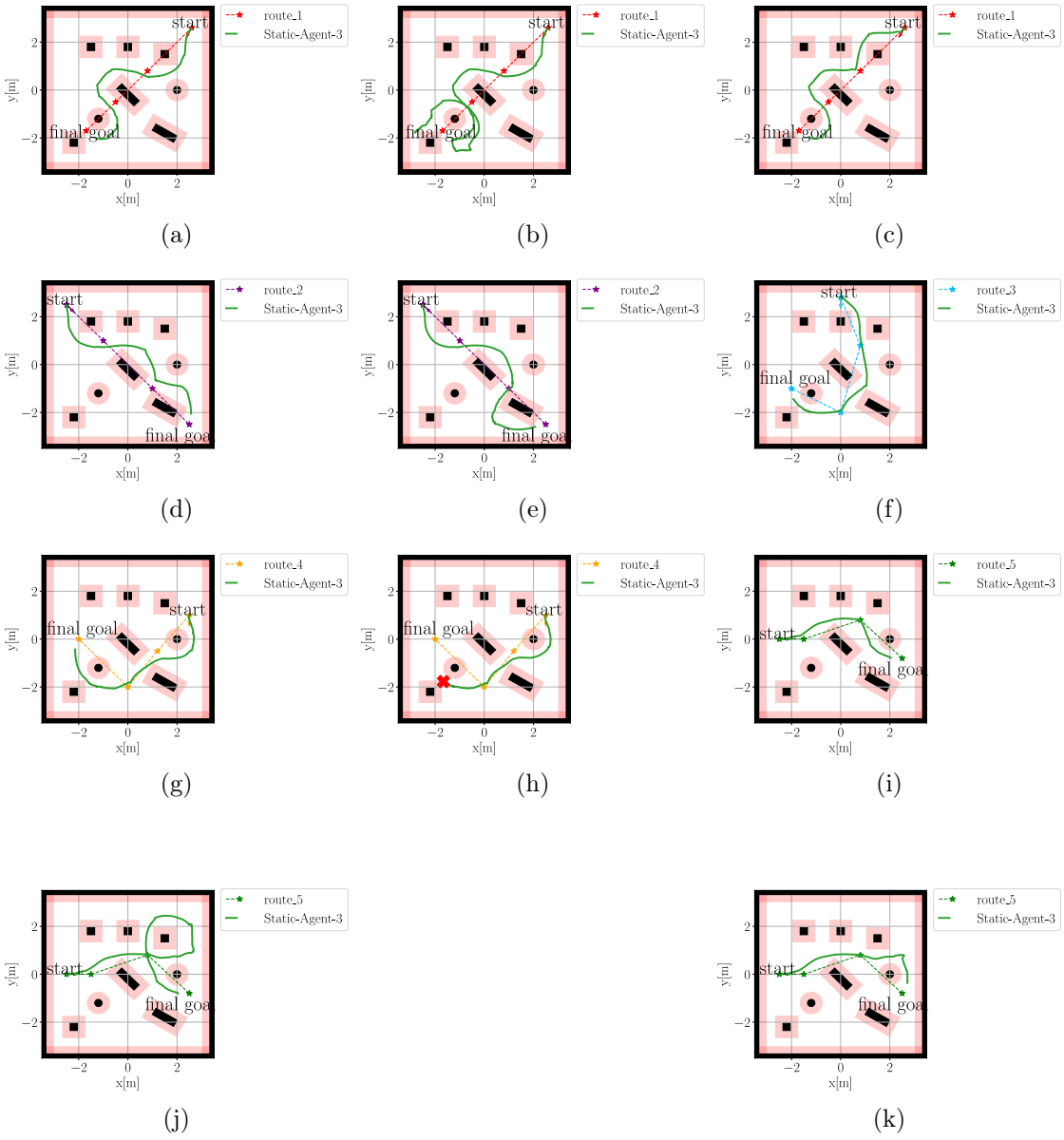


Abbildung 5.18: Die gefolgten Wege von Static-Agent-3 in Hindernis-Konfiguration-2. Das rote Kreuz am Ende eines Weges markiert eine Kollisionsstelle

In Hindernis-Konfiguration-2 wurde festgestellt, dass die Agenten es nicht richtig gelernt haben, enge Bereiche zu vermeiden. Auf Route_1 zum Beispiel wurde erwartet, dass die Agenten den letzten Zielpunkt wegen der Hindernisvermeidung nicht erreichen. Die haben es aber trotzdem versucht, durch den Punkt zu fahren und die minimale Laser-Distanz

dabei war zwischen 0.45 und 0.53 m. Dies wäre im Training eine Kollision gewesen. Die Agenten wurden aber bei der Evaluation toleriert, denn auch auf dem echten Roboter können seitliche Kollisionen (nur seitliche Kollisionen) toleriert werden, solange die minimale Laser-Distanz nicht kleiner als 0.45 m ist.

Bezüglich der Änderung der Geschwindigkeit stellt man fest, dass Static-Agent-1 und Static-Agent-3 auch ohne Einschränkung in der Beschleunigung keine großen Sprünge bei der Änderung machen. Die maximale Änderung der linearen Geschwindigkeit über alle Routen liegt durchschnittlich bei 0.38 m/s für Static-Agent-1 und bei 0.35 m/s für Static-Agent-3. Der Grund für die flüssigere Änderung kann an der Verwendung vom *Generalized State Dependent Exploration (gSDE)* liegen. *gSDE* ist eine Explorationsmethode, die Stable-Baselines3 bietet, um eine flüssige Exploration beim Training zu erreichen. Die Methode ist in [34] beschrieben.

Im statischen Kontext kann man insgesamt festhalten, dass Static-Agent-3 die beste Performanz gezeigt hat.

