Universität Augsburg

Fakultät für Angewandte Informatik

Masterarbeit

Lokale Navigation von Mikromobilitätsfahrzeugen mittels Reinforcement Learning

---

vorgelegt von:	Marco Tröster
eingereicht am:	31. 07. 2023
Studiengang:	Informatik (M.Sc.)
Anfertigung am Lehrstuhl:	Lehrstuhl für Mechatronik
	Fakultät für Angewandte Informatik
1. Gutachter:	Prof. Dr.-Ing. Lars Mikelsons
2. Gutachter:	Prof. Dr. Jörg Hähner
Wissenschaftlicher Betreuer:	Lennart Luttkus, M.Sc.

Kurzfassung

Diese Arbeit zeigt, wie Mikromobilitätsfahrzeuge, z.B. E-Scooter, die sichere Navigation durch belebte Zonen mittels autonomer Fahrsoftware erlernen können. Da entsprechende Fahrzeuge unter anderem auf dem Gehweg oder in Fußgängerzonen unterwegs sind, wird zunächst eine geeignete Simulationsumgebung für die Interaktion zwischen Fahrzeug und Fußgängern anhand des Social Force Modells entwickelt. Darauf aufbauend werden Fahrverhaltensweisen mittels Deep Reinforcement Learning trainiert. Anhand von Unfallmetriken wird abschließend eine Qualitätssicherung des erlernten Fahrverhaltens auf Kartenmaterial des virtuellen Universitätscampus durchgeführt und ein Vergleich der Unfallzahlen mit ähnlichen Projekten aus der Literatur vorgenommen.

Unter anderem kann der bestehende Ansatz *RobotSF* [1] um die Simulation von Fußgängerzonen und Gehwegen erweitert werden, wobei eine 19-fache Beschleunigung der für die Simulationslogik benötigten Rechenzeit erzielt wird. Durch die verbesserte Effizienz ist es möglich, echtes Kartenmaterial während des Trainings und der Evaluation zu verwenden. Zudem führen die Lernexperimente mit einer deutlich vereinfachten Belohnungsstruktur zu vielen, interessanten Fahrverhaltensweisen, die Kollisionen mit Fußgängern in dichten Menschenmengen durch das Erlernen einer defensiven Strategie komplett vermeiden können.

Inhaltsverzeichnis

1 Einleitung

Aufgrund der um mehrere Magnituden potenteren Rechenkapazitäten moderner Hardware finden Tiefe Neuronale Netze in vielerlei Gebieten des täglichen Lebens breite Anwendung. Unter anderem werden Neuronale Netze in der Bild- und Spracherkennung und beim Übersetzen und Generieren von Texten und Bildern eingesetzt. Auch für komplexe Aufgaben der Mess- und Regelungstechnik wie z.B. die autonome Steuerung von Fahrzeugen sind Neuronale Netze hervorragend geeignet. Mithilfe von Methoden des Bestärkenden Lernens (engl. Reinforcement Learning) können interessante Verhaltensweisen in Simulationen erlernt und anschließend in der Realität erprobt werden. Für die autonome Fortbewegung der Zukunft ist neben klassischen Automobilen vor allem die Mikromobilitätsklasse mit E-Scootern oder kleinen Lieferrobotern aufgrund ihrer vielseitigen Anwendungsmöglichkeiten interessant. Wegen der anspruchsvollen Interaktion mit Fußgängern ist aktuell das Fahren von E-Scootern in Fußgängerzonen und auf Gehwegen laut StVO verboten [2]. Da jedoch auf vielen Werksgeländen und in der Lagerlogistik sehr wohl autonome Fahrzeuge in fußgängerbelebten Zonen zum Einsatz kommen [3], scheint eine breite Erprobung von Mikromobilitätsfahrzeugen in Fußgängerzonen und auf Gehwegen nur eine Frage der Zeit zu sein. Die Entwicklung autonomer Fahrsoftware kann die sichere Nutzung von Mikromobilität entscheidend vorantreiben.

Um die Herangehensweise an die Entwicklung autonomer Fahrsoftware besser zu verstehen, muss zunächst das Gebiet das Autonomen Fahrens näher betrachtet werden. Es handelt sich im Wesentlichen um die beiden Aufgabenbereiche der Lokalen bzw. Globalen Navigation. Hierbei entspricht die Globale Navigation einem Routenplaner wie beispielsweise einem Navigationsgerät, das kürzeste Wege vom aktuellen Standpunkt zum Zielort berechnet. Hingegen fallen in den Aufgabenbereich der Lokalen Navigation alle Vorgänge, die sonst ein menschlicher Fahrer übernommen hätte. Eine Fahrsoftware muss demnach das Fahrzeug beschleunigen bzw. abbremsen und lenken. Zudem beobachtet sie die anderen Verkehrsteilnehmer und schätzt deren Bewegungen ein, um Kollisionen zu vermeiden. Des weiteren müssen Verkehrszeichen und Ampeln erkannt

und hinsichtlich der Einhaltung der Verkehrsregeln interpretiert werden. Da die Globale Navigation anhand von Kartenmaterial bereits sehr gut erforscht ist und effiziente Navigationsalgorithmen zum Finden kürzester Wege z.B. in Form von Heuristiken wie dem A* Algorithmus zur Verfügung stehen, ist vor allem das Feld der Lokalen Navigation mit Neuronalen Netzen zur Verarbeitung hochdimensionaler Sensordaten interessant.

Zur Entwicklung autonomer Fahrsoftware existieren bereits zahlreiche Forschungsprojekte, unter anderem die Ansätze *RobotSF* [1] und *Multi-Robot* [4], [5]. Vor allem *RobotSF* ist interessant, da bei diesem Ansatz Fußgänger durch das Social Force Modell [6], [7] gesteuert werden und mit einem steuerbaren Fahrzeug interagieren. Aufgrund der unakzeptabel hohen Kollisionsraten mit Fußgängern von 13% bei niedriger und 47% bei hoher Verkehrsdichte ist der Ansatz jedoch noch stark verbesserungsfähig. Daher wurde zu Beginn dieser Arbeit der Austausch mit den Forschern aus Triest gesucht, die hinter der Entwicklung von *RobotSF* stehen. Es wurde berichtet, dass die Lernexperimente vermutlich noch Fortschritte gezeigt hätten, aber aufgrund der sehr langen Trainingszeiten von teilweise über einem Monat kaum durchführbar waren und vorzeitig abgebrochen werden mussten. Um das Experimentieren mit möglichst vielen Ansätzen im Rahmen dieser und folgender Arbeiten zu ermöglichen, konzentriert sich diese Arbeit neben der Verbesserung der Verkehrssicherheit deshalb auch auf die Verbesserung der Effizienz der Simulationsumgebung. Dies umfasst sowohl die Optimierung der Simulationsberechnungen als auch den Einsatz effektiverer Trainingsverfahren wie Proximal Policy Optimization [8]. Dadurch sind schon nach Simulations- bzw. Trainingszeiten von wenigen Stunden sehr gute Ergebnisse erwartbar.

Es soll zunächst die effiziente Erlernbarkeit sicherer Fahrverhaltensweisen bezüglich der Lokalen Navigation von Mikromobilitätsfahrzeugen mittels Reinforcement Learning demonstriert werden. Anschließend wird die Qualität der erlernten Verhaltensweisen anhand von geeigneten Unfallmetriken evaluiert und mit ähnlichen Arbeiten verglichen.

2 Grundlagen

Dieser Abschnitt behandelt Grundlagen bezüglich kinematischer Fahrzeugmodelle und dem Social Force Modell, um diese später zur Simulation der Interaktion zwischen einem Mikromobilitätsfahrzeug und Fußgängern zu vereinen. Da eine Fahrsoftware für ein steuerbares Fahrzeug entstehen soll, wird außerdem auf die Grundlagen zum Markov Decision Process und Techniken des Bestärkenden Lernens eingegangen. Dies dient der mathematischen Formalisierung von Simulationsumgebung und Lernverfahren.

2.1 Kinematische Fahrzeugmodelle

In diesem Abschnitt werden kinematische Bewegungsmodelle zur Simulation von Fahrzeugen eingeführt. Da sich diese Arbeit nicht auf ein spezifisches Fahrzeug fokussiert, sondern allgemeine Erkenntnisse über die effiziente Erlernbarkeit von Fahrverhaltensweisen mit einer Vielzahl verschiedenster Fahrzeuge liefern soll, werden zwei sehr grundlegende Kinematiken anhand des Fahrradmodells und der Differential Drive Kinematik vorgestellt. Die folgenden Informationen stammen aus dem Buch „Wheeled mobile robotics: from fundamentals towards autonomous systems“ [9] und beziehen sich auf dessen Abschnitte 2.2.1 und 2.2.2.

2.1.1 Kinematisches Fahrradmodell

Das Fahrradmodell beschreibt die Bewegung eines Zweiradfahrzeugs mit Vorder- und Hinterachse, wobei die Lenkung durch eine Drehung des Vorderrads um die z-Achse des Koordinatensystems erreicht wird. Es können auch herkömmliche Automobile mit Vorder- und Hinterachse durch das Fahrradmodell abgebildet werden, wenn vereinfachend davon ausgegangen wird, dass sich auf beiden Achsen jeweils nur ein Rad in der Mitte befindet. Für eine idealisierte Simulation wird außerdem angenommen, dass es keinen Reibungswiderstand gibt und dass sich das Fahrzeug in einer flachen 2D Ebene bewegt.

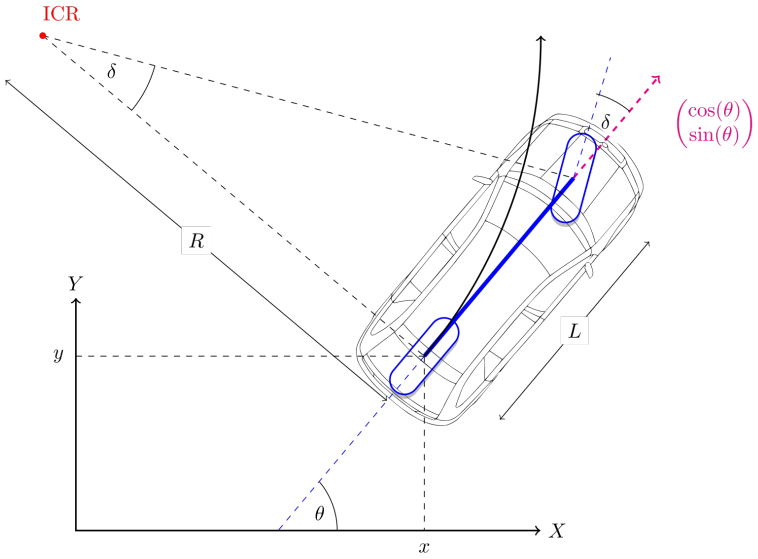

Abbildung 2.1: Skizze des kinematischen Fahrradmodells [10]

Wie in Abbildung 2.1 zu sehen ist, befindet sich das Fahrzeug mit dem Hinterrad an der Position $(x, y)^T$ und hat das Vorderrad um den Lenkwinkel δ eingeschlagen. L beschreibt die Distanz zwischen Vorder- und Hinterachse, θ die Ausrichtung, v die Geschwindigkeit und a die Beschleunigung des Fahrzeugs. $\Delta\theta$ repräsentiert hingegen die Winkelgeschwindigkeit, mit der sich das Fahrzeug auf seiner Kreisbahn um den Momentanpol bewegt. Die Berechnung des neuen Fahrzeugzustands nach Δt kann mit folgenden Gleichungen formalisiert werden:

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ \theta \\ v \end{pmatrix} = \begin{pmatrix} v \cdot cos(\theta) \\ v \cdot sin(\theta) \\ v \cdot \frac{tan(\delta)}{L} \\ a \end{pmatrix} \tag{2.1}$$

Anhand der neuen Geschwindigkeit und Ausrichtung des Fahrzeugs wird dessen gerichtete Bewegungsdynamik bestimmt. Der Fahrer steuert durch die Vorgabe der Beschleunigung a und des Lenkwinkels δ. Als Repräsentation des aktuellen Fahrzeugzustands dienen die Geschwindigkeit v und der Lenkwinkel δ.

2.1.2 Differential Drive

Bei der Differential Drive Kinematik wird von einem Fahrzeug ausgegangen, dessen linke und rechte Räder unabgängig voneinander drehen können. Indem die Räder auf beiden Seiten entgegengesetzt drehen, kann sich das Fahrzeug auf der Stelle drehen.

Eine Lenkung wird erzielt, indem sich die Räder auf der kurvenabgewandten Seite schneller drehen. Auch hier wird idealisierend angenommen, dass sich das Fahrzeug in einer flachen 2D Ebene bewegt und es keinen Reibungswiderstand gibt.

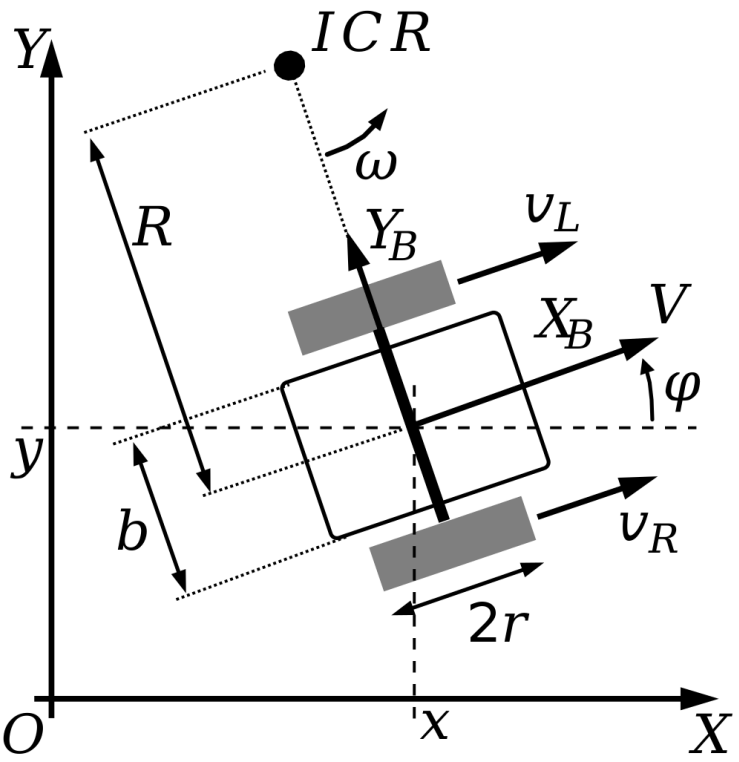

Abbildung 2.2: Skizze der Differential Drive Kinematik, [11]

Wie in Abbildung 2.2 dargestellt, befindet sich die Fahrzeugbasis $(x, y)^T$ in der Mitte der auf einer Achse aufgehängten, unabhängig voneinander drehbaren Räder mit Radius r. φ beschreibt die Ausrichtung des Fahrzeugs im Koordinatensystem und ω repräsentiert die Winkelgeschwindigkeit, mit der sich das Fahrzeug auf seiner Kreisbahn um den Momentanpol mit Geschwindigkeit v dreht. Die Räder rotieren mit den Rotationsgeschwindigkeiten ω_R und ω_L um ihre Achse der Breite b und legen pro Umdrehung eine Strecke in Höhe ihres Umfangs von $2r\pi$ zurück. Mit x_B und y_B wird die zum Fahrzeug relative Bewegung nach vorne bzw. zur Seite beschrieben. Folgende Gleichungen modellieren die Differential Drive Kinematik:

$$\frac{d}{dt}\begin{pmatrix} x_B \\ y_B \\ \varphi \end{pmatrix} = \begin{pmatrix} v \cdot x_B \\ v \cdot y_B \\ \omega \end{pmatrix} = \begin{pmatrix} \frac{r}{2} \cdot (\omega_R + \omega_L) \\ 0 \\ \frac{r}{b} \cdot (\omega_R - \omega_L) \end{pmatrix} \tag{2.2}$$

Für eine erleichterte Steuerung gibt der Fahrer die Geschwindigkeit V und Winkelge-

schwindigkeit ω vor, die anschließend in die passenden Rotationsgeschwindigkeiten ω_R und ω_L der Räder übersetzt werden, um eine entsprechende Bewegung zu erzielen. Als Repräsentation des aktuellen Fahrzeugzustands dienen die zur Fahrzeugausrichtung relative Vorwärtsbewegung x_B und die Winkelgeschwindigkeit ω.

$$\omega_R = \frac{2V + \omega b}{2r}, \omega_L = \frac{2V - \omega b}{2r} \tag{2.3}$$

$$\frac{d}{dt}\begin{pmatrix} x \\ y \\ \varphi \end{pmatrix} = \begin{pmatrix} V \cdot cos(\varphi) \\ V \cdot sin(\varphi) \\ \omega \end{pmatrix} \tag{2.4}$$

Bei der Modellierung der resultierenden Bewegung ergibt sich im Vergleich zum Fahrradmodell eine sehr ähnliche Kinematik, wobei $V = v$ und $\varphi = \theta$ entspricht.

2.2 Simulation von Fußgängern mit Social Force

Da das Fahrzeug auf Gehwegen bzw. in Fußgängerzonen zum Einsatz kommen soll, befasst sich dieser Abschnitt mit der Simulation von Fußgängern, die sich entweder alleine oder in Gruppen bewegen. Mithilfe des *Social Force* Modells [6] können die Bewegungen der einzelnen Fußgänger mit additiven Kräften in Form von 2D Vektoren beschrieben werden. Das ursprüngliche Modell kann durch zusätzliche Kräfte zur Umsetzung von Gruppendynamiken [7] und der Interaktion mit Fahrzeugen [1] erweitert werden.

2.2.1 Grundlagen Social Force

Im Social Force Modell bestimmen Anziehungs- und Abstoßungskräfte die Richtung, in die die einzelnen Fußgänger laufen. Pro Fußgänger wird eine gewichtete Summe aller auf ihn wirkenden Kräfte gebildet, um die angestrebte Laufrichtung zu bestimmen. Anschließend werden die resultierenden Kräfte in träge Bewegungen umgesetzt, die die Fußgänger auf ihre angestrebten Gehgeschwindigkeiten beschleunigen und anschließend auch halten, bis schließlich am Ziel abgebremst wird.

Die *Desired Force* ist eine anziehende Kraft, die durch den Vektor von der Position des Fußgängers zum Ziel beschrieben wird und somit den Fußgänger geradlinig in Richtung des Ziels bewegt. Zur Vermeidung von Kollisionen wird der Fußgänger mittels *Obstacle Force* von Hindernissen abgestoßen, was durch das virtuelle Potentialfeld

$F = -\frac{\partial}{\partial p} dist(o,p)^{-2}$ der reziprogen Distanz zwischen dem Fußgänger p und dem Hindernis o modelliert wird, wie in Abbildung 2.3 zu sehen ist. Des weiteren beschreibt die *Social Force* sowohl die Neugier der Fußgänger, sich von anderen Fußgängern angezogen zu fühlen, als auch eine gewisse Abstoßung gegenüber Fremden zu empfinden, wenn diese zu nahe kommen. Durch die Differenzen der Bewegungsvektoren und Positionen zweier Fußgänger bewegt eine anziehende Kraft die Fußgänger aufeinander zu. Kollisionen werden durch eine abstoßende, orthogonale Kraft vermieden, sobald sich die Fußgänger zu nahe kommen, sodass sich die Fußgänger gegenseitig ausweichen.

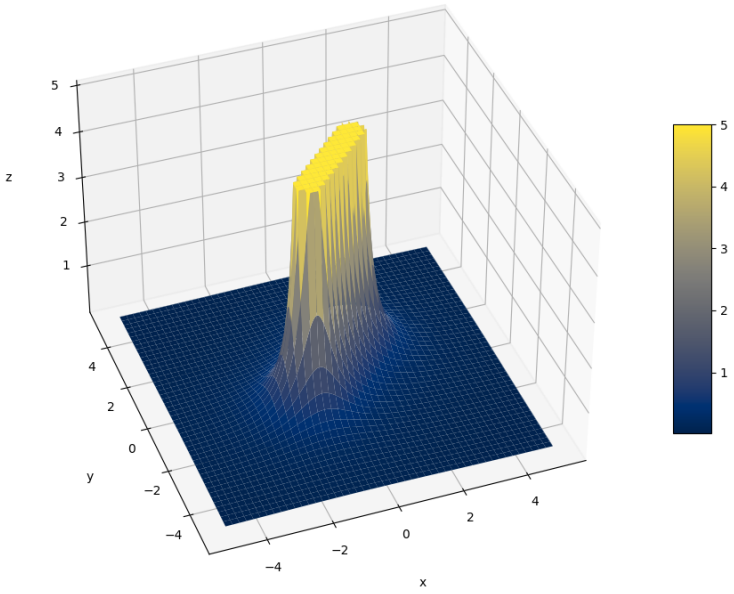

Abbildung 2.3: Potentialfeld der auf die Fußgänger wirkenden Abstoßungskraft von Hindernissen. Die x- und y-Achse entsprechen der modellierten 2D Ebene, die z-Achse repräsentiert die Stärke des Potentialfelds, das um das Hindernis wirkt. Bei der abstoßenden Kraft handelt es sich um die negative Magnitude bezüglich der Fußgängerposition.

2.2.2 Social Force mit Gruppen

Bisher wurde nur die Modellierung einzelner Fußgänger betrachtet, die sich unabhängig voneinander bewegen. Da aber ca. 70% aller Fußgänger in Gruppen von ca. 2-4 Fußgängern unterwegs sind [7], wird das Social Force Modell entsprechend um abstoßende und anziehende Kräfte erweitert, um Gruppendynamiken zu beschreiben.

Anhand einiger Experimente wurde bei Gruppen häufig beobachtet, dass sich Gruppenmitglieder in einer orthogonalen Linie zur Laufrichtung anordnen. Mittels orthogonaler Abstoßung vom und gleichzeitiger Anziehung zum Gruppenzentrum, kann ein entsprechender Effekt erzielt werden, was durch die *Group Repulsive Force* und *Group Cohesive Force* umgesetzt wird. Die *Group Gaze Force* modelliert hingegen das Bedürfnis, sich während der Fortbewegung als Gruppe mit Augenkontakt zu unterhalten. Äußere Gruppenmitglieder laufen daher etwas vorne weg, was zur typischen V-Form führt, falls genügend Platz vorhanden ist.

2.2.3 Simulation des Ausweichverhaltens bezüglich Fahrzeugen

Nun sind die Fußgänger zwar in der Lage, untereinander und mit Hindernissen zu interagieren, aber ignorieren bisher völlig das Fahrzeug. In der Realität würden Fußgänger aber sehr wohl einem herannahenden Fahrzeug ausweichen, sofern sie dieses rechtzeitig wahrnehmen können. Daher wird eine abstoßende Kraft zwischen Fahrzeug und Fußgängern definiert, die das Fahrzeug als gewöhnliches Hindernis betrachtet, dem die Fußgänger ausweichen müssen.

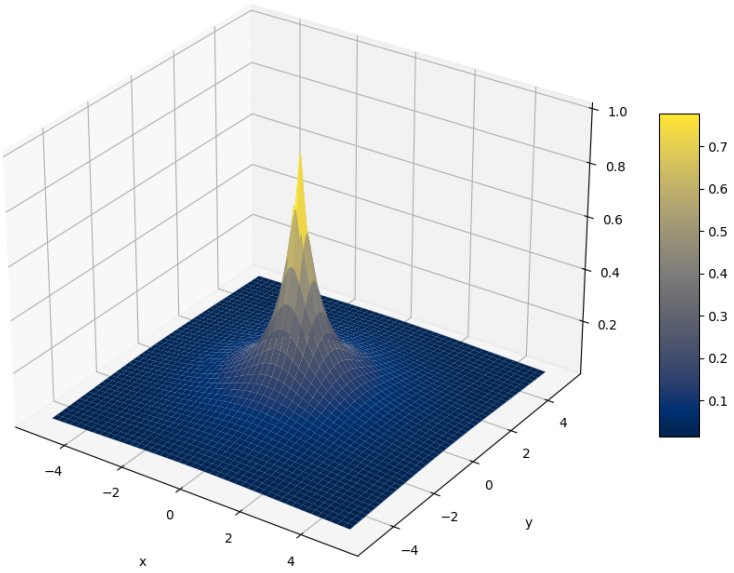

Abbildung 2.4: Potentialfeld der auf die Fußgänger wirkenden Abstoßungskraft von Fahrzeugen. Die x- und y-Achse entsprechen der modellierten 2D Ebene, die z-Achse repräsentiert die Stärke des Potentialfelds, das um die Position des Fahrzeug wirkt. Bei der abstoßenden Kraft handelt es sich um die negative Magnitude bezüglich der Fußgängerposition.

Die *Ped-Robot Force* wird ähnlich zur Abstoßung von Hindernissen als virtuelles Potentialfeld umgesetzt, was durch die partielle Ableitung $F = -\frac{\partial}{\partial p} dist(v,p)^{-2}$ der reziprogen Distanz zwischen Fußgänger p und Fahrzeug v beschrieben wird, wie in Abbildung 2.4 dargestellt. Dies führt dazu, dass Fußgänger einem stehenden Fahrzeug auf einer elliptischen Bahn ausweichen. Ein Beweis der Berechnungsformel liegt im Appendix bei.

2.2.4 Mögliche Erweiterungen

Für die Simulation zusätzlicher Verkehrsteilnehmer gibt es ebenfalls Modelle anhand von Social Force. Dies ermöglicht beispielsweise die Steuerung von E-Scootern [12] oder anderer Fahrzeuge [13] mittels Social Force. Entsprechende Ansätze werden jedoch im Folgenden nicht weiter thematisiert, da nur die Interaktion zwischen einem steuerbaren Fahrzeug und Fußgängern betrachtet werden soll. Es wäre jedoch denkbar, per Social Force gesteuerte Fahrzeuge zu einem späteren Zeitpunkt der Simulation hinzuzufügen, sofern die Modellierung derartiger Interaktionen mit der autonomen Fahrsoftware sinnvoll erscheint.

2.3 Markov Decision Process

Ein Markov Decision Process (MDP) modelliert die Interaktion eines Agenten mit seiner Umgebung [14]. Der Agent kann hierfür zu jedem diskreten Zeitpunkt t eine Aktion a_t anhand des für ihn sichtbaren Systemzustands o_t auswählen und durchführen, wodurch das System vom Zustand s_t in den Zustand s_{t+1} überführt wird. Als Rückmeldung erhält der Agent eine Belohnung r_t und eine aktualisierte Sicht o_{t+1} auf den neuen Systemzustand s_{t+1}.

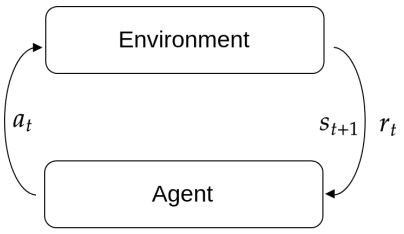

Abbildung 2.5: Veranschaulichung eines Markov Decision Process

Um einen MDP zu modellieren, muss die Umgebung eine Übergangsfunktion $\rho : S \times A \mapsto$

$S \times R$ bereitstellen, welche den Ausgangszustand s_t nach Durchführung der Aktion a_t auf den resultierenden Zustand s_{t+1} und die erhaltene Belohnung r_t abbildet; S, A und R beschreiben jeweils den Zustands-, Aktions- und Belohnungsraum. In den meisten realitätsnahen Anwendungsfällen ist zudem die Wahrnehmung eines Agenten bezüglich seiner Umgebung unvollständig, sodass der Agent nur einen Ausschnitt des Gesamtsystems sieht, beispielsweise durch entsprechende Sensoren. Es wird folglich von einem Partially Observable Markov Decision Process (POMDP) gesprochen, da der Agent nur den für ihn sichtbaren Teil o_t des gesamten Systemzustandes s_t wahrnimmt. Eine zusätzliche Abbildung $\omega : S \mapsto O$ übersetzt den Systemzustand s_t in den für den Agenten sichtbaren Teilzustand o_t des Wahrnehmungsraums O (engl. Observation Space).

Durch die abstrakte Definition der Übergangsfunktion sind vielseitige Umsetzungen einer Umgebung möglich, unter anderem realitätsgetreue Simulationen mit physikalischen Formeln, zufallsbasierte Modelle oder gar trainierte Neuronale Netze. Ein POMDP liegt auch insbesondere beim Autonomen Fahren vor, da der Fahragent mittels Sensorik immer nur einen Ausschnitt der Realität wahrnimmt. Beispielsweise kann ein Objekt durch ein anderes Objekt temporär verdeckt sein, sodass es der Agent nicht sehen kann. Dennoch muss der Agent auch latente Wahrnehmungen in seine Entscheidungen mit einbeziehen.

Die durch die Übergangsfunktion ρ definierten Transitionen zwischen den Zuständen spannen einen Spielegraph auf, bei dem die Zustände den Knoten und die im jeweiligen Zustand wählbaren Aktionen den gerichteten Kanten entsprechen. Durch die Interaktion des Agenten mit seiner Umgebung wird der Graph durchlaufen, was sich in einer Abfolge an Zuständen, Aktionen und Belohnungen in Form einer Trajektorie τ manifestiert. Eine Trajektorie kann als eine Menge an Erfahrungen (s_t, a_t, r_t, s_{t+1}), den sogenannten SARS-Tupeln, für die jeweiligen Zeitschritte $t \in \{t_0 ... t_{term}\}$ verstanden werden. Sobald ein Terminalzustand des Spielegraphs erreicht wird, endet die Trajektorie. Der Zeitabschnitt zwischen dem initialen Zeitpunkt t_0 und dem terminalen Zeitpunkt t_{term} wird im Kontext des Bestärkenden Lernens auch als Episode bezeichnet.

$$\tau = \{(s_t, a_t, r_t, s_{t+1}) | t \in \{t_0 ... t_{term}\} \wedge \rho(s_t, a_t) = (s_{t+1}, r_t)\} \tag{2.5}$$

Zur Lösung eines MDP verfolgt der Agent das Ziel, die über eine Episode gesammelten Belohnungen (engl. Return, Gain) zu maximieren, was mit der Formel $G_t = \sum_{l=0}^{\infty} r_{t+l} \cdot \gamma^l$ ausgedrückt werden kann. $\gamma \in (0, 1)$ stellt einen Diskontierungsfaktor dar, der

Belohnungen in der Gegenwart stärker als Belohnungen in der Zukunft gewichtet, sodass weit in der Zukunft liegende Belohnungen kaum noch zum Return beitragen, da $\lim_{l\to\infty} r_{t+l} \cdot \gamma^l = 0$. Für Spielegraphen ohne Terminalzustände können auch Trajektorien mit einem ausreichend großen Zeithorizont Δt als eine abgeschlossene Episode betrachtet werden. Aus praktischen Gesichtspunkten genügen je nach Problemstellung meist Zeithorizonte zwischen 100 und 1000 Zeitschritten, was durch die Wahl des Hyperparameters γ gesteuert wird. Eine typische Parametrisierung ist $\gamma = 0.99$, was Belohnungen nach 50, 100, 200 und 500 Zeitschritten mit einem Faktor von jeweils 0.61, 0.36, 0.13 und 0.01 zum Return beitragen lässt.

2.4 Lernverfahren nach Sutton und Barto

In den folgenden Abschnitten werden die grundlegenden Lernverfahren des Bestärkenden Lernens eingeführt, wie es im Standardwerk „Reinforcement Learning: An Introduction“ von Sutton und Barto [14] beschrieben wird.

2.4.1 Value Iteration

Bei der Value Iteration schätzt der Agent die echten Zustandsbewertungen V der erwarteten Returns $E[G_t]$ für jeden Zustand s_t mit der Value Function $V\phi$ und deren anpassbaren Parametern ϕ. Es wird typischerweise von einem perfekten, deterministischen Umgebungsmodell ausgegangen, dessen Trajektorien $\tau \in T$ systematisch bezüglich ihres Returns G_τ ausgewertet werden. Da es sich beim Umgebungsmodell um einen Spielegraphen handelt, entspricht das Finden optimaler Aktionen dem Finden kürzester Wege mithilfe des Bellman-Ford Algorithmus, wenn die multiplikativ inversen, diskontierten Belohnungen als Kantengewichte herangezogen werden. Dies steht im direkten Bezug zu Bellmans Optimalitätsgleichung aus der Dynamischen Programmierung, die besagt, dass sich eine optimale Lösung immer aus optimalen Teillösungen zusammensetzt. Demnach ist die Bewertung $V(s_t)$ des Zustands s_t als die Summe aus der unmittelbaren Belohnung r_t und der maximal zu erwartenden, kumulierten Belohnungen $\sum_{l=1}^{\infty} \gamma r_{t+l}$ aller von s_t aus erreichbaren Folgezustände $\{s_{t+1}, s_{t+2}, ...\}$ definiert. Es ist zu bemerken, dass in der Vergangenheit bereits erhaltene Belohnungen keinen Einfluss auf zukünftige Entscheidungen haben, da aus der Sicht des Agenten zum jeweiligen Zeitpunkt t im Zustand s_t die Vergangenheit unveränderbar ist.

$$V(s_t) = r_t + max\{G_\tau | \tau \epsilon T\} \tag{2.6}$$

Das einer Value Iteration zugrundeliegende, deterministische Modell kann auch durch stochastische Zustandsübergänge erweitert werden. Wird nun im Zustand s_t die Aktion a_t gewählt, geht die Umgebung nur zu einer gewissen Wahrscheinlichkeit $P(s_{t+1}|s_t, a_t)$ in den Folgezustand s_{t+1} über, worauf der Agent keinen Einfluss mehr hat. Folglich müssen für eine erwartungsgetreue Schätzung von $E[G_t]$ alle Untertrajektorien möglicher Folgezustände aus dem Trajektorienbaum nach ihren bedingten Eintrittswahrscheinlichkeiten $P(s_{t+1}|s_t, a_t)$ gewichtet werden.

Die Value Iteration ist sehr einfach umzusetzen, allerdings gibt es die große Einschränkung, dass insbesondere von einem perfekten Umgebungsmodell mit endlichen Zustands- und Aktionsräumen augegangen wird. Selbst für perfekt modellierbare Brettspiele wie Schach gibt es mindestens 10^{120} verschiedene Zugfolgen und die Größe des Zustandsraums aller möglichen Spielstellungen wird auf ca. 10^{50} geschätzt [15]. Es ist daher zwingend erforderlich, heuristische Suchalgorithmen einzusetzen, da eine erschöpfende Auswertung des Spielegraphen vollkommen aussichtslos wäre. Auch für den Anwendungsfall des Autonomen Fahrens mit kontinuierlichen Zustands- und Aktionsräumen ist die Value Iteration nicht praktikabel, da zunächst alle Zustände und Aktionen diskretisiert werden müssten, wodurch die Genauigkeit verloren geht. Es besteht zudem das große Problem, dass die Umgebung für die systematische Durchsuchung des Spielegraphen ständig zurückgesetzt werden muss, um alle möglichen Aktionen a_t des Zustands s_t , im stochastischen Fall sogar mehrmals, durchzuprobieren. Daher werden im Folgenden nur Verfahren behandelt, die anhand der kontinuierlichen Interaktion des Agenten mit seiner Umgebung lernen können.