5.3.2 Dynamische Umgebung

Zusätzlich zur Evaluation während des Trainings (siehe Abbildung 5.11) wird das Verhalten von den Agenten im folgenden weiter untersucht, um zu sehen, was die tatsächlich gelernt haben. Gegenüber den Fußgängern wird Dynamic-Agent-1 mit Static-Agent-1 verglichen. Es wird weiterhin untersucht, warum Dynamic-Agent-2 eine schlechte Performanz hat.

Die Reward-Funktion (siehe Abschnitt 5.1.3) sollte Dynamic-Agent-1 und Dynamic-Agent-2 dazu führen, eine lineare Geschwindigkeit unter 0.1 m/s in der Komfortzone von Fußgängern einzuhalten. Daher werden bei dieser Evaluation folgende Metriken berücksichtigt:

- Die maximale lineare Geschwindigkeit: dadurch kann man feststellen, wie vorsichtig die Agenten in der dynamischen Umgebung fahren. Hier könnte man erwarten, dass die nicht schnell fahren, denn in der Evaluation in der statischen Umgebung wurde festgestellt, dass die Änderung der linearen Geschwindigkeit ohne Limitierung der Beschleunigung bei etwa 0.4 m/s liegt, was an der Verwendung von der *gSDE* liegen könnte (siehe Abschnitt 5.3.1).

- Die Anzahl der Schritte, bei denen die Agenten in und aus der Komfortzone von Fußgängern sind: dadurch kann man eine Aussage darüber treffen, ob die Agenten das Bewegungsmodell von den Fußgängern vorhersehen und es vermeiden, deren Komfortzone zu betreten oder ob sie die Komfortzone betreten, aber die Geschwindigkeitsgrenze nicht überschreiten.
- Die Anzahl von Überschreitungen der Geschwindigkeitsgrenze in der Komfortzone von den Fußgängern.
- Die Anzahl von durch die Fußgänger verursachten Kollisionen: es gibt Fälle, bei denen der Roboter die Geschwindigkeitsgrenze in der Komfortzone einhält, aber ein Fußgänger läuft über ihn.
- Die Anzahl von Versuchen bei denen die Agenten den Zielpunkt erreicht haben.

Bei der Evaluation wird eine Umgebung verwendet, wie in Abbildung 5.9 beschrieben wurde. Zwischen einem Start- und Zielpunkt werden für jeden Agenten 50 Versuche durchgeführt und daraus die eben beschriebenen Metriken ermittelt.

Abbildung 5.19 stellt zwei Beispiele aus der Evaluation von Dynamic-Agent-1 dar:

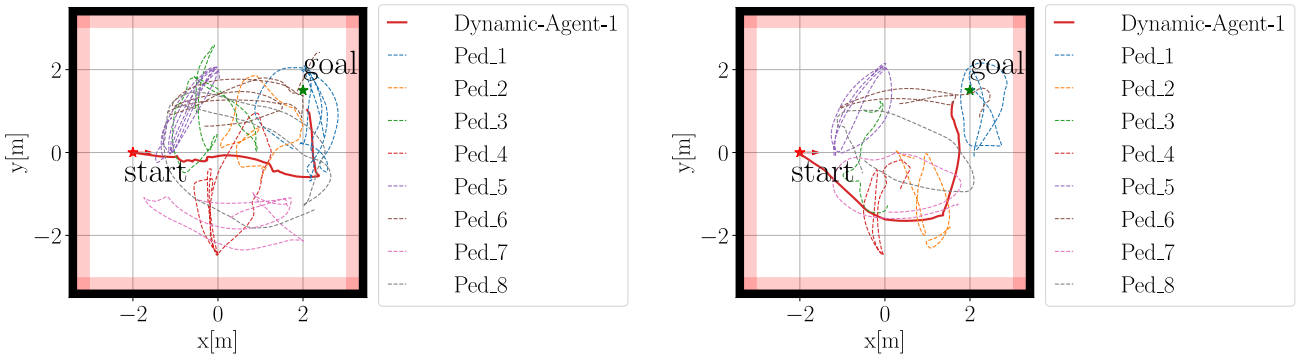


Abbildung 5.19: In den Abbildungen 5.19a und 5.19b sind zwei Fälle zu sehen, in denen die Bewegungsplanung vom Roboter stark von den Fußgängern beinflusst wird. Bei 5.19b vor allem wurde der Roboter von den Fußgängern nach rechts direkt nach dem Start gedrängt. Die beigefügten Videos im Anhang zeigen deutlicher das Verhalten vom Roboter gegenüber den Fußgängern.

In den 50 Versuchen hat Dynamic-Agent-1 insgesamt 3279 Schritte gemacht. Davon war er 3214 Schritte außerhalb der Komfortzone von den Fußgängern und 65 Schritte in deren Komfortzone. Der Agent hat 32 Male den Zielpunkt erreicht, 13 Male die Geschwindigkeitsgrenze in der Komfortzone von den Fußgängern überschritten und 5 Kollisionen wurden von den Fußgängern verursacht. Von den 13 Fällen, in denen die Geschwindigkeitsgrenze überschritten wurde haben 4 Fälle direkt nach dem Start stattgefunden und da der Agent direkt nach dem Start nicht alle Informationen über die dynamichen Hindernisse hat, nämlich die drei letzten Laserscans, sind bei der Überschreitung nur 9 Überschreitungen relevant. Der Agent hat also nur in 9 von den 50 Versuchen die Geschwindigkeitsgrenze in der Komfortzone von den Fußgängern überschritten. Die maximale Geschwindigkeit vom Agenten über die 50 Versuche liegt durchschnittlich bei 0.8 m/s.
Aus diesen Ergebnissen sieht man, dass der Agent anders als die Erwartung eine hohe Geschwindigkeit erreichen konnte. Der hat es gelernt, die Bewegung von den Fußgängern aus der Entfernung zu berücksichtigen und deren Komfortzone zu vermeiden. Der ist daher selten (bei etwa 2% der gesamten Schritte) in deren Komfortzone. Dies kann zudem in den beigefügten Videoaufnahmen beobachtet werden. In der Komfortzone von den Fußgängern hat er zwar 9 Male die Geschwindigkeitsgrenze überschritten, was aber im Vergleich zu der gesamten Anzahl von Schritten in deren Komfortzone (65) ein gutes Ergebnis ist.

Static-Agent-1 hat zudem insgesamt 2275 Schritte gemacht. Davon hat er 62 Schritte in der Komfortzone von den Fußgängern verbracht. Der Agent hat 15 Male den Zielpunkt erreicht, 33 Male die Geschwindigkeitsgrenze überschritten und nur 2 Kollisionen wurden von den Fußgängern verursacht. Von den 33 Fällen, in denen die Geschwindigkeitsgrenze überschritten wurde, sind 2 Fälle direkt nach dem Start stattgefunden und die maximale Geschwindigkeit vom Agenten liegt durchschnittlich auch bei 0.8 m/s.

Wenn man Dynamic-Agent-1 mit Static-Agent-1 vergleicht, stellt man fest, dass Dynamic-Agent-1 besser mit den Fußgängern umgeht. Static-Agent-1 hat in 31 Fällen die Geschwindigkeitsgrenze überschritten im Vergleich zu 9 für Dynamic-Agent-1. In 32 Versuchen hat außerdem Dynamic-Agent-1 den Zielpunkt erreicht, wobei der Wert bei 15 für Static-Agent-1 liegt. In den Videoaufnahmen bei der Evaluation kann man das unterschiedliche Verhalten der Agenten deutlicher erkennen. In vielen Fällen berücksichtigt Dynamic-Agent-1 die Bewegung von den Fußgängern schon früh in seiner Bewegungs-

planung und geht dadurch besser mit denen um. Static-Agent-1 auf der anderen Seite kommt am Ziel meistens nur, wenn der Weg frei ist.

Während die Ergebnisse von Dynamic-Agent-1 in der dynamischen Umgebung gut sind, ist es interessant sein Verhalten in statischen Umgebungen zu evaluieren. Der Agent wurde daher auf Route_1 in der statischen Hindernis-Konfiguration-1 (siehe Abbildung 5.12a) evaluiert. In 10 versuchen hat er durchschnittlich 28.3 s pro Versuch gebraucht, um am finalen Ziel anzukommen und seine maximale Geschwindigkeit liegt durchschnittlich bei 0.93 m/s. Der braucht durchschnittlich etwa nur 3 s länger als Static-Agent-1 und kann gut mit statischen Hindernissen umgehen. Sein gefahrener Weg ist in Abbildung 5.20 dargestellt:

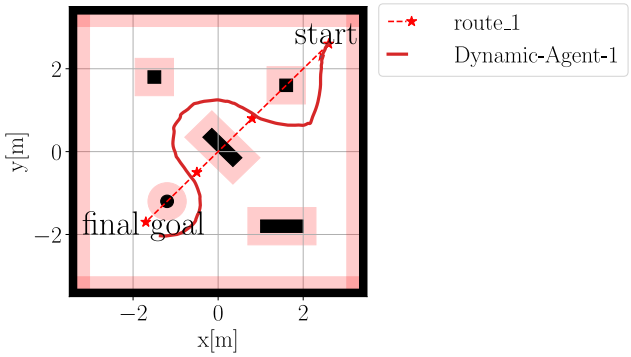


Abbildung 5.20: Dynamic-Agent-1 auf Route_1 in Hindernis-Konfiguration-1

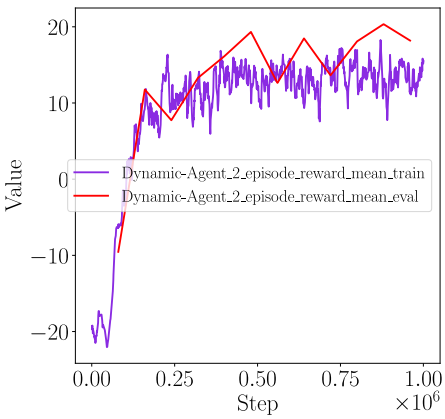
Dynamic-Agent-2 hat in nur 5 Versuchen den Zielpunkt erreicht. Der Agent war 43 Male in der Komfortzone von den Fußgängern und er hat jedes Mal die Geschwindigkeitsgrenze überschritten. Zwei Kollisionen wurden von den Fußgängern verursacht. Obwohl 16 Fälle bei der Überschreitung der Geschwindigkeitsgrenze direkt nach dem Start stattfanden, hatte der Agent Probleme im Umgang mit dynamischen Hindernissen. Der Grund von diesem Problem kann die Sichtweite vom Agenten sein. Der Radius von seinem Wahrnehmungsfeld beträgt 1.5 m im Vergleich zu 3 m für Dynamic-Agent-1. Im Vergleich zu 0.8 m/s bei Dynamic-Agent-1 hatte der Agent in der Evaluation eine maximale Geschwindigkeit von durchschnittlich 0.39 m/s, was wiederum auf die kleine Sichtweite zurückgeführt werden kann.

Static-Agent-3 und Dynamic-Agent-2 haben den selben Beobachtungsraum und im statischen Kontext hat Static-Agent-3 die beste Performanz gezeigt(siehe Tabelle A.5 und A.6).