2.4.2 Temporal Difference Learning

Beim Temporal Difference Learning (deutsch TD-Lernen) wird der erwartete Return $E[G_t]$ anhand von Bewertungen $Q(s_t, a_t)$ für das Auswählen der Aktion a_t im Zustand s_t beschrieben. Wie es der Name bereits andeutet, wird hierfür die Differenz zwischen zwei aufeinanderfolgenden Zeitschritten gebildet, um in der Zukunft liegende Belohnungen über mehrere Zeitschritte zu propagieren. Die Belohnung G_t entspricht der Summe aus der im Zeitschritt t erhaltenen Belohnung r_t und dem bestmöglichen, geschätzten Return $V(s_{t+1})$ des nächsten Zeitschritts $t + 1$, falls die optimale Aktion $a_{t+1} = max_a\{Q(s_{t+1}, a)\}$ gewählt wird, wie bei Bellmans Optimalitätsgleichung beschrieben.

$$G_t = r_t + \gamma \cdot max_a\{Q(s_{t+1}, a)\} \tag{2.7}$$

Wenn es sich bei s_t um einen Terminalzustand handelt, besteht ein Sonderfall, da keine weiteren Aktionen mehr durchgeführt werden können, sodass auch keine weiteren Belohnungen zu erwarten sind, d.h. $\gamma \cdot max_a\{Q(s_{t+1}, a)\} = 0$. Somit verbleibt $G_t = r_t$. Die Formel kann durch die Indikatorfunktion $term : S \mapsto \{0, 1\}$ erweitert werden, um die Betrachtung von Terminalzuständen zu modellieren. Für Terminalzustände mit $term(s_t) = 1$ entfällt nun der zweite Term der Summe.

$$G_t = r_t + (1 - term(s_t)) \cdot \gamma \cdot max_a\{Q(s_{t+1}, a)\} \tag{2.8}$$

Das für das TD-Lernen charakteristische Vorgehen, die Belohnungen des nächsten Zeitschritts mit Q zu schätzen, anstatt bis zum Ende der Episode zu warten und den tatsächlich beobachteten Return G_t heranzuziehen, wird als Bootstrapping bezeichnet. Lernmethoden, die stattdessen bis zu einem Terminalzustand warten und dann den tatsächlichen Return verwenden, zählen zur Familie der Monte-Carlo Methoden. Es existieren zudem Mischformen wie TD(λ), bei denen sowohl die tatsächlichen, diskontierten Rewards für einen Zeithorizont $\Delta\lambda$ als auch eine Schätzung mithilfe von Q für alle weiteren Summanden in die Berechnung des Returns eingehen.

Anders als bei der Value Iteration werden beim TD-Lernen nicht alle Aktionen für jeden Zustand systematisch durchprobiert. Stattdessen interagiert der Agent kontinuierlich mit seiner Umgebung, wodurch eine Trajektorie erzeugt wird. Dies entspricht einem zufälligen Ablaufen (engl. Random Walk) des Spielegraphen. Der Zufall beim Auswählen zwischen allen möglichen Aktionen in einem Zustand s_t wird dabei durch die Wahrscheinlichkeitsverteilung $\pi(s_t)$ modelliert, die im Kontext des Bestärkenden Lernens auch als Strategie (engl. Policy) bezeichnet wird. Ein komplett zufälliges Explorieren der Trajektorien, bei dem alle Aktionen gleichberechtigt gewählt werden, wäre allerdings ähnlich ineffizient wie das systematische Durchsuchen des Spielegraphen. Daher soll das bereits gelernte Wissen zur Bewertung aller möglichen Aktionen mithilfe der Q-Funktion in die Strategie π einfließen, sodass besonders vielversprechende Trajektorien deutlich intensiver exploriert werden. Eine gierige Strategie, die immer die optimale Aktion $a_t = argmax_a\{Q(s_t, a)\}$ mit dem höchsten angenommenen Return G_t auswählt, kann jedoch nur sehr schlecht längerfristige Pläne mit einer weit in der Zukunft liegenden Belohnung lernen, da sie kurzfristige Pläne mit niedrigen Belohnungen bevorzugen würde. Daher wird eine sogenannte ϵ-greedy Strategie verwendet, die mit der Wahrscheinlichkeit ϵ eine uniform zufällige Aktion exploriert und mit der Gegenwahrscheinlichkeit $1 - \epsilon$ die beste Aktion gierig ausnutzt. Typischerweise wird ϵ im Laufe des Trainings reduziert, sodass zu Beginn viel und gegen Ende immer weniger

exploriert wird.

$$\pi_\epsilon(s_t) = \begin{cases} \text{argmax}_a\{Q(s_t, a)\}, & Pr(X \geq \epsilon) \\ \text{uniform}_a(s_t), & Pr(X < \epsilon) \end{cases} \tag{2.9}$$

Damit das Modell Q_ϕ mit anpassbaren Parametern ϕ die erwarteten Returns schätzen kann, d.h. $Q_\phi(s_t, a_t) \approx E[G_t]$, wird nun das Gradientenabstiegsverfahren (engl. Gradient Descent) eingeführt. Hierbei wird der Fehlerterm L minimiert, indem entgegen der ansteigenden Magnitude des Fehlers verschoben wird. Die Anpassung der Modellparameter $p\epsilon\phi$ erfolgt entsprechend ihres Beitrags zum Fehlerterm, indem ihre partiellen Ableitungen des Fehlerterms, die sog. Gradienten, ausgewertet werden. Um den Fehlerterm L langsam zu minimieren, werden die Gradienten mit der Lernrate $\alpha\epsilon(0, 1)$ multipliziert.

$$p_{new} = p - \alpha \frac{\partial}{\partial p} L \tag{2.10}$$

Im speziellen Fall geht es um den Schätzfehler L zwischen den geschätzten Belohnungen $\hat{Q}(s_t, a_t)$ und den erwarteten Belohnungen $E[G_t]$. Zur Quantifizierung des Schätzfehlers wird die mittlere, quadratische Abweichung (engl. Mean Squared Error, MSE) herangezogen.

$$MSE(Q_\phi, E[G]) = \frac{1}{N} \sum_{n=1}^{N} \|Q_\phi(s_n, a_n) - E[G_n]\|^2 \tag{2.11}$$

Das Lernen von $Q_\phi \sim E[G]$ stellt somit ein Minimierungsproblem bezüglich des Schätzfehlers $L = \sum_{n=1}^{N} \|Q_\phi(s_n, a_n) - E[G_n]\|^2$ und der anpassbaren Modellparameter ϕ dar.

$$\underset{\phi}{\text{argmin}} \ \frac{1}{N} \sum_{n=1}^{N} \|Q_\phi(s_n, a_n) - E[G_n]\|^2 \tag{2.12}$$

Zur Anpassung der Modellparameter $p\epsilon\phi$ müssen nun die partiellen Ableitungen $\frac{\partial}{\partial p} L$ ausgewertet werden. Zur Vereinfachung des resultierenden Terms wird die Differenzierung des halbierten Fehlers betrachtet. Dies ist äquivalent zu einer Anpassung mit einer doppelt so großen Lernrate α.

$$\begin{aligned}
\frac{\partial}{\partial p}L &= \frac{\partial}{\partial p}\frac{1}{2N}\sum_{n=1}^{N}\|Q_\phi(s_n,a_n) - E[G_n]\|^2 \\
&= \frac{\partial}{\partial p}\frac{1}{2N}\sum_{n=1}^{N}(Q_\phi(s_n,a_n) - E[G_n])^2 \\
&= \frac{1}{2N}\sum_{n=1}^{N}\frac{\partial}{\partial p}(Q_\phi(s_n,a_n) - E[G_n])^2 \\
&= \frac{1}{2N}\sum_{n=1}^{N}2(Q_\phi(s_n,a_n) - E[G_n])\frac{\partial}{\partial p}(Q_\phi(s_n,a_n) - E[G_n]) \\
&= \frac{1}{N}\sum_{n=1}^{N}(Q_\phi(s_n,a_n) - E[G_n])\frac{\partial}{\partial p}Q_\phi(s_n,a_n)
\end{aligned} \tag{2.13}$$

Da der erwartete Return $E[G_t]$ nicht bekannt ist, muss dieser geschätzt werden. Hierfür können wie bereits erwähnt vielfältige Methoden von reinen Monte-Carlo Verfahren über Verfahren mit Bootstrapping bis hin zu Mischformen wie TD-λ verwendet werden. Typisch ist eine Umsetzung wie in Gleichung 2.14 mittels Bootstrapping des nächsten Zeitschritts.

$$\begin{aligned}
p_{new} &= p - \alpha \cdot \frac{1}{m}\sum_{i=1}^{m}(Q_\phi(s_t,a_t) - E[G_t])\frac{\partial}{\partial p}Q_\phi(s_t,a_t) \\
E[G_t] &= r_t + (1 - term(s_t))\gamma max_a\{Q_\phi(s_{t+1},a)\}
\end{aligned} \tag{2.14}$$

Das Modell des Agenten entspricht in der Regel einem Tiefen Neuronalen Netz Q_ϕ mit anpassbaren Parametern ϕ, was als Deep Q-Network (DQN) bezeichnet wird [16]. Für sehr kleine Probleme kommen auch tabellarische Ansätze mit expliziten Werten pro Zustands-Aktions-Paar infrage. Aufgrund der großen Zustands- und Aktionsräume im Anwendungsfall des Autonomen Fahrens sind diese jedoch uninteressant und werden im Folgenden nicht weiter behandelt. Neben Genauigkeitsproblemen im Umgang mit nachträglich diskretisierten, vormals kontinuierlichen Aktionsräumen ergeben sich beim TD-Lernen in der Praxis zudem Probleme, da die Strategie π durch ϵ beeinflusst ist. Wird ϵ falsch gewählt, liefert das Training oftmals unbrauchbare Ergebnisse. Daher werden im Folgenden Verfahren betrachtet, die die während des Trainings verwendete Strategie zur Exploration der Umgebung selbst regulieren können.

2.4.3 Policy Gradient Methoden

Die bisher besprochenen Lernmethoden schätzen jeweils die erwartete Belohnung $E[G_t]$ eines Zustands s_t oder eines Zustands-Aktions-Paars (s_t, a_t). Hingegen wird bei Policy Gradient Methoden eine Approximation $\pi_\theta(s_t)$ der optimalen Strategie $\pi(s_t)$ mit trainierbaren Parametern θ erlernt. π_θ stellt hierbei die Wahrscheinlichkeitsverteilung zur Auswahl der Aktionen beim Durchlaufen des Spielegraphs dar. Das Training erhöht Wahrscheinlichkeiten für gute Aktionen und senkt Wahrscheinlichkeiten für schlechte Aktionen, sodass sich die Strategie π_θ im Gegensatz zur ϵ-greedy Strategie beim TD-Lernen von selbst reguliert.

Um die Eigenschaften einer optimalen Strategie zu verstehen, wird die erwartete Belohnung $E[G_t]$ im Zustand s_t für Trajektorien $\tau \epsilon T$ aus der Menge aller möglichen Trajektorien T betrachtet. Per Definition des Erwartungswerts entspricht der erwartete Return $E[G_t]$ der Summe der beobachteten Returns $R(\tau)$ der jeweiligen Trajektorien $\tau \epsilon T$, gewichtet nach Eintrittswahrscheinlichkeit $P(\tau, \theta)$ der Trajektorie unter den aktuellen Modellparametern θ.

$$E[G_t] = \sum_{\tau \epsilon T} R(\tau) P(\tau, \theta) \tag{2.15}$$

Um den erwarteten Return zu maximieren, müssen die Parameter θ entsprechend angepasst werden, um ertragreiche Trajektorien bevorzugt auszuwählen. Es handelt sich folglich um ein Maximierungsproblem des erwarteten Returns $E[G_t]$ mit anpassbaren Parametern θ.

$$\overset{\text{argmax}}{\theta} \sum_{\tau \epsilon T} R(\tau) P(\tau, \theta) \tag{2.16}$$

Maximierungsprobleme sind ebenfalls durch das Gradientenabstiegsverfahren lösbar, indem die Maximierung zu einer Minimierung des multiplikativ Inversen der Zielfunktion U umgewandelt wird, d.h. $\overset{\text{argmax}}{\theta}\ U = \overset{\text{argmin}}{\theta}\ L$ mit $U = -L$. Da dies trivial möglich ist, soll im Folgenden nur das Maximierungsproblem betrachtet werden. Nun folgt die Auswertung der Gradienten $\nabla_\theta E[G_t]$ zur Bestimmung der partiellen Ableitungen für die anschließende Modellaktualisierung.

$$\begin{aligned}\nabla_\theta E[G_t] &= \nabla_\theta \sum_{\tau \epsilon T} R(\tau) P(\tau, \theta) \\ &= \sum_{\tau \epsilon T} R(\tau) \nabla_\theta P(\tau, \theta) \\ &= \sum_{\tau \epsilon T} R(\tau) \frac{P(\tau, \theta)}{P(\tau, \theta)} \nabla_\theta P(\tau, \theta) \\ &= \sum_{\tau \epsilon T} P(\tau, \theta) R(\tau) \frac{\nabla_\theta P(\tau, \theta)}{P(\tau, \theta)}\end{aligned} \tag{2.17}$$

Mithilfe des Logarithmierungstricks $\frac{\frac{\partial}{\partial x} f(x)}{f(x)} = \frac{\partial}{\partial x} log(f(x))$ wird die Formel für die Policy Gradients in ihre typische Form gebracht.

$$\sum_{\tau \epsilon T} P(\tau, \theta) R(\tau) \frac{\nabla_\theta P(\tau, \theta)}{P(\tau, \theta)} = \sum_{\tau \epsilon T} P(\tau, \theta) R(\tau) \nabla_\theta log(P(\tau, \theta)) \tag{2.18}$$

Durch die Gewichtung der einzelnen Summanden mit ihren relativen Häufigkeiten $P(\tau, \theta)$ ist die Summe als eine Approximation über eine repräsentative Menge an Trajektorien zu verstehen, die sich während der Interaktion des Agenten mit seiner Umgebung ergeben. Es ist also keineswegs erforderlich, alle möglichen Trajektorien erschöpfend auszuwerten. Eine Stichprobe $\{\tau_1 ... \tau_m\}$ aus m Trajektorien stellt eine empirische Schätzung der Policy Gradients dar, bei der die Trajektorien entsprechend ihrer relativen Häufigkeiten $P(\tau, \theta)$ in der Summe bereits enthalten sind, sodass eine Gewichtung mit $P(\tau, \theta)$ entfällt.

$$\nabla_\theta E[G_t] \approx \frac{1}{m} \sum_{\tau \epsilon \{\tau_1 ... \tau_m\}} R(\tau) \nabla_\theta log(P(\tau, \theta)) \tag{2.19}$$

Um $P(\tau, \theta)$ schrittweise aus der Abfolge der Zustände und Aktionen zusammenzusetzen, werden nun die einzelnen Zustandsübergänge einer Trajektorie τ betrachtet. Die bedingte Wahrscheinlichkeit für einen durch das SARS-Tupel (s_t, a_t, r_t, s_{t+1}) definierten Zustandsübergang ergibt sich aus dem Produkt von $\pi_\theta(a_t|s_t)$ und $P(s_{t+1}|s_t, a_t)$. Dabei steht $\pi_\theta(a_t|s_t)$ für die Wahrscheinlichkeit, dass vom Modell π_θ im Zustand s_t die Aktion a_t gewählt wird. Hingegen entspricht $P(s_{t+1}|s_t, a_t)$ der Wahrscheinlichkeit, mit der nach dem Auswählen der Aktion a_t im Zustand s_t der Folgezustand s_{t+1} über die stochastische Übergangsfunktion ρ des Umgebungsmodells eintritt. Werden die einzelnen Zustandsübergänge zu einer Trajektorie τ verkettet, ergibt sich die Eintrittswahrscheinlichkeit $P(\tau, \theta)$ als Produkt der bedingten Wahrscheinlichkeiten, dass die jeweiligen Zustandsübergänge eintreten, nachdem die vorherigen Zustandsübergänge

bereits eingetreten sind.

$$P(\tau, \theta) = \prod_{t=t_0}^{t_{term}} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t) \tag{2.20}$$

Nun kann der noch ausstehende Term $\nabla_\theta log(P(\tau, \theta))$ aus der Formel der Policy Gradients weiter ausgewertet werden. Zunächst wird der Term in eine Summe der Logarithmen umgewandelt. Anschließend entfallen die Gradienten des Umgebungsmodells per Konstantenregel, da diese nicht von den Modellparametern θ abhängen.

$$\begin{aligned}
\nabla_\theta log\left(P(\tau, \theta)\right) &= \nabla_\theta log(\prod_{t=t_0}^{t_{term}} P(s_{t+1}|s_t, a_t)\pi_\theta(a_t|s_t)) \\
&= \nabla_\theta \sum_{t=t_0}^{t_{term}} (log(P(s_{t+1}|s_t, a_t)) + log(\pi_\theta(a_t|s_t))) \\
&= \sum_{t=t_0}^{t_{term}} \nabla_\theta log(P(s_{t+1}|s_t, a_t)) + \sum_{t=t_0}^{t_{term}} \nabla_\theta log(\pi_\theta(a_t|s_t)) \\
&= \sum_{t=t_0}^{t_{term}} \nabla_\theta log(\pi_\theta(a_t|s_t))
\end{aligned} \tag{2.21}$$

Nun wird abschließend in die zuvor erhaltene Formel zur Approximation der Policy Gradients eingesetzt. $R(\tau)$ kann durch die beobachteten Belohnungen $G_t = \sum_{l=0}^{\infty} r_{t+l} \cdot \gamma^l$ durch eine Monte-Carlo Schätzung approximiert werden, wenn τ einer abgeschlossenen Episode entspricht. Bessere Schätzungen für $R(\tau)$ mittels TD-Lernen werden im folgenden Abschnitt zu Actor-Critic Methoden 2.5.1 besprochen.

$$\begin{aligned}
\nabla_\theta E[G_t] &\approx \frac{1}{m} \sum_{\tau \epsilon \{\tau_1 \ldots \tau_m\}} R(\tau) \nabla_\theta log(P(\tau, \theta)) \\
&= \frac{1}{m} \sum_{\tau \epsilon \{\tau_1 \ldots \tau_m\}} R(\tau) \sum_{t=t_0}^{t_{term}} \nabla_\theta log(\pi_\theta({a_t}^{(\tau)}|{s_t}^{(\tau)}))
\end{aligned} \tag{2.22}$$

Interessanterweise spielen die Eintrittswahrscheinlichkeiten des stochastischen Umgebungsmodells ρ für die Berechnung der Policy Gradients keine Rolle. Dies macht entsprechende Lernverfahren auf jede Problemstellung anwendbar, bei der der Agent Trajektorien durch Interaktion mit seiner Umgebung erzeugt, was insbesondere auch auf den Anwendungsfall des Autonomen Fahrens zutrifft. Wie bereits in einem vorherigen Abschnitt erwähnt, treten die Trajektorien in einer repräsentativen Stichprobe gemäß ihrer relativen Häufigkeiten auf und sind statistisch voneinander unabhängig. Im Fall von simulierten Umgebungen, die oft trivial replizierbar sind, kann der Agent auch in

mehreren Umgebungen gleichzeitig Aktionen auswählen und somit viele Trajektorien auf einmal erzeugen. Dies stabilisiert das Training und macht Policy Gradient Methoden besonders gut mit Rechenressourcen skalierbar.