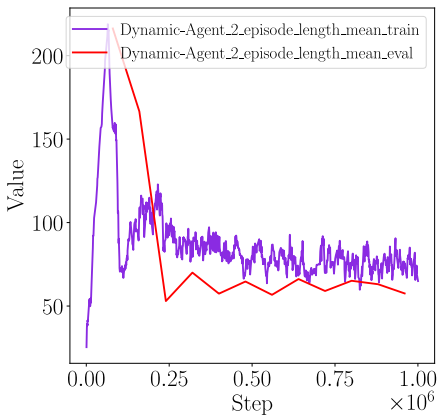
Wenn man aber den dynamischen Fall betrachtet, braucht Dynamic-Agent-2 mindestens die 2 letzten Laserscans, um die Geschwindigkeit von den dynamischen Hindernissen abzuleiten. Mit den 3 letzten Laserscans könnte er deren Beschleunigungen ableiten. Bei der maximal möglichen linearen Geschwindigkeit von 1 m/s könnte der Agent theoritisch nach 0.30 m die Beschleunigung von den Füßgängern ableiten, denn ein Schritt dauert 0.1 s. Das bedeutet, der Agent hätte theoritisch die Möglichkeit drei Schritte, nachdem ein Fußgänger sein Wahrnehmungsfeld betritt, seine Geschwindigkeit unter 0.1 m/s zu setzen, bevor er die Komfortzone vom Fußgänger betritt. Die Bewegung von den Fußgängern im Voraus zu berücksichtigen ist jedoch in diesem Fall schwierig, denn der Agent hat eine knappe Reaktionszeit. Das weitere Problem ist, wenn die Fußgänger nah am Roboter sind, haben sie eine kleine Geschwindigkeit, was es schwer macht, sie von statischen Hindernissen zu unterscheiden. Die kleine Sichtweite ist also ein Problem.

Zur Lösung dieses Problems wurde im Training weitere Versuche mit einer Sichtweite von 2 m und dann 3 m erfolglos durchgeführt. Da die Erhöhung der Sichtweite bei einer Auflösung von 0.05 m die Größe der Karte erhöht, wurde der Agent zuerst in einer statischen Umgebung trainiert, um zu sehen, ob die Ergebnisse mit der neuen Kartengröße in der statischen Umgebung weiterhin gut sind. Durch die Änderung der Kartengröße ist die Performanz jedoch gefallen und das Ergebnis war unbrauchbar. Es wurden weitere Netzarchitekturen ausprobiert, was die Performanz aber nicht verbessert hat.

Ein weiterer Versuch mit Dynamic-Agent-2 bestand darin, die Größe der Karte gleich zu halten und mit einer Auflösung von 0.1 m der Radius vom Wahrnehmungsfeld auf 3 m zu setzen. Der Agent wurde neu trainiert und die Ergebnisse sind in Abbildung 5.21 zu sehen.



(a) Durchschnittlicher Reward



(b) Durchschnittliche Episodenlänge

Abbildung 5.21: Ergebnisse vom neuen Training von Dynamic-Agent-2 mit 3 m als Radius vom Wahrnehmungsfeld

Nach dem neuen Training wurden die Metriken neu evaluiert und dabei wurde eine deutlich bessere Performanz erreicht als beim ersten Training. Der Agent hat diesmal in 33 Versuchen den Zielpunkt erreicht im Vergleich zu 5 beim ersten Training. Die maximale Geschwindigkeit liegt durchschnittlich bei 0.87 m/s. Beim ersten Training lag die bei 0.39 m/s und dies kann die Vermutung im Abschnitt 5.3.1 begründen, dass die kurze Sichtweite die kleine Geschwindigkeit verursacht hat. Der Agent war zudem bei 119 Schritten in der Komfortzone von den Fußgängern im Vergleich zu 65 bei dynamic-Agent-1. Dabei ist er aber meistens langsamer gefahren und hat die Geschwindigkeitsgrenze nur 14 Male überschritten. Bei den Kollisionen fanden 5 direkt nach dem Start statt und 3 wurden von den Fußgängern verursacht.

Wie bei Dynamic-Agent-1 wurde angesichts der guten Performanz das Verhalten vom neu trainierten Dynamic-Agent-2 auf Route_1 in Hindernis-Konfiguration-1 untersucht. In dieser statischen Umgebung sind die Ergebnisse schlecht aufgefallen. In 10 Versuchen ist der Agent 8 Male kollidiert. In den beigefügten Videos im Anhang kann beobachtet werden, dass der Agent in vielen Situationen lange vor den statischen Hindernissen steht, bevor er den freien Weg nimmt. Der Grund dafür könnte sein, dass der Agent erstmal versucht, zu warten, bis der Weg frei wird. Bei der Evaluation in der dynamischen Umgebung kann man auch in den beigefügten Videos beobachten, dass der Agent im Vergleich zu Dynamic-Agent-1 die Bewegung der Fußgänger in vielen Situationen nicht antizipiert.

Der versucht einfach zu warten, bis die Fußgänger den Weg befreien. Die mehreren Kollisionen in der statischen Umgebung könnten an der Erhöhung der Auflösung liegen, denn mit einer Auflösung von 0.1 m kann der Agent Distanzänderungen unter 0.1 m nicht unterscheiden.

6 Fazit

In dieser Arbeit wurde ein Prototyp für den Einsatz von Deep Reinforcement Learning auf dem Husky-Roboter entwickelt. Dabei wurde eine Simulationsumgebung in Pybullet für das Training entworfen und ein 2D-Laserscanner wurde für die Erkennung der Hindernisse verwendet. Es wurden zwei Darstellungsformen von den Laser-Daten erprobt, eine Darstellung in Form einer Karte und die rohen Laserstrahlen. Die Agenten wurden sowohl in statischen als auch in dynamischen Umgebungen trainiert. Die dynamischen Hindernisse wurden durch Fußgänger modelliert, die nach dem Social-Force-Model gesteurt sind.

Im Experiment wurden ein paar RL-Algorithmen ausprobiert, unter anderem PPO, DDPG, SAC. Der SAC Algorithmus wurde in der Arbeit verwendet, denn er hat die besten Ergebnisse geliefert und er hat sich weniger sensibel zu Hyperparametern erwiesen.

Wie die Ergebnisse es zeigen, waren die Agenten beim Training in der statischen Konfiguration großten Teils in der Lage, die Hindernisse zu vermeiden und zum Ziel zu fahren. In dieser Konfiguration konnte zudem gezeigt werden, dass Static-Agent-2 trotz einer expliziten Limitierung der Beschleunigung ein richtiges Verhalten lernen konnte. Diese Limitierung kann bei Systemen nützlich sein, die stark oszillieren. Zudem hat Static-Agent-3 mit der bildlichen Darstellung von den Laser-Daten das beste Ergebnis gezeigt. Bei Umgebungen mit vielen Hindernissen konnte festgestellt werden, dass die Agenten in vielen Situationen es nicht gelernt haben, sehr enge Bereiche zu vermeiden. Die fahren in solchen Bereichen zwar sehr langsam, aber das bessere Verhalten wäre, Bereiche zu vermeiden, die sehr eng sind, oder in denen der Roboter keinen Ausweg hat.

Im dynamischen Kontext konnte man den Einfluss von den Fußgängern auf der Bewegungsplanung von Dynamic-Agent-1 erkennen. Der Agent konnte die Bewegung von den Fußgängern in seiner Bewegungsplanung berücksichtigen.

Static-Agent-3 und Dynamic-Agent-2 haben den selben Beobachtungsraum aber obwohl Static-Agent-3 beim Training im statischen Kontext gute Ergebnisse gezeigt hat, hatte

Dynamic-Agent-2 große Probleme dabei, mit Fußgängern umzugehen. Die Probleme liegen an seinem kleinen Wahrnehmungsfeld. Nach Erhöhung der Auflösung von der Karte von 0.05 m auf 0.1 m und somit der Radius vom Wahrnehmungsfeld von 1.5 m auf 3 m konnten Verbesserungen gegenüber Füßgängern erreicht werden, aber durch die Erhöhung der Auflösung ist der Agent öfter gegen statische Hindernisse Kollidiert.

Angesichts der Ergebnisse kann man festhalten, dass das Projekt konzeptionell erfolgreich durchgeführt wurde. Durch den Prototyp konnte gezeigt werden, dass die Agenten trotz einiger Limitationen direkt aus den Laser-Daten es lernen können, kollisionsfrei und zeiteffizient zu navigieren. Die Limitationen vom Prototyp konnten zudem erkannt werden, die für weitere Verbesserungen wichtig sind.

6.1 Ausblick

Für weitere Verbesserungen ist es notwendig, weitere Tests durchzuführen, um weitere Situationen zu identifizieren, in denen die Agenten fehlschlagen. Nachdem diese Situationen identifiziert wurden, sollten weitere Trainings unter Berücksichtigung dieser Situationen bei der Modellierung der Umgebung durchgeführt werden. Die Agenten sollten vor allem im Training engen Bereichen ausgesetzt werden, damit sie es lernen, die zu vermeiden. Für Dynamic-Agent-2 sollte ebenfalls bei einer Auflösung von 0.05 m die Sichtweite erhöht werden, und weitere Netze sollten modelliert werden, die aus größeren Bildern das gewünschte Verhalten der Agenten in sowohl statischen als auch dynamischen Umgebungen lernen können.

Für weitere Arbeiten, ist es außerdem zu empfehlen, im dynamischen Kontext Recurrent-Neural-Networks, LSTM Netze zum Beispiel, auszuprobieren, die sich gut eignen, um temporale Informationen zu extrahieren. Zudem wäre es interessant, verschiedene Verhalten von Fußgängern zu untersuchen, wie Fußgänger, die den Roboter beachten und Platz für ihn machen, Fußgänger, die den Roboter nicht ausweichen oder Gruppen von Fußgängern und dabei verschiedene Geschwindigkeiten von den Fußgängern auszuprobieren.

Die Agenten sollten abschließend auf der realen Hardware getestet werden. Es wurden im Rahmen der Arbeit ein paar Tests auf dem realen Roboter durchgeführt. Wegen technischen Problemen auf dem Roboter konnten die Tests nicht zu Ende durchgeführt werden

und die Ergebnisse wurden daher in der Arbeit nicht übernommen. Es konnte aber festgestellt werden, dass die in der Pybullet-Umgebung trainierten Agenten ohne weitere Anpassungen auf dem realen Husky-Roboter mit ROS unter Verwendung der Odometry für die Lokalization in vielen Situationen erfolgreich waren. Daher lohnt es sich, Pybullet für weitere Arbeiten zu verwenden. Das Training mit Pybullet in parallelen Umgebungen ist zudem wesentlich schnell aber die Evaluation während des Trainings könnte das Training verlangsamen, wenn das verwendete Trainingsframework keine Evaluation in parallelen Umgebungen unterstützt.