Nun besteht allerdings aufgrund des monoton steigenden Verhaltens der Logarithmusfunktion, d.h. $\frac{\partial}{\partial \pi_\theta(a_t|s_t)} log(\pi_\theta(a_t|s_t)) > 0$, das Problem, dass die Auswahlwahrscheinlichkeit $\pi_\theta(a_t|s_t)$ für positive Returns immer erhöht und für negative Returns immer gesenkt wird. Ist $R(\tau)$ immer positiv, werden die Wahrscheinlichkeiten bei jedem Update erhöht, aber nie gesenkt. Vor allem bei selten gewählten Aktionen mit kleinem $\pi_\theta(a_t|s_t)$ ist die Magnitude $\nabla_\theta log(\pi_\theta(a_t|s_t))$ besonders groß, sodass einzelne, schlechte Modell-Updates die komplette Policy umfassend verändern. Dies ist vor allem problematisch, da bei den Policy Gradient Methoden die nächsten Trajektorien anhand der aktuellen Strategie erzeugt werden, wodurch teilweise ganze Trainingsläufe unbrauchbare Ergebnisse liefern. Es wäre daher sinnvoll, $R(\tau)$ mit einer geeigneten Grundlinie (engl. Baseline) zu korrigieren und die Auswirkungen schlechter Modellaktualisierungen abzumildern, was unter anderem in den folgenden Abschnitten behandelt wird.

2.5 Erweiterungen für Policy Gradient Methoden

In den folgenden Abschnitten werden Erweiterungen für die Policy Gradient Methoden anhand der Actor-Critic Methoden, des Generalized Advantage Estimator [17], der Trust Regions [18] und des Proximal Policy Optimization Lernverfahrens [8] besprochen.

2.5.1 Actor-Critic Methoden

Bei den Actor-Critic Methoden wird die Strategie π_θ aus den Policy Gradient Methoden durch ein Modell für die Baseline V_ϕ erweitert, um die Returns beim Policy Update zu korrigieren, sodass die neue Strategie gute Aktionen öfter und schlechte Aktionen seltener auswählt. Die agierende Komponente (Strategie) wird als Actor und die korrigierende Komponente (Baseline) als Critic bezeichnet, was namensgebend für die Familie der Actor-Critic Methoden ist. Um die Korrektur des Returns durch eine Baseline zu verstehen, wird der Begriff des Vorteils (engl. Advantage) eingeführt. Es handelt sich beim Advantage $A^\pi(s_t, a_t)$ der Aktion a_t um die Differenz zwischen der durch die Auswahl der Aktion erhaltenen Belohnung $Q^\pi(s_t, a_t)$ und der Grundbewertung $V^\pi(s_t)$ des Zustands s_t [17]. Mit A^π wird angedeutet, dass die Strategie π die Aktionen wählt.

$$A^{\pi}(s_t, a_t) = Q^{\pi}(s_t, a_t) - V^{\pi}(s_t) \tag{2.23}$$

Aktionen, die in besser bewertete Folgezustände führen, haben einen positiven Advantage und umgekehrt. Werden die Policy Gradients mit den Advantages statt den Returns multipliziert, passt sich die Strategie wie gewünscht an, sodass gute Aktionen öfter und schlechte Aktionen seltener gewählt werden. Es wird bei dieser Variante auch von Advantage Actor-Critic gesprochen, kurz A2C. Wenn zusätzlich die zum Policy Update verwendeten Trajektorien aus mehreren Simulationen stammen, handelt es sich um Asynchronous Advantage Actor-Critic (A3C). Demnach wird $E[A_t]$ statt $E[G_t]$ maximiert.

Eine Korrektur um den echten Advantage $A^{\pi}(s_t, a_t)$ könnte umgesetzt werden, indem $V^{\pi}(s_t) = \frac{1}{n} \sum_a Q^{\pi}(s_t, a)$ als der Durchschnitt über die erwarteten Belohnungen aller möglichen Aktionen gebildet wird. Dies ist jedoch nicht praktikabel, da Q in der Regel unbekannt ist und somit eine Approximation durch TD-Lernen erfordert. Wenn aber bereits eine gute Schätzung für Q existiert, ist es nicht mehr sinnvoll, eine zusätzliche Strategie π zu lernen. Anstatt V mit Q zu schätzen, lässt sich Q über dessen Definition $Q(s_t, a_t) = r + \gamma max_a\{Q(s_{t+1}, a)\}$ als diskontierte Bewertung des bestmöglichen Folgezustands ausdrücken. Da V seinerseits als die Bewertung des bestmöglichen Folgezustands definiert ist, kann $max_a\{Q(s_{t+1}, a)\}$ auch durch V repräsentiert werden. Wird in die Formel des Advantage eingesetzt, ergibt sich die temporale Differenz δ_t^V der Zustandsbewertungen $V(s_t)$, $V(s_{t+1})$ zweier aufeinanderfolgender Zustände s_t und s_{t+1} nur durch die beobachtete Belohnung r_t und V.

$$\delta_t^V = Q(s_t, a_t) - V(s_t) = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{2.24}$$

Wie im Abschnitt über Temporal Difference Learning bereits erwähnt, können neben dem geschätzten Return auch die beobachteten Belohnungen der Trajektorie über einen Zeithorizont $\Delta\lambda$ mit in die TD-Schätzung einfließen, was in etwa TD(λ) aus [14] entspricht. Eine Schätzung des Advantage $\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V$ für $\lambda = k$ Schritte zeigt interessante Eigenschaften aufgrund der teleskopierenden Summe.

$$\hat{A}_t^{(1)} = \delta_t^V = -V(s_t) + r_t + \gamma V(s_{t+1}) \tag{2.25}$$

$$\begin{aligned}\hat{A}_t^{(2)} &= \delta_t^V + \gamma\delta_{t+1}^V \\ &= (-V(s_t) + r_t + \gamma V(s_{t+1})) + \gamma(-V(s_{t+1}) + r_{t+1} + \gamma V(s_{t+2})) \\ &= -V(s_t) + r_t + \gamma r_{t+1} + \gamma^2 V(s_{t+2})\end{aligned} \tag{2.26}$$

$$\hat{A}_t^{(k)} = \sum_{l=0}^{k-1} \gamma^l \delta_{t+l}^V = -V(s_t) + r_t + \gamma^1 r_{t+1} + ... + \gamma^{k-1} r_{t+k-1} + \gamma^k V(s_{t+k}) \tag{2.27}$$

Für sehr große Zeithorizonte $k \to \infty$ wird das Bootstrapping durch $\gamma^k V(s_{t+k})$ immer stärker diskontiert und geht gegen 0, sodass für die Schätzung des Vorteils nur die Bewertung des aktuellen Zustands $V(s_t)$ und der Return $G_t = \sum_{l=0}^{\infty} \gamma^l r_{t+l}$ verbleiben.

$$\hat{A}_t^{(\infty)} = \sum_{l=0}^{\infty} \gamma^l \delta_{t+l}^V = -V(s_t) + \sum_{l=0}^{\infty} \gamma^l r_{t+l} = G_t - V(s_t) \tag{2.28}$$

Nun folgt die Definition des Generalized Advantage Estimator $GAE(\gamma, \lambda)$ [17] als exponentiell gleitender Durchschnitt der k-Schritte Schätzer:

$$\hat{A}_t^{GAE(\gamma,\lambda)} := \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l}^V = (1-\lambda)(\lambda^0 \hat{A}_t^{(1)} + \lambda^1 \hat{A}_t^{(2)} + \lambda^2 \hat{A}_t^{(3)} + ...) \tag{2.29}$$

Der GAE weist ähnlich wie TD(λ) interessante Extremfälle für $\lambda = 0$ und $\lambda = 1$ auf. $GAE(\gamma, 0)$ ist eine Schätzung basierend auf der temporalen Differenz bezüglich des nächsten Zeitschritts mittels Bootstrapping. Hingegen verwendet $GAE(\gamma, 1)$ nur die diskontierten Belohnungen ohne Bootstrapping wie bei Monte-Carlo Methoden. So gesehen ermöglicht GAE eine fließende Abwägung zwischen TD-Schätzungen und beobachteter Returns. Eine gute Parametrisierung ist $\lambda = 0.95$.

Da nun mit dem GAE ein geeigneter Schätzer für die Advantages vorliegt, muss noch ein Modell für V gelernt werden. Hierfür genügt es, das Gradientenabstiegsverfahren auf ein Neuronales Netz V_ϕ mit trainierbaren Parametern ϕ anzuwenden, indem die quadratische Abweichung zwischen der Schätzung des Modells $V_\phi(s_n)$ und den empirischen Returns $\hat{V}_n = \sum_{l=0}^{\infty} \gamma^l r_{t+l}$ minimiert wird, was einem TD-Lernen von V statt Q entspricht.

$$\underset{\phi}{\text{argmin}} \ \frac{1}{N} \sum_{n=1}^{N} \| V_\phi(s_n) - \hat{V}_n \|^2 \tag{2.30}$$

2.5.2 Trust Regions und Proximal Policy Optimization

Mithilfe der Trust Regions [18] und weiteren Verbesserungen werden die Actor-Critic Methoden nochmals um stabilisierende und effizienzsteigernde Komponenten ergänzt, woraus das Lernverfahren der Proximal Policy Optimization (PPO) [8] resultiert. Hierfür wird das bisherige Optimierungsproblem der Actor-Critic Methoden mit GAE als Schätzer für den Advantage $\hat{A}_t$ erneut betrachtet.

$$\underset{\theta}{\text{argmax}} \; E_t[\hat{A}_t] \tag{2.31}$$

Da die Erzeugung der Trajektorien während des Trainings teilweise sehr rechenintensiv sein kann, wäre es sinnvoll, die Trainingsdaten für mehrere aufeinanderfolgende Policy Updates zu verwenden, um eine höhere Trainingsdateneffizienz (engl. Sample Efficiency) zu erreichen. Hierzu führt die Trust Region Policy Optimization (TRPO) [18] eine neue Zielfunktion ein, die dieselben Trainingsdaten solange wiederverwendet, bis sich die aktualisierte Strategie π_θ zu stark von der ursprünglichen Strategie $\pi_{\theta_{old}}$ unterscheidet. Die Kullback-Leibler Divergenz $KL(\pi_\theta||\pi_{\theta_{old}})$ quantifiziert den Unterschied δ zwischen den Strategien bezüglich der in den Trainingsdaten enthaltenen Trajektorien $\tau \epsilon T$. Räumlich betrachtet spannt die Distanz δ eine sichere Zone (sog. Trust Region) um die ursprüngliche Strategie auf, womit die Schrittweite der kumulierten Policy Updates beschränkt wird, was namensgebend für das Verfahren ist.

$$\begin{gathered}\underset{\theta}{\text{argmax}} \; E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t\right] \\ \text{subject to}\, E_t[KL(\pi_{\theta_{old}}(\cdot, s_t)||\pi_\theta(\cdot, s_t))] \leq \delta\end{gathered} \tag{2.32}$$

Wie zu sehen ist, fügt TRPO den Policy Gradients aus den Actor-Critic Methoden eine sog. Policy Ratio $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}$ zwischen den alten und neuen Auswahlwahrscheinlichkeiten bezüglich der Aktionen aus den Trainingsdaten hinzu. Die entsprechende Zielfunktion stammt aus der Conservative Policy Iteration und wird daher L^{CPI} genannt. Aufgrund der zusätzlichen Bedingung durch die Kullback-Leibler Divergenz ist das resultierende Trainingsverfahren inkompatibel zu gängigen Umsetzungen des Gradientenabstiegsverfahrens. Es erscheint deshalb sinnvoll, die Divergenz als Regularisierungsterm in die Formel des Optimierungsproblems zu integrieren, um einen leichter umsetzbaren Algorithmus zu erhalten.

$$\underset{\theta}{\text{argmax}} \; E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\hat{A}_t - \beta \cdot KL(\pi_{\theta_{old}}(\cdot, s_t)||\pi_\theta(\cdot, s_t))\right] \tag{2.33}$$

Aufgrund der Problematik, einen passenden Wert für β zu finden und die Gradienten der

Kullback-Leibler Divergenz zu berechnen, ist dieses Verfahren jedoch nicht praktikabel. Daher greift die Proximal Policy Optimization (PPO) die Zielfunktion L^{CPI} auf und löst dessen Probleme durch die Beschränkung der Schrittweite mittels Gradient Clipping. Die resultierende Zielfunktion $L^{CLIP}(\theta)$, der sog. Clipped Surrogate Loss und das zugehörige Optimierungsproblem setzen sich folgendermaßen zusammen.

$$\overset{\text{argmax}}{\theta} \; {L_t}^{CLIP}(\theta) \tag{2.34}$$

$$\begin{aligned} L^{CLIP}(\theta) &= E_t[min({r_t}(\theta)\hat{A}_t, {r_t}^{CLIP}(\theta)\hat{A}_t)] \\ {r_t}^{CLIP}(\theta) &= clip({r_t}(\theta), 1-\epsilon, 1+\epsilon) \end{aligned} \tag{2.35}$$

Es fällt auf, dass das erste Policy Update mit $\theta = \theta_{old}$ dem ursprünglichen Actor-Critic Verfahren entspricht, da $r_t(\theta_{old}) = 1$. Zudem erhöhen bzw. senken die Advantages $\hat{A}_t$ die Auswahlwahrscheinlichkeiten $\pi_\theta(a_t|s_t)$ wie gehabt je nach Vorzeichen. Dies führt bei mehrmaligem Verwenden der Trainingsdaten jedoch zu Problemen, da sich Policy Updates aufgrund von vorherigen Policy Updates selbst verstärken können, wodurch die Schrittweite zu groß wird. Um zu verstehen, wie $L^{CLIP}(\theta)$ die Gradienten effektiv beschränken kann, werden die Gradienten näher betrachtet.

$$\begin{aligned} \nabla_\theta {L_t}^{CPI} &= \nabla_\theta E_t\left[{r_t}(\theta)\hat{A}_t\right] \\ &= E_t\left[\hat{A}_t \nabla_\theta {r_t}(\theta)\right] \\ &= E_t\left[\hat{A}_t \nabla_\theta \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\right] \\ &= E_t\left[\hat{A}_t \nabla_\theta exp\left(log\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)}\right)\right)\right] \\ &= E_t\left[\hat{A}_t \nabla_\theta exp(log(\pi_\theta(a_t|s_t)) - log(\pi_{\theta_{old}}(a_t|s_t)))\right] \\ &= E_t\left[\hat{A}_t \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \nabla_\theta(log(\pi_\theta(a_t|s_t)) - log(\pi_{\theta_{old}}(a_t|s_t)))\right] \\ &= E_t\left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{old}}(a_t|s_t)} \hat{A}_t(\nabla_\theta log(\pi_\theta(a_t|s_t)) - \nabla_\theta log(\pi_{\theta_{old}}(a_t|s_t)))\right] \\ &= E_t\left[{r_t}(\theta)\hat{A}_t \nabla_\theta log(\pi_\theta(a_t|s_t))\right] \end{aligned} \tag{2.36}$$

Wie erwartet ergeben sich die Policy Gradients für die Maximierung der erwarteten Advantages $E_t[\hat{A}_t \nabla_\theta log(\pi_\theta(a_t|s_t))]$, jedoch zusätzlich mit der Policy Ratio $r_t(\theta)$ gewichtet. Um die Auswirkungen des Clipping zu verstehen, werden als nächstes die Gradienten der durch ϵ beschränkten Zielfunktion $L^{CLIP}(\theta)$ für den Fall ausgewertet, dass das Minimum den beschränkten Term ${r_t}^{CLIP}(\theta)\hat{A}_t$ wählt.

$$\begin{aligned}\nabla_\theta {L_t}^{CLIP} &= \nabla_\theta E_t[\hat{A}_t {r_t}^{CLIP}(\theta)] \\ &= E_t[\hat{A}_t \nabla_\theta {r_t}^{CLIP}(\theta)] \\ &= E_t[\hat{A}_t {r_t}^{CLIP}(\theta) \nabla_\theta log({r_t}^{CLIP}(\theta))]\end{aligned} \tag{2.37}$$

Bei der Auswertung von $\nabla_\theta log({r_t}^{CLIP}(\theta))$ ist zu beachten, dass das Clipping nur zum Tragen kommt, wenn $r_t(\theta) \notin [1-\epsilon, 1+\epsilon]$. Andernfalls ergeben sich dieselben Gradienten wie bei der Auswertung von L^{CPI}. Ist die Policy Ratio jedoch vom Clipping betroffen, wird ${r_t}^{CLIP}(\theta)$ jeweils durch eine der Konstanten $1-\epsilon$ bzw. $1+\epsilon$ ersetzt. Folglich tragen entsprechende Trainingsbeispiele mit beschränkten Policy Ratios nicht zum Policy Update bei, da deren Gradienten durch die Konstantenregel entfallen. Nun muss noch die zusammengesetzte Zielfunktion L^{CLIP} als Ganzes untersucht werden.

$$L^{CLIP}(\theta) = E_t[min(r_t(\theta)\hat{A}_t, {r_t}^{CLIP}(\theta)\hat{A}_t)] \tag{2.38}$$

Da die Terme $r_t(\theta)\hat{A}_t$ und ${r_t}^{CLIP}(\theta)\hat{A}_t)$ für Policy Ratios $r_t(\theta) \in [1-\epsilon, 1+\epsilon]$ innerhalb der Trust Region dasselbe Ergebnis liefern, werden nun die Fälle betrachtet, bei denen die Ratio außerhalb liegt. Handelt es sich um eine Aktion mit negativem Advantage, wird $\pi_\theta(a_t|s_t)$ bei folgenden Policy Updates immer kleiner, wodurch die Policy Ratio gegen 0 geht. Für $r_t(\theta) < (1-\epsilon)$ wählt das Minimum den beschränkten Term, da aufgrund des negativen Advantage $(1-\epsilon)\hat{A}_t < r_t(\theta)\hat{A}_t$. Liegt hingegen eine Aktion mit positivem Advantage vor, erhöht sich $\pi_\theta(a_t|s_t)$ bei jedem weiteren Policy Update, sodass dessen Policy Ratio stark wächst. Für $r_t(\theta) > (1+\epsilon)$ wählt das Minimum ebenfalls den beschränkten Term, da aufgrund des positiven Advantage $(1+\epsilon)\hat{A}_t < r_t(\theta)\hat{A}_t$. Somit erzwingt der Clipped Surrogate Loss effektiv, dass nur Trainingsbeispiele mit einer Policy Ratio $r_t(\theta) \in [1-\epsilon, 1+\epsilon]$ innerhalb der Trust Region zum Policy Update beitragen. Da die Policy Gradients von L^{CLIP} direkt mit der Policy Ratio multipliziert werden, führt PPO keine Policy Updates durch, bei denen sich die alte und neue Strategie um mehr als einen Faktor von ϵ unterscheidet. Dadurch können die Trainingsdaten gefahrlos wiederverwendet werden, um die Trainingsdateneffizienz gegenüber anderen Policy Gradient Verfahren deutlich zu steigern.

Das Lernverfahren der Proximal Policy Optimization kann noch weiter verfeinert werden. Zur leichteren Differenzierbarkeit der Policy Ratio wird diese wie in Gleichung 2.36 exponenziert und logarithmiert, um die Quotientenregel zu vermeiden. Damit $\nabla_\theta log(\pi_\theta(a_t|s_t))$ für besonders kleine Warhscheinlichkeiten nicht gegen ∞ geht, wird ein Term $\psi = 10^{-8}$ zum Argument des Logarithmus addiert, um den Term nach oben

zu beschränken. Dies verhindert ebenfalls besonders große Policy Updates und macht den Algorithmus stabiler. Da bei den Advantages hauptsächlich das Vorzeichen wichtig ist, um die Richtung zu ermitteln, in die die Wahrscheinlichkeiten angepasst werden, bietet es sich an, die Advantages bezüglich ihrer Standardnormalverteilung zu normalisieren. Zusätzlich ist es auch möglich, nicht nur bei den die Policy Ratios, sondern auch bei den normalisierten Advantages ein Clipping vorzunehmen. Um sicherzustellen, dass ausreichend exploriert wird, kann zusätzlich ein Entropieterm hinzugefügt werden, der die Wahrscheinlichkeiten der Strategie hin zu einer Gleichverteilung bewegt.

Für den Anwendungsfall des Autonomen Fahrens bietet PPO einige interessante Erweiterungen, die stabile und effiziente Trainingsläufe ermöglichen. Vor allem die Garantie, dass keine Überanpassungen durch zu optimistische Schätzungen vorgenommen werden, führt dazu, dass fast alle Trainingsläufe erfolgreich sind. Die erhöhte Sample Efficiency ist zudem interessant, um PPO auf rechenintensivere Trainingsumgebungen wie den CALRA Simulator anzuwenden.

2.6 Lernen mit kontinuierlichen Aktionsräumen

Bisher wird immer von diskreten Zuständen und Aktionen als Knoten bzw. Kanten des Spielegraphs ausgegangen. Bei vielen Kontrollproblemen liegen jedoch kontinuierliche Zustands- und Aktionsräume vor, was eine Anpassung der Lernalgorithmen und Modelle erfordert. Wird der Agent durch ein Neuronales Netz repräsentiert, ist die Verarbeitung kontinuierlicher Zustandsräume trivial möglich. Bei der Modellierung kontinuierlicher Aktionsräume gibt es hingegen Probleme. Das Erlernen von Q_ϕ benötigt die Bewertung des bestmöglichen erreichbaren Folgezustands, was für eine unendliche Anzahl möglicher Aktionen bzw. Folgezustände nicht ohne größere Anpassungen des Lernverfahrens umsetzbar ist. Wird der Agent durch eine Strategie π_θ repräsentiert, muss das Neuronale Netz eine Aktion inklusive ihrer Eintrittswahrscheinlichkeit vorhersagen. Um die bisherigen Algorithmen weiterhin zu verwenden, wäre eine nachträgliche Diskretisierung des Aktionsraums denkbar, indem eine ausreichend hohe Feingranularität gewählt wird. Dies würde bei einem echten Fahrzeug jedoch zu teilweise ruckartigen Bewegungen führen. Daher thematisieren die folgenden Abschnitte das Erlernen stochastischer Strategien mittels normalverteilter Zufallsgrößen. Die Stochastizität ist nicht nur bei der Exploration der Trainingsumgebung nützlich, sondern ermöglicht einer autonomen Fahrsoftware auch, Unsicherheit bei ihren Vorhersagen über die Standardabweichung auszudrücken. Dies kann entscheidend zur Verkehrssicherheit beitragen.

Um das komplette Kontinuum des Aktionsraums abzudecken, sagt π_θ eine Werteverteilung $P[X]$ vorher, dessen Zufallsgröße X im Aktionsraum liegt. Dies wird typischerweise umgesetzt, indem das Modell anstatt einer expliziten Aktion den Mittelwert μ und die Standardabweichung σ vorhersagt, mit denen die Aktion als normalverteilte Zufallsgröße X aus $\mathcal{N}(\mu, \sigma^2)$ gezogen wird. Ein entsprechender Ansatz funktioniert insbesondere auch für mehrdimensionale Aktionsräume $A = (X^{(1)}...X^{(n)})$, indem für jede Dimension k eine Normalverteilung $\mathcal{N}^{(k)}(\mu_k, {\sigma_k}^2)$ mit Zufallsgröße $X^{(k)}$ vorhergesagt wird. Nun muss noch die Wahrscheinlichkeit $\pi_\theta(a_t|s_t)$ bestimmt werden, dass im Zustand s_t die Aktion a_t gezogen wird. Die k-te Wahrscheinlichkeit $\pi(u_k|\mu_k, \sigma_k)$, dass u_k aus der Normalverteilung $\mathcal{N}(\mu_k, {\sigma_k}^2)$ gezogen wird, kann wie folgt berechnet werden.

$$\pi(u_k|\mu_k, \sigma_k) = \frac{1}{\sigma_k\sqrt{2\pi}} \cdot exp(-\frac{(u_k - \mu_k)^2}{2{\sigma_k}^2}) \tag{2.39}$$

Die Wahrscheinlichkeit $\pi_\theta(a_t|s_t) = \prod_{k=1}^{n} \pi(u_k|\mu_k, \sigma_k)$ setzt sich aus dem Produkt aller Wahrscheinlichkeiten $\pi(u_k|\mu_k, \sigma_k)$ der einzelnen Dimensionen des Aktionsraums zusammen. Wie bereits beim Policy Gradient Theorem gesehen, kann der Logarithmus eines Produkts in die Summe der Logarithmen umgewandelt werden.

$$log(\pi_\theta(a_t|s_t)) = log\left(\prod_{k=1}^{n} \pi(u_k|\mu_k, \sigma_k)\right) = \sum_{k=1}^{n} log(\pi(u_k|\mu_k, \sigma_k)) \tag{2.40}$$

Um die Modellparameter der gelernten Strategie π_θ zu optimieren, werden die logarithmierten Wahrscheinlichkeiten jeweils nach den vorhergesagten Mittelwerten μ_k und Standardabweichungen σ_k partiell abgeleitet. Da die Mittelwerte und Standardabweichungen jeweils einem Ausgabeneuron des Modells entsprechen, erfolgt anschließend die Anpassung der Modellparameter θ mittels Backpropagation.

$$\frac{\partial log(\pi_\theta(a_t|s_t))}{\partial \mu_k} = \frac{\partial}{\partial \mu_k} \sum_k log(\pi(u_k|\mu_k, \sigma_k)) = \sum_k \frac{\partial}{\partial \mu_k} log(\pi(u_k|\mu_k, \sigma_k)) \tag{2.41}$$

$$\frac{\partial log(\pi_\theta(a_t|s_t))}{\partial \sigma_k} = \frac{\partial}{\partial \sigma_k} \sum_k log(\pi(u_k|\mu_k, \sigma_k)) = \sum_k \frac{\partial}{\partial \sigma_k} log(\pi(u_k|\mu_k, \sigma_k)) \tag{2.42}$$

Wie zu sehen ist, können die einzelnen Ausgabeneuronen für σ_k und μ_k aufgrund der Summenregel gesondert abgeleitet werden. Zudem hängen die Ableitungen nur von σ_k bzw. μ_k ab, da diese ausschließlich im jeweiligen Term $log(\pi(u_k|\mu_k, \sigma_k))$ vorkommen, sodass die restlichen Summanden durch die Konstantenregel entfallen. Die Auswertungen der Terme $\frac{\partial}{\partial \mu_k} log(\pi(u_k|\mu_k, \sigma_k))$ und $\frac{\partial}{\partial \sigma_k} log(\pi(u_k|\mu_k, \sigma_k))$ führt zu folgenden

Ergebnissen, wie in [19] beschrieben.

$$\frac{\partial}{\partial \mu_k} log(\pi(u_k|\mu_k, \sigma_k)) = \frac{\partial}{\partial \mu_k} \frac{1}{\sigma_k \sqrt{2\pi}} exp\left(-\frac{(u_k - \mu_k)^2}{2{\sigma_k}^2}\right) = \frac{u_k - \mu_k}{\sigma^2} \tag{2.43}$$

$$\frac{\partial}{\partial \mu_k} log(\pi(u_k|\mu_k, \sigma_k)) = \frac{\partial}{\partial \mu_k} \frac{1}{\sigma_k \sqrt{2\pi}} exp\left(-\frac{(u_k - \mu_k)^2}{2{\sigma_k}^2}\right) = \frac{(u_k - \mu_k)^2 - \sigma^2}{\sigma^3} \tag{2.44}$$

Aufgrund der besseren numerischen Stabilität wird in der Praxis oftmals $log(\sigma)$ statt σ vom Neuronalen Netz vorhergesagt und anschließend exponenziert. Zudem kann die Vorhersage um stabilisierende Techniken wie State Dependent Exploration [20] oder dessen Nachfolger Generalized State Dependent Exploration [21] erweitert werden. Hintergrund sind starke Schwankungen, wenn in jedem Zeitschritt eine neue Standardabweichung verwendet wird. Die State Dependent Exploration fixiert deshalb die Standardabweichungen pro Trainingsepisode und modelliert die Standardabweichungen als eine Matrix mit Multivariate Gaussians. Bei der Generalized State Dependent Exploration wird die Fixierung der Standardabweichungen während der Exploration abgewandelt, indem das Zeitintervall nicht mehr eine ganze Episode, sondern eine gewisse Anzahl n an Zeitschritten umfasst. Außerdem werden für die Vorhersage der Mittelwerte und Standardabweichungen nicht mehr zwei gesonderte Modelle, sondern ein einziges Modell verwendet, bei dem sich nur die Neuronen der letzten Schicht unterscheiden.

2.7 Modellbasiertes Lernen

Vor allem für rechenintensive Simulationstechniken kann es sinnvoll sein, die Simulation durch ein Neuronales Netz zu repräsentieren, das effizienter berechnet werden kann. Dies wird als Modellbasiertes Lernen bezeichnet, da der Agent nicht direkt in der echten Simulationsumgebung sondern in einem Modell der Umgebung trainiert. Dies erfordert jedoch, dass ein entsprechendes Modell der echten Simulationsumgebung existiert. Es müssen sowohl die Übergangsfunktion ρ als auch die Wahrnehmungsfunktion ω mit Neuronalen Netzen repräsentiert werden. Viele wichtige Pionierarbeiten wie die World Models [22] gehen auf Schmidhuber et. al zurück und sind seit jüngster Vergangenheit durch die Dreamer Veröffentlichungen [23], [24], [25] von Hafner et. al auf breite Problemstellungen anwendbar.

World Models demonstrieren, wie Neuronale Netze das populäre Computerspiel Doom

und weitere Spiele nachahmen können. Mittels Variational Autoencoder (VAE) [26] wird eine latente Repräsentation für die Wahrnehmungen des Agenten erlernt. Darauf aufbauend lernt ein Dynamikmodell (RNN) die Übergangsfunktion ρ zwischen den latenten Repräsentationen aufeinanderfolgender Zeitschritte. Entscheidend scheint zu sein, dass der VAE in der Lage ist, eine sehr kompakte und flexible Repräsentation der visuellen Wahrnehmungen des Agenten zu lernen. Interessanterweise entstehen bei der Modifikation einzelner Dimensionen der latenten Repräsentation trotzdem noch sinnvolle Bilder, was bei den ungeordneten Repräsentationen herkömmlicher Autoencoder in der Regel nicht der Fall ist. Die Dimensionen der latenten Repräsentation scheinen wie Feature Flags zu funktionieren, die entsprechende visuelle Effekte bei ihrer Aktivierung einblenden. Da es sich bei der Repräsentation um kontinuierliche Features handelt, ist die Aktivierung ebenfalls kontinuierlich, wie in Abbildung 2.6 anhand der 16 Regler aus der interaktiven Visualisierung des trainierten VAE von Doom zu erkennen ist. Durch die semantische Bedeutung der latenten Repräsentation ist auch das Erlernen eines großen Dynamikmodells mit vielen trainierbaren Parametern möglich. Im Kontext des Bestärkenden Lernens kann das Modell des Agenten auf ein Minimum reduziert werden, da die Extraktion und Aufbereitung hochwertiger Informationen aus Videosequenzen durch den Encoder und das Dynamikmodell erfolgen. Unter anderem kann erstmalig ein anspruchsvolles Videospiel zur Steuerung eines Rennautos auf einer kurvigen Rennstrecke anhand von Videodaten aus der Vogelperspektive erlernt werden.

Darauf aufbauend entwickelt Hafner et. al die Dreamer-Architektur als Erweiterung der World Models und wendet diese erfolgreich auf die MuJoCo Spiele zum Erlernen der Fortbewegung mit humanoiden Körpern an. Insbesondere ist Dreamer ebenfalls in der Lage, die Spiele anhand von Videodaten aus der Vogelperspektive zu lernen, was zuvor nur mit sehr spezialisierten und ressourcenintensiven Lernverfahren möglich war. Namesgebend für Dreamer ist die Idee, dass während des Träumens anhand von echten Erfahrungen hypothetische Szenarien halluziniert werden. Wie bei den World Models lernt der Agent nicht in der echten Simulation, sondern in Träumen, die mit einer kurzen Sequenz aus echten Situationen starten und dann durch das Dynamikmodell fortgeführt werden, je nachdem welche Aktionen der Agent auswählt.

Unterschiede zwischen den World Models und der Dreamer-Architektur bestehen vor allem in der latenten Repräsentation und dem zugehörigen VAE, um diese zu extrahieren. Dreamer verwendet seit Version 2 eine kategorische anstatt einer kontinuierlichen Repräsentation. Hintergrund hierfür ist, dass die damit verbundene Diskretisierung

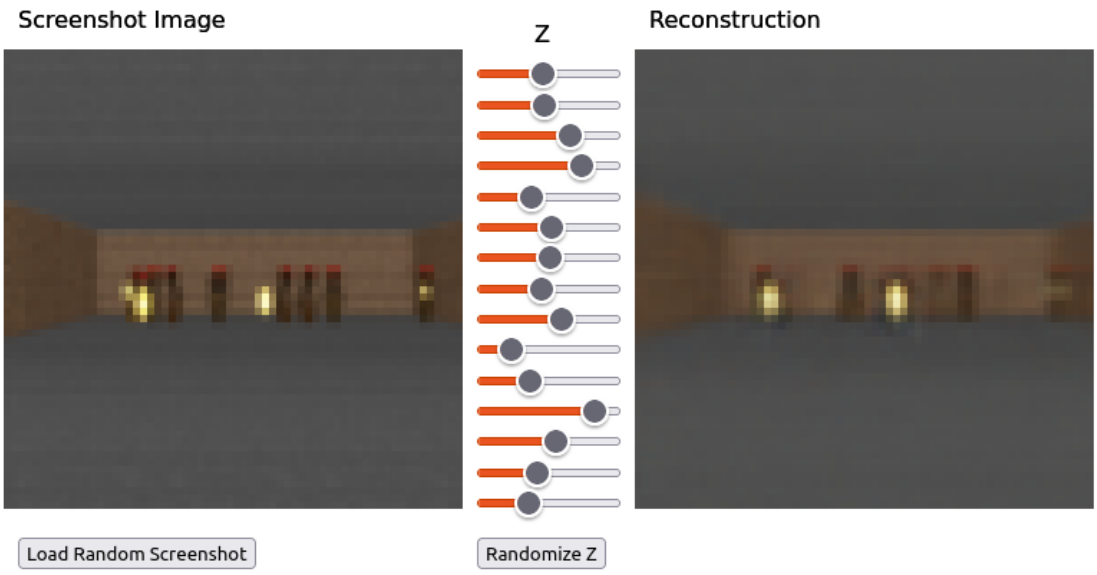

Abbildung 2.6: Latente Repräsentation von DoomViz aus der Live-Demonstration der World Models [22], [27]

das Erlernen von Konzepten erzwingt. Mit 32 Klassifikationen und jeweils 32 Klassen ergeben sich 32^{32} mögliche Repräsentationen, was immer noch mehr als ausreichend ist, um vielfältige Situationen abzubilden. Zudem enthält die latente Repräsentation selbst bereits Informationen bezüglich der Dynamiken der wahrgenommenen Videosequenz, indem der Zustand der vom Dynamikmodell verarbeiteten Videosequenz vorheriger Standbilder konkateniert wird. Dies stellt dem Agent hochwertige Informationen bezüglich der Dynamiken seiner Umgebung zur Verfügung.