Literaturverzeichnis

[1] URL https://www.kiwibot.com/. – [Online; accessed 31-07-2022]

[2] URL https://www.ottonomy.io/. – [Online; accessed 31-07-2022]

[3] URL https://avidbots.com/meet-neo/. – [Online; accessed 31-07-2022]

[4] URL http://wiki.ros.org/diff_drive_controller. – [Online; accessed 14-10-2022]

[5] *PyBullet Quickstart Guide*. – URL https://docs.google.com/document/d/10sXEhzFRSnvFcl3XxNGhnD4N2SedqwdAvK3dsihxVUA/edit#. – [Online; accessed 09-12-2022]

[6] Advantech: *Autonomous mobile robot market expanding at record pace*. 2020. – URL https://www.controleng.com/articles/autonomous-mobile-robot-market-expanding-at-record-pace/. – [Online; accessed 31-07-2022]

[7] Advantech: *Smart 5G Patrol Robots Equipped with Advantech's MIC-770 Edge Computer Deployed to Fight Coronavirus*. 2020. – URL https://www.advantech.eu/resources/news/smart-5g-patrol-robots-equipped-with-advantech%E2%80%99s-mic-770-edge-computer-deployed-to-fight-coronavirus. – [Online; accessed 31-07-2022]

[8] A/S, Mobile Industrial R.. – URL https://www.mobile-industrial-robots.com/solutions/robots/mir100/. – [Online; accessed 31-07-2022]

[9] Berg, Jur van den ; Lin, Ming ; Manocha, Dinesh: Reciprocal Velocity Obstacles for real-time multi-agent navigation. In: *2008 IEEE International Conference on Robotics and Automation*, 2008, S. 1928–1935

[10] Billy Okal, Timm L.. – URL https://github.com/spencer-project. – [Online; accessed 15-10-2022]

[11] Borenstein, J. ; Koren, Y.: Real-time obstacle avoidance for fast mobile robots. In: *IEEE Transactions on Systems, Man, and Cybernetics* 19 (1989), Nr. 5, S. 1179–1187

[12] Borenstein, J. ; Koren, Y.: Real-time obstacle avoidance for fast mobile robots in cluttered environments. In: *Proceedings., IEEE International Conference on Robotics and Automation*, 1990, S. 572–577 vol.1

[13] Borenstein, J. ; Koren, Y.: The vector field histogram-fast obstacle avoidance for mobile robots. In: *IEEE Transactions on Robotics and Automation* 7 (1991), Nr. 3, S. 278–288

[14] Cardona, Manuel ; Palma, Allan ; Manzanares, Josué: COVID-19 Pandemic Impact on Mobile Robotics Market. In: *2020 IEEE ANDESCON*, 2020, S. 1–4

[15] Chen, Changan ; Liu, Yuejiang ; Kreiss, Sven ; Alahi, Alexandre: *Crowd-Robot Interaction: Crowd-aware Robot Navigation with Attention-based Deep Reinforcement Learning*. 2018. – URL https://arxiv.org/abs/1809.08835

[16] Chen, Yu F. ; Everett, Michael ; Liu, Miao ; How, Jonathan P.: Socially aware motion planning with deep reinforcement learning. In: *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, S. 1343–1350

[17] Chen, Yu F. ; Everett, Michael ; Liu, Miao ; How, Jonathan P.: *Socially Aware Motion Planning with Deep Reinforcement Learning*. 2017. – URL https://arxiv.org/abs/1703.08862

[18] Choi, Baehoon ; Kim, Beomseong ; Kim, Euntai ; Yang, Kwang W.: A modified dynamic window approach in crowded indoor environment for intelligent transport robot. In: *2012 12th International Conference on Control, Automation and Systems*, 2012, S. 1007–1009

[19] Choi, Jaewan ; Lee, Geonhee ; Lee, Chibum: *Reinforcement learning-based dynamic obstacle avoidance and integration of path planning*. 2021. – URL https://doi.org/10.1007/s11370-021-00387-2

[20] Cui, Yuxiang ; Zhang, Haodong ; Wang, Yue ; Xiong, Rong: *Learning World Transition Model for Socially Aware Robot Navigation*. 2020. – URL https://arxiv.org/abs/2011.03922

[21] Everett, Michael ; Chen, Yu F. ; How, Jonathan P.: *Motion Planning Among Dynamic, Decision-Making Agents with Deep Reinforcement Learning*. 2018. – URL https://arxiv.org/abs/1805.01956

[22] Eysenbach, Benjamin ; Levine, Sergey: Maximum Entropy RL (Provably) Solves Some Robust RL Problems. In: *CoRR* abs/2103.06257 (2021). – URL https://arxiv.org/abs/2103.06257

[23] Fox, D. ; Burgard, W. ; Thrun, S.: The dynamic window approach to collision avoidance. In: *IEEE Robotics & Automation Magazine* 4 (1997), Nr. 1, S. 23–33

[24] Haarnoja, Tuomas ; Zhou, Aurick ; Abbeel, Pieter ; Levine, Sergey: *Soft Actor-Critic: Off-Policy Maximum Entropy Deep Reinforcement Learning with a Stochastic Actor*. 2018. – URL https://arxiv.org/abs/1801.01290

[25] Han, Ruihua ; Chen, Shengduo ; Wang, Shuaijun ; Zhang, Zeqing ; Gao, Rui ; Hao, Qi ; Pan, Jia: Reinforcement Learned Distributed Multi-Robot Navigation With Reciprocal Velocity Obstacle Shaped Rewards. In: *IEEE Robotics and Automation Letters* 7 (2022), Nr. 3, S. 5896–5903

[26] Helbing, Dirk ; Molnár, Péter: Social force model for pedestrian dynamics. In: *Physical Review E* 51 (1995), may, Nr. 5, S. 4282–4286. – URL https://doi.org/10.1103%2Fphysreve.51.4282

[27] Lapan, Maxim: *Deep Reinforcement Learning, Das umfassende Praxis-Handbuch*. mitp Verlags GmbH & Co. KG, Frechen, 2020. – ISBN 978-3-7475-0036-1