Ein weiterer, wichtiger Unterschied besteht in der Art, wie die Modelle trainiert werden. Im Gegensatz zu den World Models lernt Dreamer nicht zuerst einen VAE, dann ein Dynamikmodell und trainiert schließlich den Agent. Stattdessen wird bereits während VAE und Dynamikmodell noch lernen auch der Agent trainiert. Dies ist notwendig, da der Agent für das Sammeln der Trainingsdaten zur Anpassung des Umgebungsmodells verwendet wird, um mit der Zeit brauchbare Repräsentationen der Realität und ein gutes Dynamikmodell zu erhalten. Damit die echte Trainingsumgebung möglichst ganzheitlich exploriert wird, ist es insbesondere nicht ausreichend, zufällige Aktionen auszuführen, weshalb ein bereits gut trainierter Agent benötigt wird. Daher trainieren Umgebungsmodell und Agent ähnlich wie Generator und Discriminator bei GANs immer abwechselnd, um sich mit der Zeit gegenseitig zu steigern.

3 Analyse bestehender Ansätze

In diesem Kapitel werden bereits existierende Ansätze zum Erlernen einer Fahrsoftware für Mikromobilitätsfahrzeuge beschrieben. Speziell geht es einerseits um den Ansatz *RobotSF* [1], bei dem ein Fahrzeug mit Fußgängern interagiert, die per Social Force simuliert werden. Alternativ gibt es auch *MultiRobot* Ansätze [4], [5], die stattdessen viele autonome Fahrzeuge miteinander interagieren lassen, anstatt die beweglichen Hindernisse durch Fußgänger zu simulieren.

3.1 Ansätze bezüglich des Lernverfahrens

Gemeinsam haben die Ansätze, dass meist ein oder mehrere Roboter der Mikromobilitätsklasse per Deep Reinforcement Learning trainiert werden. Als Trainingsalgorithmen kommen oftmals Policy Gradient Methoden wie beispielsweise Asynchronous Advantage Actor-Critic (A3C) oder Proximal Policy Optimization (PPO) zu Einsatz, da diese gut mit Parallelität und Rechenressourcen skalieren. Im Fall von rechenintensiven Simulationen können auch Lernverfahren wie Deep Deterministic Policy Gradient (DDPG) und Soft Actor-Critic (SAC) interessant sein, um eine bessere Trainingsdateneffizienz zu erreichen [28].

3.2 Ansätze bezüglich Fahrzeug und Sensorik

Bezüglich der verwendeten Fahrzeuge kommen verschiedenste Modelle zum Einsatz. Je nach Fahrzeug bietet sich eine unterschiedliche Kinematik zur Simulation an. Beispielsweise eignet sich das Bicycle Model, um E-Scooter zu modellieren. Für Fahrzeuge mit Allradantrieb oder Ketten kommt Differential Drive infrage. Fan et. al [4] geht sogar so weit, einen trainierten Agent bewusst mit einer abgewandelten Kinematik zu evaluieren, um die Robustheit der erlernten Fahrsoftware zu testen. Als Sensorik wird in der Mikromobilität meist ein LiDAR-Sensor in Kombination mit einer Zielpeilung verwendet, was alle 3 betrachteten Veröffentlichungen gemeinsam haben.

3.3 Ansätze bezüglich der Simulationsumgebung

Für die Repräsentation der Entitäten und des Kartenmaterials bei der Simulation gibt es auch vielerlei Ansätze [28]. Unter anderem werden Entitäten mit Vektorgrafiken dargestellt. Ein weiterer Ansatz diskretisiert die Karte und alle Entitäten als Raster (engl. Occupancy Grid). Weitere Ansätze wie CALRA [29] setzen auf fotorealistische Grafiken mittels optimierter Spieleframeworks wie der Unreal Engine. Der jeweilige Ansatz hat großen Einfluss auf die verwendeten Datenstrukturen und Algorithmen, um die Entitäten und Sensoren zu simulieren.

3.4 Ansätze bezüglich der Evaluation

Zur Evaluation der trainierten Modelle gibt es noch keine einheitlichen Standards [28]. Dies macht es sehr schwer, die verschiedenen Ansätze der Forscher fair zu vergleichen. Eine Vergleichbarkeit ist aber ohnehin kaum herstellbar, da die Vielfalt bezüglich Fahrzeugen und Sensorik teilweise unüberwindbare Inkompatibilitäten verursacht. Als Metrik werden meist Erfolgs- bzw. Unfallraten für die relative Häufigkeit an Zielfahrten bzw. Unfällen verwendet.

4 Konzeption und Umsetzung

In diesem Kapitel wird auf die Konzeption und Umsetzung der Trainingsumgebung eingegangen, um die vom Agent während der Trainingsläufe erlernten Verhaltensweisen zu steuern. Dies umfasst unter anderem den verwendeten Simulator, die Fahrzeugsensorik und -kinematik, die verwendeten Modelle zur Umsetzung einer Fahrsoftware inklusive zugehöriger Lernverfahren und die Gestaltung des Kartenmaterials.

4.1 Diskussion der Konzeption

Im folgenden Abschnitt werden die Vor- und Nachteile der Ansätze *RobotSF* und *Multi-Robot* diskutiert. Anschließend wird ein Konzept für die Umsetzung der Trainingsumgebung entwickelt, in der später die Trainingsläufe durchgeführt werden sollen. Für die Diskussion werden jeweils die Veröffentlichungen Caruso et. al [1] für *RobotSF* und Fan et. al [4] bzw. Shunyi et. al [5] für *Multi-Robot* herangezogen. Zudem dient Kiran et. al [28] als Überblick, um weitere, populäre Ansätze zusammenzufassen.

4.1.1 Umsetzung dynamischer Hindernisse

Der entscheidendste Unterschied zwischen den *Multi-Robot* und *RobotSF* Ansätzen besteht in der Modellierung dynamischer Hindernisse. Das Konzept von *Multi-Robot* sieht hierzu vor, die dynamischen Hindernisse in Form von vielen autonomen Fahrzeugen umzusetzen. Hingegen implementiert *RobotSF* dynamische Hindernisse anhand von Fußgängern, die mit dem Social Force Modells gesteuert werden.

Da diese Arbeit die Interaktion zwischen dem Fahrzeug und Fußgängern auf Gehwegen und in Fußgänerzonen thematisiert, passt der Ansatz von *RobotSF* deutlich besser. Eine Kombination der beiden Ansätze wäre jedoch möglich. Es wird entschieden, zunächst Fußgänger durch Social Force umzusetzen und erst zu einem späteren Zeitpunkt die Simulationsumgebung um mehrere autonome Fahrzeuge zu erweitern.

4.1.2 Modellierung des Kartenmaterials

Zur Modellierung der simulierten Entitäten in Form des Fahrzeugs, der Fußgängern und der statischen Hindernisse kommen zwei Ansätze aus der Grafikprogrammierung infrage. Die Entitäten können zum einen durch Vektorgrafiken repräsentiert werden. Zum anderen ist es auch möglich, die Karte in eine Rasterstruktur einzuteilen und die entsprechenden Entitäten durch Occupancy Grids darzustellen.

Diese Arbeit strebt eine Validierung des Fahrzeugs auf dem virtuellen Campus der Universität Augsburg an. Occupancy Grids sind aufgrund der festen Rastergröße nur schlecht skalierbar und können daher nur sehr bedingt große Karten darstellen. Außerdem liegen die Entitäten bei populären Implementierungen des Social Force Modells wie beispielsweise PySocialForce [30] ohnehin in Form von Vektorgrafiken vor. Anhand des auf GitHub veröffentlichten Quelltexts [31] des *RobotSF* Ansatzes ist ersichtlich, dass hier PySocialForce verwendet wird, jedoch in Kombination mit Occupancy Grids. Dies erscheint sehr umständlich zu sein, da zu jedem Simulationsschritt zwischen den Repräsentationen als Vektorgrafiken und Occupancy Grids konvertiert werden muss.

Es wird daher entschieden, die Umsetzung von *RobotSF* als Basis zu verwenden, aber die Repräsentation der Entitäten auf Vektorgrafiken umzustellen. Die Fahrzeuge und Fußgänger werden als Kreise dargestellt. Hindernisse entsprechen jeweils einem aus einzelnen Linien zusammengesetzten Polygon. Da diese Repräsentation der Umsetzung aus PySocialForce entspricht, wird dessen Schnittstelle zum Simulator deutlich vereinfacht. Das entsprechende Kartenmaterial wird von OpenStreetMap importiert, sodass die Gebäudeumrisse als Hindernisse dienen. Anschließend wird die Karte um dynamische Hindernisse in Form der Fußgänger ergänzt.

4.1.3 Umsetzung der simulierten Fahrzeuge

Bei der Simulation von Mikromobilitätsfahrzeugen kommt häufig die Differential Drive Kinematik zum Einsatz. Eine weitere, sehr einfache Kinematik stellt das Fahrradmodell dar. Alle Umsetzungen von *RobotSF* als auch von *Multi-Robot* verwenden übereinstimmend die Differential Drive Kinematik. Dies liegt daran, dass jeweils ein kleiner allradgetriebener Roboter in der Realität erprobt wird, der eine entsprechende Kinematik aufweist.

Um aber beispielsweise auch Fahrzeugtypen wie einen E-Scooter adäquat modellieren

zu können, scheint das kinematische Fahrradmodell eine sinnvolle Alternative darzustellen. Folglich wird entschieden, sowohl das Differential Drive Modell, als auch das kinematische Fahrradmodell umzusetzen. Außerdem soll die Schnittstelle zwischen Fahrzeug und Simulator für die Unterstützung weiterer Fahrzeugkinematiken offen gehalten werden, um eine möglichst große Bandbreite an Fahrzeugen zu unterstützen. Die Implementierung von *RobotSF* für Differential Drive wird übernommen. Für das kinematische Fahrradmodell dient der auf GitHub veröffentlichte Quelltext von Winston H. [32] als Vorlage.

4.1.4 Umsetzung der Fahrsoftware

Für die Umsetzung der Steuersoftware hinsichtlich der Lokalen Navigation müssen alle Aufgaben, die sonst ein menschlicher Fahrer ausführt, durch die Software übernommen werden. Dies umfasst unter anderem die Steuerung des Fahrzeugs, indem beschleunigt, gebremst und gelenkt wird. Hierfür muss die Software den Lenkwinkel und die Beschleunigung mehrmals pro Sekunde vorgeben. In die Bestimmung dieser beiden Stellgrößen geht eine Vielzahl an Informationen ein, die teilweise aus hochdimensionalen Sensordaten extrahiert und anschließend entsprechend verarbeitet werden müssen. Beispielsweise beobachtet die Fahrsoftware die Positionen und Dynamiken anderer Verkehrsteilnehmer mittels Kameras und/oder LiDAR-Sensoren, um Kollisionen zu vermeiden. Außerdem sind für die Einhaltung der Verkehrsregeln eine Fülle an weiteren Informationen wie beispielsweise die Positionen und Phasen von Ampeln oder die Erkennung von Verkehrsschildern erforderlich.

Für einige Teilaufgaben des Autonomen Fahrens gibt es bewährte Lösungen, wie die Lenkung per Stanley-Controller oder die Geschwindigkeitskontrolle mittels PID-Controller, welche jedoch oftmals bei schneller Fahrt oder Notbremsungen an ihre Grenzen kommen. Die Wahrnehmung der Umgebung wird typischerweise mit Neuronalen Faltungsnetzen (engl. Convolutional Neural Networks, CNN) gelöst. Da sich die nicht-lineare Funktionsapproximation mittels Neuronaler Netze sehr gut für die Auswertung hochdimensionaler Sensordaten und die Vorhersage komplexer Stellgrößen eignet, ist es naheliegend, auch die restlichen Aufgaben mittels Neuronaler Netze umzusetzen. Dies wird auch übereinstimmend von allen betrachteten Implementierungen der Ansätze *RobotSF* und *Multi-Robot* bestätigt, die ihre Fahrsoftware als Neuronale Netze repräsentieren und per Deep Reinforcement Learning trainieren. Es wird daher entschieden, die Fahrsoftware mittels Tiefer Neuronaler Netze umzusetzen.

4.1.5 Umsetzung der Sensorik

Als Fahrzeugsensorik benötigt der Agent sowohl eine Zielpeilung, als auch eine Wahrnehmung seiner Umgebung. Außerdem muss er für die Steuerung des Fahrzeugs dessen aktuelle Dynamik kennen. Als Zielpeilung dient meist die relative Zieldistanz und der Winkel zum Ziel. Für die Wahrnehmung kommen hauptsächlich Kameras und/oder LiDAR-Sensoren infrage. Da es sich bei der Mikromobilität oft um kleine Fahrzeuge handelt, für deren Umsetzung hochwertige Kameras viel teuer und fehleranfälliger wären, werden typischerweise nur LiDAR-Sensoren eingesetzt.

Eine Umsetzung der Wahrnehmung durch einen LiDAR-Sensor ist auch übereinstimmend mit allen betrachteten Ansätzen bezüglich *RobotSF* und *Multi-Robot*. Vermutlich liegt dies daran, dass eine entsprechende Simulationsumgebung deutlich leichter umzusetzen ist, da keine hochauflösenden Grafiken berechnet werden müssen, wie es andernfalls für Kameras notwendig wäre. Bezüglich der Zielpeilung werden auch ähnliche Ansätze mit relativen Zieldistanzen und -winkeln gewählt.

Es wird daher entschieden, die Sensorik für die Wahrnehmung mit einem LiDAR-Sensor und die Zielpeilung durch eine relative Zieldistanz und den relativen Winkel zum Ziel umzusetzen. Die Dynamik des Fahrzeugs entspricht der vom kinematischen Fahrzeugmodell bereitgestellten Sensorik und wird dort gekapselt.

4.1.6 Umsetzung der Navigation und Fußgängersteuerung

Nach einer Begutachtung von *RobotSF* wird klar, dass die dort vorgesehene Navigation des Fahrzeugs die Problemstellung deutlich erschwert, da die Route lediglich aus Start- und Endpunkten besteht, die zufällig auf der Karte positioniert werden. Wenn die Punkte am anderen Ende der Karte hinter möglichen Hindernissen liegen, können sehr schwierige Szenarien entstehen. Moderne, kartengestützte Navigationsgeräte zur Globalen Navigation können deutlich kleinteiliger interpolierte Routen mit vielen Wegpunkten vorgeben. Eine Vorgabe der Route anhand von vielen Wegpunkten scheint daher sinnvoll, um die Komplexität bei der Lokalen Navigation durch die Fahrsoftware zu senken.

Die Umsetzung von *RobotSF* bei der Steuerung von Fußgängern weist außerdem erhebliche Schwächen auf. Ähnlich wie bei der Vorgabe der Routen für das Fahrzeug werden auch hier zufällige Start- und Zielpunkte auf der Karte gewählt. Dies führt jedoch zu

Problemen, da sich mittels Social Force gesteuerte Fußgänger immer nur geradlinig auf ihr Ziel zu bewegen und daher nicht um größere Hindernisse wie beispielsweise Gebäude herummanövrieren können. Um ein typisches Fußgängerverhalten auf Gehwegen zu erzielen, müssen auch hier Routen anhand von mehreren Wegpunkten vorgegeben werden, damit die Fußgänger Stecken laufen können, die Kurven enthalten. Das bisherige Fußgängerverhalten kann beibehalten werden, um Fußgängerzonen zu modellieren. Der entsprechende Bereich soll jedoch durch eine feste Zone beschränkt werden, um die eingangs geschilderte Problematik von Social Force mit großen Hindernissen zu vermeiden.

Da die bisherige Fußgängersteuerung keine Fußgängerzonen und -routen vorsieht, wird entschieden, die Implementierung von *RobotSF* entsprechend zu überarbeiten. Auch die Gruppierungslogik der Fußgänger wird gemäß Moussaïd et. al [7] angepasst. Zudem wird die Navigationsaufgabe des Fahrzeugs ebenfalls durch aus mehreren Wegpunkten bestehenden Routen vereinfacht.

4.1.7 Auswahl der Lernverfahren

Als Lernverfahren kann entweder eine Belohnungsfunktionen $Q\phi$ bzw. $V\phi$ oder eine Strategie π_θ mittels Bestärkendem Lernen approximiert werden. Da es sich beim Steuern eines Fahrzeugs um ein kontinuierliches Kontrollproblem handelt, werden entsprechende Aufgaben typischerweise mit einer stochastischen Strategie umgesetzt. In allen betrachteten Ansätzen kommen deshalb Varianten der Policy Gradient Methoden in Form von A3C bzw. PPO zum Einsatz. Dies ist hauptsächlich durch die bessere Skalierbarkeit mit Rechenressourcen zu begründen. Caruso et. al [1] testet zusätzlich einen Ansatz mit DQN, der jedoch in allen Kategorien deutlich schlechter als die mit A3C erlernte Strategie abschneidet.

Hinsichtlich der Trainingsdateneffizienz sind für gewöhnlich Verfahren wie DDPG und SAC [28] vorzuziehen, da diese das Modell bereits während dem Sammeln der Trainingsdaten aktualisieren und gesammelte Erfahrungen vielmals durch ein Prioritized Replay Memory für Modellaktualisierungen wiederverwenden. Aufgrund der besseren Eignung für den konkreten Anwendungsfall des Autonomen Fahrens und der Beiträge von PPO zur Steigerung der Trainingsdateneffizienz und Stabilität kann es dennoch sinnvoll sein, Policy Gradient Methoden heranzuziehen. Dies ist insbesondere der Fall, wenn die Simulationsumgebung nicht zu rechenintensiv ist und entsprechende Rechenres-

sourcen vorliegen, um die Skalierbarkeit moderner Hardware auszunutzen. Für Policy Gradient Methoden spricht außerdem, dass eine Strategie statt einer Schätzung der Belohnungsfunktion gelernt wird, wodurch Unsicherheiten bezüglich der gewählten Aktionen ausgedrückt werden können. Dies kann unter anderem bei der Falsifizierung trainierter Fahragenten sehr nützlich sein, um Schwachstellen der Fahrsoftware zu ermitteln und anschließend zu beheben.

Es wird deshalb der Entschluss gefasst, eine Strategie mithilfe von Policy Gradient Methoden zu erlernen. Als konkrete Algorithmen werden PPO und A3C näher in Betracht gezogen. Wie bereits in Abschnitt 2.5.2 angedeutet, ist PPO eine Weiterentwicklung von A3C und trainiert deutlich stabiler und effizienter mit den gesammelten Trainingsdaten, da mehrere Lernschritte auf denselben Daten durchführbar sind. Es wird daher PPO als Lernverfahren gewählt.

4.1.8 Umsetzung der Modellstruktur

Bei der Umsetzung einer Lokalen Navigation auf Basis von Neuronalen Netzen mittels LiDAR-Sensorik kommen typischerweise Neuronale Faltungsnetze, gefolgt von einigen vollvermaschten Neuronenschichten zum Einsatz. Um Bewegungen abzubilden, werden die Standbilder mehrerer aufeinanderfolgender Zeitschritte zusammengefügt.

Der Fahragent wird durch ein Neuronales Netz modelliert, dessen Actor- und Critic-Komponente jeweils aus zwei vollvermaschten Schichten mit 64 Neuronen bestehen. Der Actor repräsentiert die Policy, die pro Aktuator einen Mittelwert und eine Standardabweichung vorhersagt, womit normalverteilte Zufallswerte gezogen werden. Hingegen schätzt der Critic die Grundbelohnung (Baseline) des aktuellen Zustands, was einer Regression mit einem einzigen Ausgabeneuron entspricht. Als Eingabe erhalten Actor und Critic den von den Sensoren gelieferten, aktuell beobachtbaren Systemzustand, der aus der Dynamik des Fahrzeugs, der Zielpeilung und den vom Strahlensensor gemessenen Abständen zu Hindernissen besteht. Um Hindernisse besser erkennen zu können, werden nebeneinander liegende Strahlen per 1D Faltung zu aussagekräftigeren Mustern zusammengefasst. Die jeweiligen Zeitschritte werden mithilfe der Eingabekanäle repräsentiert. Alternativ können auch 2D Faltungen zum Einsatz kommen, sodass die Faltungskerne zeitschrittübergreifende Muster abtasten, was jedoch aufgrund der kleinen Anzahl an Zeitschritten verworfen wird, da es keinen Vorteil gegenüber 1D Faltungen bietet. Auch die Fahrdynamiken und Zielpeilungen werden für mehrere Zeitschritte geliefert und

per Flatten-Schicht zu einem flachen Eingabevektor verarbeitet, der mit den flachen, vorverarbeiteten Strahlendaten konkateniert wird.

Eine entsprechende Vorverarbeitung der Strahlendaten erfolgt in einem sog. Feature Extractor, der aus 4 Faltungsschichten besteht, jeweils gefolgt von einer Rectified Layer Unit (ReLU) Aktivierung und einer Dropout-Schicht, um Überanpassungen zu vermeiden. Da die vollvermaschten Neuronenschichten von Actor und Critic flache Feature-Vektoren erwarten, enthält der Feature Extractor eine abschließende Flatten-Schicht. Um die Eingabedimensionen darauffolgender Schichten zu reduzieren, werden im Feature Extractor jeweils die Ausgabedimensionen bezüglich der Eingabedimensionen der Faltungen halbiert. Dies erfordert, dass die Anzahl der vom LiDAR-Sensor gemessenen Strahlen durch 16 teilbar ist. Als Anzahl der Filter wird für die ersten beiden Faltungen 64 und für die letzten beiden Faltungen 16 gewählt. Die Faltungskerne weisen eine Größe von 3 auf.

Sehr ähnliche Umsetzungen der Modellstruktur kommen auch in den Ansätzen von *RobotSF* und *Multi-Robot* jeweils zum Einsatz. Die exakten Modellstrukturen weichen dabei etwas voneinander ab, haben aber gemeinsam, dass das Modell die LiDAR-Daten per Feature Extractor in Form einiger Faltungsschichten vorverarbeitet und anschließend die Actor- und Critic-Komponente durch vollvermaschte Neuronenschichten repräsentiert. Da Fan et. al [4] das Fahrverhalten aus einer offensiven und defensiven Strategie zusammensetzt, wird ein defensives Modell ohne Feature Extractor konzipiert, das nur Standbilder sieht. Dem offensiven Modell wird hingegen ein Feature Extractor bereitgestellt, der die Sensordaten mehrerer Zeitschritte als Eingabe erhält.

4.1.9 Auswahl der Belohnungsstruktur

Da der Agent das Ziel verfolgt, die während des Trainings erhaltenen Belohnungen zu maximieren, kommt der Wahl der Belohnungsstruktur eine zentrale Rolle zu, um die erlernten Fahrverhaltensweisen zu steuern.

Die Umsetzung der Belohnungsstruktur von *RobotSF* sieht eine große, punktuelle Belohnung für das Erreichen des Ziels vor. Außerdem erhält der Agent häufige, kleine Belohnungen entsprechend der Distanzverkürzung zum Ziel, um Annäherungen an das Ziel anzuregen. Des weiteren wird eine Strafe vergeben, wenn sich innerhalb eines der 8 gleichmäßig um den Agent angeordneten Sektoren Fußgänger befinden.

Die Bestrafung wird pro Sektor bestimmt und wächst gemäß der sinkenden Distanz zum nächstgelegenen Fußgänger im Sektor. Dadurch soll erzwungen werden, dass der Agent einen Mindestabstand zu Fußgängern einhält. Zudem wird der Agent bestraft, wenn er eine Kollision mit Fußgängern oder statischen Hindernissen verursacht. Bei der von Fan et. al [4] beschriebenen Umsetzung von *Multi-Robot* gibt es ebenfalls eine Belohnung für das Erreichen des Ziels und eine Bestrafung für Kollisionen mit statischen und dynamischen Hindernissen. Außerdem werden auch schon beinahe aufgetretene Kollisionen gemäß der Distanz zwischen Fahrzeug und Hindernis bestraft. Um eine Annäherung an das Ziel anzuregen wird statt einer Belohnung für Annäherungen eine Bestrafung für die Vergrößerung der Zieldistanz vergeben.

Die aus den beiden Ansätzen resultierenden Belohnungssignale sind recht komplex und teilweise nur schwer für den Agent anhand der ihm zur Verfügung stehenden Sensordaten interpretierbar. Werden die Terme der Belohnungsstruktur falsch gewichtet, kann dies schnell zu starken Fehlanreizen führen. Um zu beweisen, dass geeignetes Fahrverhalten bereits mit einer sehr einfachen Belohnungsstruktur erzielbar ist, wird eine eigene Belohnungsfunktion folgendermaßen definiert: Der Agent erhält eine Belohnung von 1 für das Erreichen des Ziels, eine Bestrafung von -2 für eine Kollision mit einem Fußgänger oder einem Hindernis und zusätzlich eine insgesamt über die einzelnen Simulationsschritte verteilte, sehr kleine Bestrafung von -0.1, die dazu anregen soll, möglichst schnell das Ziel zu erreichen.

$$r(s_t) = \frac{-0.1}{\text{max\_steps}} + \begin{cases} 1 & ,\text{reached\_waypoint}(s_t) \\ -2 & ,\text{is\_collision}(s_t) \\ 0 & ,\text{else} \end{cases} \tag{4.1}$$

Die Bestrafung für Kollisionen wird bewusst gleich für Fußgänger und Hindernisse gewählt, da eine Unterscheidung aus den LiDAR-Daten eines Standbilds für den defensiven Agent kaum möglich ist und außerdem dazu anregen würde, die im Kartenmaterial vorkommenden Hindernisumrisse ggf. auswendig zu lernen. Beim offensiven Agent kann eine Unterscheidung zwischen Fußgängern und statischen Hindernissen sinnvoll sein, da ihm die Sensordaten mehrerer Zeitschritte zur Verfügung stehen, woraus die Bewegungsdynamiken der Fußgänger ersichtlich sind. Um eine aus mehreren Wegpunkten zusammengesetzte Route fahren zu können, entspricht das Anfahren jedes einzelnen Wegpunkts einer neuen Episode. Da bei der Erstellung der Routen darauf geachtet wird, dass sich die ersten Wegpunkte nah beim Startpunkt des Fahrzeugs befinden und

anhand der Positionierung der Fußgänger keine Interaktion zu erwarten ist, entfällt die Notwendigkeit einer Belohnung für Zielannäherungen. Aufgrund der Funktionsweise von Advantage Actor-Critic Methoden wie PPO ist ein exaktes Erlernen einer komplexen Belohnungsstruktur nicht erforderlich. Es genügt eine ungefähre Schätzung des Advantage mit korrektem Vorzeichen, damit die Wahrscheinlichkeiten zur Auswahl von Aktionen mit der Zeit entsprechend erhöht bzw. gesenkt werden.

4.1.10 Wahl der Evaluationsmetriken

Um die Qualität der trainierten Fahragenten zu bestimmen, werden in Anlehnung an den *RobotSF* Ansatz von Caruso et. al [1] dieselben Unfallmetriken gewählt, um später eine Vergleichbarkeit der Ergebnisse herzustellen.

Die Unfallmetriken messen die relativen Häufigkeiten, wie oft eine Route von Anfang bis Ende unfallfrei gefahren wird bzw. wie häufig Unfälle mit Fußgängern oder Hindernissen auftreten. Es findet eine Unterscheidung in 4 disjunkte Kategorien statt. Unfallfrei bis zum Ende gefahrene Routen werden mit der *Route Completion Rate* gemessen. Hingegen messen die *Pedestrian Collision Rate* und *Obstacle Collision Rate* Unfälle mit Fußgängern bzw. stationären Hindernissen. In allen weiteren Fällen tritt zwar kein Unfall auf, aber der Agent findet auch nicht innerhalb der dafür vorgesehenen Simulationszeit ans Ziel der Route, was mit der *Timeout Rate* bemessen wird.

Weitere Veröffentlichungen bezüglich des *Mulit-Robot* Ansatzes verwenden lediglich eine sog. *Success Rate* für zu Ende gefahrene Routen, was in den Unfallmetriken bereits enthalten ist.

4.2 Konzeption der Simulationsumgebung

Nachdem die grundlegenden, konzeptionellen Entscheidungen für die Umsetzung des Simulators getroffen sind, werden nun die Datenstrukturen und Algorithmen im Detail ausgearbeitet. Die Umgebung simuliert ein steuerbares Fahrzeug, das statischen und dynamischen Hindernissen ausweichen soll. Bei den Hindernissen handelt es sich jeweils um stationäre Gebäude und Fußgänger, die sich in einer 2D Ebene bewegen. Die Aktuatoren und Sensoren des Fahrzeugs werden in Form eines Markov Decision Process bereitgestellt, um die Steuerung des Fahrzeugs durch einen Agent zu ermöglichen. Im folgenden werden Aufbau und Funktionsweise der Simulationsumgebung beschrieben.

4.2.1 Simulationsablauf

Zu Beginn wählt die Simulationsumgebung aus einer Menge vorgegebener Routen eine zufällige Route aus und positioniert das Fahrzeug in der Startzone der Route. Anschließend steuert der Agent das Fahrzeug von Wegpunkt zu Wegpunkt, bis es am Routenziel ist. Um das Auswendiglernen der Route zu vermeiden, werden Start- und Zielpositionen zufällig variiert. Ein Wegpunkt gilt als erreicht, wenn sich das Fahrzeug zu einem diskreten Simulationszeitpunkt nah genug am Wegpunkt befindet. Kollidiert das Fahrzeug hingegen mit einem Fußgänger oder einem Hindernis, wird die Navigation der aktuellen Route abgebrochen. In diesem Fall oder beim Erreichen des Routenziels startet die Navigation einer neuen Route.

Die Simulation setzt sich aus einer Abfolge vieler, kleiner Simulationsschritte zusammen, die jeweils den Systemzustand zu einem diskreten Zeitpunkt t repräsentieren. Zwischen zwei aufeinanderfolgenden Zeitschritten vergeht Zeit in Höhe des Zeitintervalls Δt, das je nach Anzahl der vom Fahrzeug durchführbaren Aktionen, der sog. Aktionsfrequenz, konfiguriert werden kann. Pro Zeitschritt wird zunächst von außen eine Aktion für das Fahrzeug gewählt, mit der das Fahrzeug entsprechend seiner Kinematik fährt. Daraufhin bewegen sich alle Fußgänger gemäß der Kräfte des Social Force Modells. Abschließend wird der neue Systemzustand bestimmt, mithilfe der Fahrzeugsensorik aus Sicht des Fahrzeugs erfasst und nach außen propagiert, damit die nächste Aktion gewählt werden kann.

4.2.2 Aktuatoren und Action Spaces

Die Aktuatoren modellieren die Kontrolle bezüglich der lateralen und longitudinalen Bewedungsdynamik des Fahrzeugs zum Beschleunigen, Bremsen und Lenken. Diese sind spezifisch für die zugehörige Fahrzeugkinematik.

Im Fall des Differentialgetriebenen Fahrens werden die Aktuatoren als lineare Geschwindigkeit v und Winkelgeschwindigkeit ω definiert 2.1.2. Folglich ergibt sich ein Action Space von $[0, v_{max}]$ für die lineare Geschwindigkeit und ein Action Space von $[-\omega_{max}, \omega_{max}]$ für die Winkelgeschwindigkeit des Fahrzeugs. Da sich das Fahrzeug auf der Stelle drehen kann, ist kein Rückwärtsfahren vorgesehen.

Beim Fahrradmodell werden die Aktuatoren als Beschleunigung a und Lenkwinkel δ modelliert. 2.1.1 Somit ergibt sich ein Action Space von $[-a_{max}, a_{max}]$ für die Be-

schleunigung und ein Action Space von $[-\delta_{max}, \delta_{max}]$ für den Lenkwinkel des Fahrzeugs. Aufgrund der negativen Beschleunigung kann rückwärts gefahren werden, was jedoch deaktivierbar ist.

4.2.3 Sensoren und Observation Spaces

Als Sensoren kommen ein radialer Strahlensensor (LiDAR), ein Positionssensor (GPS) und ein Sensor für die Bewegungsdynamik des Fahrzeugs zum Einsatz.

Eine Observation setzt sich aus mehreren Komponenten zusammen. Sie besteht zum einen aus der Bewegungsdynamik des Fahrzeugs, zum anderen aus der relativen Position des Fahrzeugs zum angepeilten Zielpunkt als Polarvektor. Zusätzlich gehen die Entfernungen des Strahlensensors in die Observation ein. Somit ergeben sich für die Bewegungsdynamik des Fahrzeugs Observation Spaces $[0, v_{max}]$ für die Geschwindigkeit und $[-\omega_{max}, \omega_{max}]$ bzw. $[-\phi_{max}, \phi_{max}]$ für die Lenkung ähnlich zu den Aktuatoren.

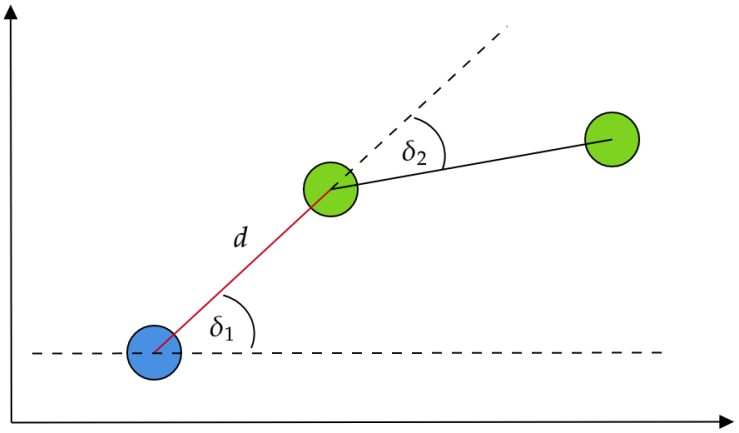

Abbildung 4.1: Sensorik der Zielpeilung zwischen dem Fahrzeug (blau) und den nächsten beiden Wegpunkten (grün). Dem Agent steht das Tupel (d, δ_1, δ_2) bezüglich Zieldistanz, Trajektorie und Orientierung im Koordinatensystem zur Verfügung.

Die Observation Spaces der Zielpeilung sind $[0, d_{max}]$, $[-\pi, \pi]$, wobei die maximal mögliche Zielentfernung $d_{max} = \sqrt{(h_{map})^2 + (w_{map})^2}$ anhand der Ausmaße der Karte bzgl. deren Breite w_{map} und Höhe h_{map} beschrieben wird. Der LiDAR-Sensor sendet seine Strahlen gleichmäßig verteilt über den Öffnungswinkel φ aus. Jeder Strahl wird durch einen Observation Space von $[0, s_{max}]$ mit fester, maximaler Scanreichweite s_{max} repräsentiert. Um die Route zielgerichteter fahren zu können, wird die Peilung des

übernächsten Ziels als Winkel zwischen der Fahrzeugposition und dem nächsten und übernächsten Zielpunkt bereitgestellt, wie in Abbildung 4.1 zu sehen ist. Dies entspricht ebenfalls einem Observation Space von $[-\pi, \pi]$.