[28] Li, Keyu ; Xu, Yangxin ; Wang, Jiankun ; Meng, Max Q.-H.: SARL: Deep Reinforcement Learning based Human-Aware Navigation for Mobile Robot in Indoor Environments. In: *2019 IEEE International Conference on Robotics and Biomimetics (ROBIO)*, 2019, S. 688–694

[29] Long, Pinxin ; Fan, Tingxiang ; Liao, Xinyi ; Liu, Wenxi ; Zhang, Hao ; Pan, Jia: *Towards Optimally Decentralized Multi-Robot Collision Avoidance via Deep Reinforcement Learning*. 2017. – URL https://arxiv.org/abs/1709.10082

[30] Looi, Chen Z. ; Ng, Danny Wee K.: A Study on the Effect of Parameters for ROS Motion Planer and Navigation System for Indoor Robot. In: *International Journal of Electrical and Computer Engineering Research* 1 (2021), Jun., Nr. 1, S. 29–36. – URL https://ijecer.org/ijecer/article/view/21

[31] Mang, Maximilian ; Schönherr, Nils: *Husky*. 2020. – URL https://autosys-lab.de/platforms/2020husky/. – [Online; accessed 28-01-2023]

[32] Okal, Billy ; Linder, Timm ; Vasquez, Dizan ; Wehner, Sven ; Islas, Omar ; Palmieri, Luigi. – URL https://github.com/srl-freiburg/pedsim_ros. – [Online; accessed 15-10-2022]

[33] Raffin, Antonin ; Hill, Ashley ; Gleave, Adam ; Kanervisto, Anssi ; Ernestus, Maximilian ; Dormann, Noah: *Stable-Baselines3: Reliable Reinforcement Learning Implementations*. 2021. – URL https://elib.dlr.de/146386/1/20-1364.pdf. – [Online; accessed 21-10-2022]

[34] Raffin, Antonin ; Stulp, Freek: Generalized State-Dependent Exploration for Deep Reinforcement Learning in Robotics. In: *CoRR* abs/2005.05719 (2020). – URL https://arxiv.org/abs/2005.05719

[35] Regier, Peter ; Gesing, Lukas ; Bennewitz, Maren: DEEP REINFORCEMENT LEARNING FOR NAVIGATION IN CLUTTERED ENVIRONMENTS. (2020). – URL https://www.hrl.uni-bonn.de/publications/papers/regier20cmla.pdf. – [Online; accessed 22-10-2022]

[36] Richard S. Sutton, Andrew G. B.: *Reinforcement Learning: An Introduction, second edition, in progress*. The MIT Press, 2014,2015. – URL https://inst.eecs.berkeley.edu/~cs188/sp20/assets/files/SuttonBartoIPRLBook2ndEd.pdf. – [Online; accessed 02-10.2022]

[37] Richard S. Sutton, Andrew G. B.: *Reinforcement Learning, An Introduction, second edition*. The MIT Press, 2018. – URL https://www.andrew.cmu.edu/course/10-703/textbook/BartoSutton.pdf. – [Online; accessed 18-09.2022]. – ISBN 978-0-262-19398-6

[38] Robotics, Clearpath: *HUSKY*. – URL https://github.com/husky/husky. – [Online; accessed 14-10-2022]

[39] Robotics, Clearpath: *HUSKY A200 UNMANNED GROUND VEHICLE*. – URL https://www.clearpathrobotics.com/wp-content/uploads/2013/02/HUSKY_A200_UGV_2013_TEASER_email.pdf. – [Online; accessed 31-07-2022]

[40] Scott, Sean: *Meet Scout*. 2019. – URL https://www.aboutamazon.com/news/transportation/meet-scout. – [Online; accessed 31-07-2022]

[41] Sutera, Enrico ; Mazzia, Vittorio ; Salvetti, Francesco ; Fantin, Giovanni ; Chiaberge, Marcello: Indoor Point-to-Point Navigation with Deep Reinforcement Learning and Ultra-wideband. In: *CoRR* abs/2011.09241 (2020). – URL https://arxiv.org/abs/2011.09241

[42] Tianyu, LIU ; Ruixin, YAN ; Guangrui, WEI ; Lei, SUN: Local Path Planning Algorithm for Blind-guiding Robot Based on Improved DWA Algorithm. In: *2019 Chinese Control And Decision Conference (CCDC)*, 2019, S. 6169–6173

A Anhang

A.1 Simulation- und Reward-parameter

Parameter	Beschreibung	Wert
r	Radius eines Rades	0.1775 m
b	Breite vom Roboter	0.383 m
s	Skalierungsfaktor in der Formel 4.2	1.34
R_s	Schritt-Reward	-0.02
col_{dist}	Kollisionsdistanz	0.53 m
$l_{critical}$	Schwellwert für die Deaktivierung vom Reward zum Ziel	0.8 m
ped_{radius}	Radius der Komfortzone von Fußgängern	1 m
$goal\_radius$	Ziel-Radius	0.5 m
$r_{collision}$	Kollision-Reward	-20
r_{goal}	Ziel-Reward	15
w	Gewichtungsfaktor von der Distanz-Änderung zum Ziel	5

Tabelle A.1: Simulation- und Reward-Parameter

A.2 Trainingsparameter

Parameter	Beschreibung	SA-1/SA-2/DA-1	SA-3/DA-2
learning_rate	Learning rate	0.001 (scheduled)	7.3e-4
buffer_size	Size of the replay buffer	1000000	300000
batch_size	Minibatch size for each gradient update	256	256
ent_coef	Entropy regularization coefficient	auto	auto
gamma	Discount factor	0.99	0.98
tau	Soft update coefficient	0.02	0.02
train_freq	Update frequency of the model	10	64
gradient_steps	Number of gradient step after each rollout	10	64
learning_starts	Number of step before starting learning	1000	1000
use_sde	Use generalized State Dependent Exploration (gSDE)	True	True
use_sde_at_warmup	Use gSDE during warmup phase	True	True
log_std_init	Initial value for the log standard deviation	-2	-2

Tabelle A.2: Stable-Baselines3 Parameter beim Training von den Agenten: Static-Agent-1(SA-1), Static-Agent-2(SA-2), etc.

A.3 Spezifikationen von der Trainings und Evaluationsplatform

Betriebsystem	Ubuntu 20.04.4 LTS
Prozessor	AMD Ryzen 5 5600H (6 Kerne, 16 Threads, 3.20 GHz bis zu 4.40 GHz)
RAM	2 × 8 GiB SODIMM DDR4 3200 MHz
Grafikkarte	NVIDIA GeForce RTX 3060 Laptop GPU 6 GB GDDR6

Tabelle A.3: Hardware und Software Spezifikationen

A.4 Metriken bei der Evaluation von den Agenten

Route	S_{rate}			$t_{\text{nav}}[s]$			$\max(v)[m/s]$			$\max(\Delta v)[m/s]$			C_{rate}			N_{timeout}		
	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3
1	1	1	1	6.73	7.67	8.59	0.94	0.92	0.96	0.44	0.13	0.30	0	0	0	0	0	0
2	1	1	1	8.86	11.14	10.01	0.94	0.93	0.96	0.44	0.13	0.44	0	0	0	0	0	0
3	1	1	1	9.78	11.72	13.78	0.96	0.91	0.98	0.36	0.11	0.31	0	0	0	0	0	0
4	1	1	1	9.56	11.22	13.37	0.92	0.87	0.96	0.28	0.10	0.30	0	0	0	0	0	0
5	1	1	1	8.02	9.92	10.65	0.92	0.81	0.95	0.41	0.14	0.37	0	0	0	0	0	0

Tabelle A.4: Ergebnisse der Evaluation von Static-Agent-1 (A1), Static-Agent-2 (A2) und Static-Agent-3 (A3) im leeren Feld

Route	S_{rate}			$t_{\text{nav}}[s]$			$\max(v)[m/s]$			$\max(\Delta v)[m/s]$			C_{rate}			N_{timeout}		
	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3
1	1	1	1	25.10	24.37	40.87	0.94	0.91	0.98	0.39	0.15	0.37	0	0	0	0	0	0
2	1	1	1	33.45	23.86	23.38	0.96	0.97	0.93	0.35	0.15	0.30	0	0	0	0	0	0
3	1	1	1	12.46	16.55	17.49	1.01	0.91	0.94	0.34	0.15	0.32	0	0	0	0	0	0
4	1	1	1	10.54	12.05	19.63	0.99	0.94	0.98	0.31	0.13	0.30	0	0	0	0	0	0
5	1	1	1	9.95	11.95	12.35	0.95	0.93	0.96	0.37	0.14	0.38	0	0	0	0	0	0

Tabelle A.5: Ergebnisse der Evaluation von Static-Agent-1 (A1), Static-Agent-2 (A2) und Static-Agent-3 (A3) in Hindernis-Konfiguration-1

Route	S_{rate}			$t_{\text{nav}}[s]$			$\max(v)[m/s]$			$\max(\Delta v)[m/s]$			C_{rate}			N_{timeout}		
	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3	A1	A2	A3
1	0.67	0.4	0.9	124.70	126.85	81.36	0.96	0.99	0.92	0.41	0.15	0.39	0	0	0	7	9	3
2	0.90	1	1	104.468	55.99	26.12	0.99	0.93	0.93	0.40	0.15	0.37	0.1	0	0	2	0	0
3	0.67	0.93	1	85.72	76.80	33.86	1.0	0.93	0.94	0.36	0.15	0.35	1	0.2	0	0	0	0
4	1	0.86	0.87	53.35	54.68	49.77	0.99	0.89	0.96	0.40	0.14	0.35	0	0.4	0.4	0	0	0
5	0.77	1	1	116.76	22.58	43.98	0.97	0.85	0.98	0.40	0.15	0.37	0	0	0	7	0	0

Tabelle A.6: Ergebnisse der Evaluation von Static-Agent-1 (A1), Static-Agent-2 (A2) und Static-Agent-3 (A3) in Hindernis-Konfiguration-2

A.5 Ordnerstruktur der zugehörigen CD

```
Basisberzeichnis der CD
├── quellecode
│   └── Quellecode des Trainings und der Evaluation
├── logs_and_images
│   ├── evaluation
│   │   └── Logdateien und Plots vom gefolgten Weg vom Roboter wäh-
│   │       rend der Evaluation
│   └── training
│       └── Logdateien und Plots vom Trainingsverlauf
├── weights
│   └── Trainierte Gewichte für die jeweiligen Agenten
├── videos
│   └── Videoaufnahmen während der Evaluation der dynamischen Agenten
└── thesis.pdf
```

Glossar

Blender Ein Open-Source 3D-Computergrafik-Tool.

Feature-Extractor Ein neuronales Netz, das Muster aus Daten extrahiert.

Gazebo Ein Open-Source 3D Robotik-Simulator.

HAW Hamburg Die HAW Hamburg ist die formalige Fachhochschule am Berliner Tor.

Policy-Network Ein neuronales Netz, das die Geschwindigkeit vom Roboter kontrolliert.

Pybullet Python-Modul für die Simulation von Robotern und für Machine-Learning Aufgaben.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen 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