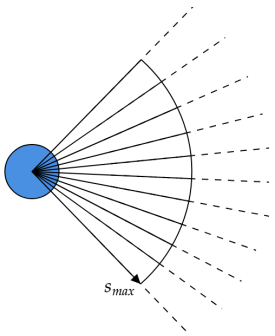

Abbildung 4.2: Schaubild eines LiDAR-Sensors mit Öffnungswinkel φ und maximaler Scandistanz s_{max}.

Da es unmöglich ist, die Bewegungsdynamiken der Fußgänger aus einem Standbild erfasster Entfernungen durch den Strahlensensor abzuleiten, stehen dem Fahrzeug immer mindestens die Strahlendaten der letzten 3 Zeitschritte zur Verfügung.

4.2.4 Kartenmaterial: Statische und dynamische Entitäten

Das Kartenmaterial setzt sich aus Vektorgrafiken zusammen, die die Fußgänger, Hindernisse und das Fahrzeug repräsentieren. Hierbei werden Fußgänger und das Fahrzeug als Kreise und Hindernisse als Liniensegmente in der 2D Ebene modelliert, was der Umsetzung von PySocialForce entspricht.

Hindernisse haben eine statische Position und können verwendet werden, um zusammengesetzte Entitäten wie Polygone oder aneinandergefügte Liniensegmente zu repräsentieren. Fußgänger und Fahrzeuge sind hingegen dynamische Entitäten und können sich auf der Karte bewegen.

4.2.5 Kollisionserkennung

Bei der Kollisionserkennung wird geprüft, ob das Fahrzeug mit anderen Entitäten wie beispielsweise Fußgängern oder Hindernissen zusammengestoßen ist. Dies entspricht einem Überlappen der geometrischen Formen, die die jeweiligen Entitäten repräsentieren. Im Folgenden werden die dafür benötigten Formeln hergeleitet und bezüglich effizienter Berechenbarkeit optimiert.

Kollisionen zwischen Fahrzeug und Fußgängern

Kollisionen zwischen dem Fahrzeug und Fußgängern können sehr einfach erkannt werden. Als Distanzmetrik dient hierbei die euklidische Distanz zwischen den Kreiszentren $C_{ped} = (x_{ped}, y_{ped})^T$ und $C_{veh} = (x_{veh}, y_{veh})^T$. Ist die Distanz kleiner als die Summe der Kreisradien, schneiden sich die Kreise und es liegt eine Kollision vor, wie in Abbildung 4.3 zu sehen ist. Die Konstellation in Fall 1, bei der sich ein Kreis vollständig innerhalb des anderen Kreises befindet, wird ebenfalls als eine Kollision betrachtet.

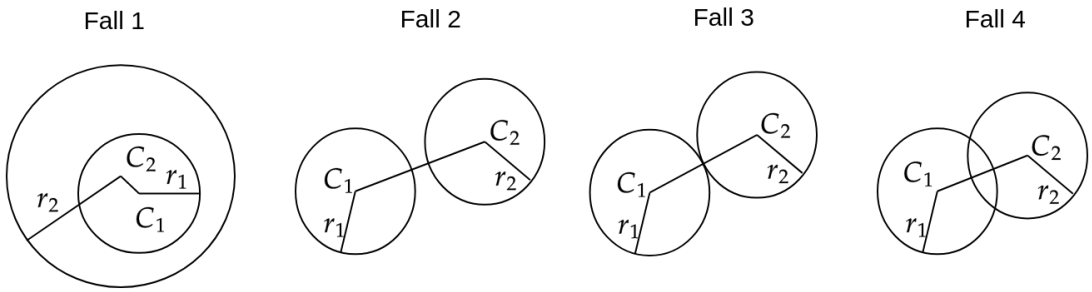

Abbildung 4.3: Fallbetrachtung möglicher Schnittpunkte zwischen zwei Kreisen mit Zentren C_1, C_2 und Radien r_1, r_2. Nur in Fall 2 liegt keine Kollision vor.

$$dist(C_{ped}, C_{veh}) \leq r_{ped} + r_{veh} \tag{4.2}$$

$$dist(C_{ped}, C_{veh}) = \|C_{veh} - C_{ped}\| = \sqrt{(x_{ped} - x_{veh})^2 + (y_{ped} - y_{veh})^2} \tag{4.3}$$

Um die kostspielige Quadratwurzelberechnung zu sparen, wird die Formel auf beiden Seiten der Ungleichung quadriert. Die zusätzlich hinzugefügte Lösung im negativen Wertebereich stellt kein Problem dar, da Distanzen und Kreisradien immer ≥ 0 sind. Somit ergibt sich:

$$dist(C_{ped}, C_{veh})^2 \leq (r_{ped} + r_{veh})^2 \tag{4.4}$$

$$(x_{ped} - x_{veh})^2 + (y_{ped} - y_{veh})^2 \leq (r_{ped} + r_{veh})^2 \tag{4.5}$$

Pro Zeitschritt muss für jeden der n Fußgänger geprüft werden, ob er mit dem Fahrzeug kollidiert, was in linearer Zeit $O(n)$ mit $O(1)$ zusätzlichem Speicher durchgeführt

werden kann. Die damit verbundenen Operationen können zudem sehr effizient und unabhänging voneinander berechnet werden.

Kollisionen zwischen Fahrzeug und statischen Hindernissen

Auch Kollisionen des Fahrzeugs mit Hindernissen müssen berechnet werden. Hierfür wird die Menge aller auf dem Kreisbogen befindlichen Punkte mit der Menge aller auf dem Liniensegment (Hindernis) befindlichen Punkten geschnitten, um alle potentiellen Schnittpunkte zu bestimmen.

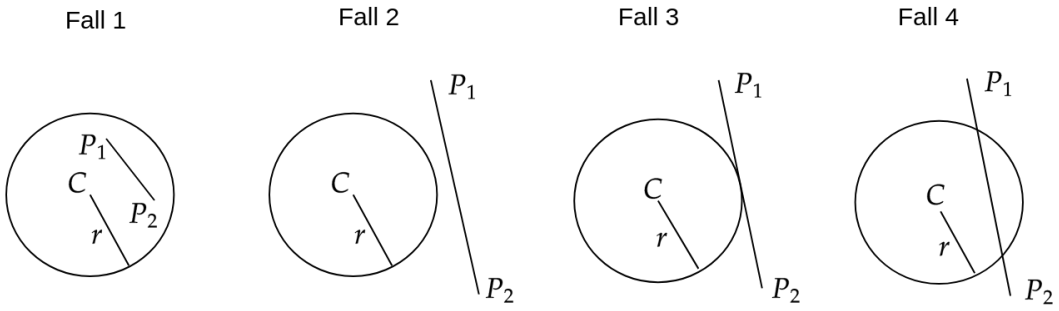

Abbildung 4.4: Fallbetrachtung möglicher Schnittpunkte zwischen einem Liniensegment zwischen P_1, P_2 und einem Kreis mit Zentrum C und Radius r. Nur in Fall 2 keine Kollision vor.

Anhand der Kreisgleichung befindet sich ein Punkt P genau dann auf der Kreislinie, wenn die Distanz zwischen der Kreismitte C und P dem Radius r des Kreises entspricht. Vereinfachend wird die Kreismitte zum Ursprung verschoben, sodass der Abstand zwischen Kreismitte und P genau $\|P\|$ beträgt.

$$M_{circle} = \{P \epsilon R^2 | r = \|P\|\} \tag{4.6}$$

Ein Hindernis ist als Liniensegment zwischen zwei Endpunkten (P_1, P_2) definiert. Die zugehörige Geradengleichung ergibt sich durch einen Punkt auf der Gerade plus ein Vielfaches μ eines Richtungsvektors. In diesem Fall ist die Gerade durch $P_1 - C$ und den Richtungsvektor $P_2 - P_1$ gegeben. Um ausschließlich auf der Gerade befindliche Punkte innerhalb des Liniensegments zu erhalten, wird das Vielfache μ des Richtungsvektors auf $0 \leq \mu \leq 1$ beschränkt.

$$M_{line} = \{\mu(P_2 - P_1) + (P_1 - C) | \mu \epsilon R \wedge 0 \leq \mu \leq 1\} \tag{4.7}$$

Nun kann die Schnittmenge der beiden Punktmengen $M_{line} \cap M_{circle}$ bestimmt werden, um auf Schnittpunkte bzw. Kollisionen zu prüfen. Hierfür wird der von P_1 nach P_2 zeigende Vektor als $s = P_2 - P_1$ und der von C nach P_1 zeigende Vektor als $t = P_1 - C$ notiert.

$$\begin{aligned} r = \|\mu s + t\| &\iff r^2 = \|\mu s + t\|^2 \\ &\iff r^2 = (\mu s + t) \cdot (\mu s + t) \\ &\iff r^2 = \mu^2 s \cdot s + 2\mu s \cdot t + t \cdot t \\ &\iff 0 = \mu^2 s \cdot s + 2\mu s \cdot t + t \cdot t - r^2 \end{aligned} \tag{4.8}$$

$$a = s \cdot s, b = 2s \cdot t, c = t \cdot t - r^2 \tag{4.9}$$

Dieses von μ abhängige Gleichungssystem führt zu einer quadratischen Gleichung mit 0 bis 2 Lösungen, die durch die Lösungsformel für quadratische Gleichungen bestimmt werden können.

$$\mu_{1,2} = \frac{-b \pm \sqrt{D}}{2a}, D = b^2 - 4ac \tag{4.10}$$

Das Gleichungssystem hat in den reellen Zahlen 0 Lösungen, wenn $D < 0$, 1 Lösung, wenn $D = 0$ und 2 Lösungen, wenn $D > 0$. Wie in Abbildung 4.4 dargestellt, liegt eine Kollision vor, sofern $0 \leq \mu \leq 1$. Zudem besteht ein Sonderfall einer Kollision, wenn sich das Liniensegment vollständig innerhalb des Kreises befindet, was gesondert geprüft werden muss. Da jede Prüfung in konstanter Zeit möglich ist, ergibt sich für o Hindernisse eine Zeitkomplexität von $0(o)$; die Speicherkomplexität ist konstant.

4.2.6 Simulation radialer Strahlensensoren (LiDAR)

Ein radialer Strahlensensor sendet eine gewisse Anzahl an Strahlen aus, deren Ausrichtungen gleichmäßig über den Öffnungsbereich des Sensors verteilt sind. Für jeden Strahl muss die Distanz zwischen der Strahlenquelle und der nächstgelegenen, getroffenen Entität innerhalb des Suchradius berechnet werden. Es ergibt sich demnach für r Strahlen, p Fußgänger und o Hindernisse eine Zeitkomplexität von $O(r(p + o))$ und eine Speicherkomplexität von $O(r)$.

Entfernungen zu Fußgängern

Um die Entfernung zwischen einer Strahlenquelle und einem vom Strahl getroffenen Fußgänger zu berechnen, wird der Fußgänger als Kreis und der Strahl als eine von der

Strahlenquelle ausgehende Halbgerade modelliert. Die Richtung des Strahls ist durch den Einheitsvektor v_{ray} gegeben.

Die Formel zur Kollisionsberechnung zwischen einer Linie und einem Kreis wurde bereits für Kollisionen zwischen Fahrzeugen und Hindernissen hergeleitet. Im Gegensatz zum vorherigen Anwendungsfall ist aber nicht nur relevant, ob eine Kollision vorliegt, sondern auch wie weit sie entfernt ist. Hierfür werden ebenfalls $\mu_{1,2}$ bestimmt und anschließend in die Formel P_{Gerade} eingesetzt, um die beiden Schnittpunkte S_1, S_2 zu bestimmen. Falls es keine Schnittpunkte gibt, ist die Entfernung ∞. Bei exakt einem Schnittpunkt ist $S_1 = S_2$, was o.B.d.A. nicht weiter betrachtet werden muss.

Nun kann die kürzeste Entfernung von der Strahlenquelle zu den Schnittpunkten S_1, S_2 als $min(dist(P_{sensor}, S_1), dist(P_{sensor}, S_2))$ bestimmt werden. Allerdings ist hierbei zu beachten, dass der Vektor von der Strahlenquelle zum Schnittpunkt ein positives Vielfaches des Strahlenvektors v_{ray} sein muss, da sich der Strahl nur in diese Richtung ausbreitet. Dies kann durch $\mu \geq 0$ geprüft werden.

$$\begin{aligned} \mu_{1,2} &= \frac{-b \pm \sqrt{D}}{2a}, D = b^2 - 4ac \\ a &= s \cdot s, b = 2s \cdot t, c = t \cdot t - r^2 \end{aligned} \tag{4.11}$$

Um sich im Fall, dass keine Kollision vorliegt, die Quadratwurzelberechnung $\sqrt{D}$ zu sparen, wird nochmals die Lösungsformel für quadratische Gleichungen betrachtet. Da $a = \|s\|^2 \implies 2a > 0$, gibt es genau dann keine Kollision, wenn entweder $D < 0$ oder $-b \pm \sqrt{D} < 0$.

$$-b \pm \sqrt{D} < 0 \iff -b + \sqrt{D} < 0 \iff \sqrt{D} < b \iff b > 0 \wedge b^2 > D \tag{4.12}$$

Somit kann die Prüfung auf $\mu < 0$ durch $b > 0$ und $b^2 > D$ ersetzt werden.

Entfernungen zu Hindernissen

Zur Entfernungsberechnung zwischen einer Strahlenquelle und Hindernissen wird eine neue Berechnungsformel benötigt, die eine Halbgerade (Strahl) mit einen Liniensegment (Hindernis) schneidet. Die Halbgerade ist gegeben durch die Position der Strahlenquelle P_{sensor} und den Richtungsvektor v_{ray} des Strahls als Einheitsvektor; das Liniensegment zwischen P_1 und P_2 ist durch den Punkt P_1 und den Richtungsvektor $v_{seg} = P_2 - P_1$ definiert. Bei der Halbgerade sind nur Vielfache $\tau \geq 0$ des Richtungsvektors erlaubt,

beim Liniensegment nur Vielfache $\mu \in [0, 1]$.

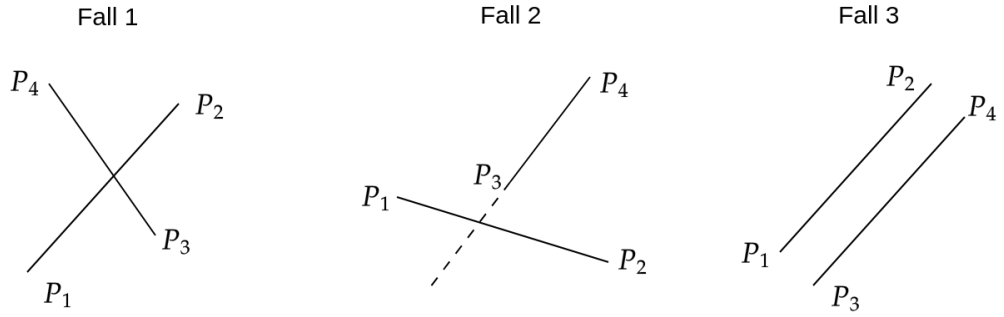

Abbildung 4.5: Fallbetrachtung möglicher Schnittpunkte zwischen zwei Liniensegmenten, jeweils definiert durch ihre Endpunkte P_1, P_2 bzw. P_3, P_4. Parallele Linien wie in Fall 3 schneiden sich nicht. Außerdem sind die Enden der Segmente zu beachten.

Nun können die beiden Punktmengen M_{ray} und $M_{obstacle}$ aufgestellt und deren Schnittmenge $M_{ray} \cap M_{obstacle}$ bestimmt werden, um alle Schnittpunkte zu erhalten.

$$M_{ray} = \{P_{sensor} + \tau v_{ray} | \tau \in R, \tau > 0\} \tag{4.13}$$

$$M_{obstacle} = \{P_1 + \mu v_{seg} | \mu \in R, 0 \leq \mu \leq 1\} \tag{4.14}$$

$$M_{ray} \cap M_{obstacle} = \{P_1 + \mu v_{seg} | \mu \in R, P_{sensor} + \tau v_{ray} = P_1 + \mu v_{seg}\} \tag{4.15}$$

Dies ergibt ein von den Unbekannten τ und μ abhängiges Gleichungssystem, bestehend aus zwei Gleichungen bezüglich der linear unabhängigen x/y Koordinaten der Vektoren. Das Gleichungssystem ist also eindeutig lösbar.

$$\begin{aligned} &P_{sensor} = (x_{sensor}, y_{sensor})^T, P_1 = (x_1, y_1)^T, \\ &v_{ray} = (x_{ray}, y_{ray})^T, v_{seg} = (x_{seg}, y_{seg})^T \\ &v_{diff} = (x_{diff}, y_{diff})^T = P_1 - P_{sensor} \end{aligned} \tag{4.16}$$

$$\begin{aligned} I)\, x_{sensor} + \tau x_{ray} &= x_1 + \mu x_{seg} \\ II)\, y_{sensor} + \tau y_{ray} &= y_1 + \mu y_{seg} \end{aligned} \tag{4.17}$$

Durch Umformen ergeben sich folgende nach τ aufgelöste Gleichungen:

$$\begin{aligned} I')\,\tau &= \frac{\mu x_{seg} + x_{diff}}{x_{ray}} \\ II')\,\tau &= \frac{\mu y_{seg} + y_{diff}}{y_{ray}} \end{aligned} \tag{4.18}$$

Nun wird τ durch Gleichsetzen eliminiert. Anschließend wird nach μ aufgelöst, um die Lösung des Gleichungssystems zu bestimmen.

$$\frac{\mu x_{seg} + x_{diff}}{x_{ray}} = \frac{\mu y_{seg} + y_{diff}}{y_{ray}} \tag{4.19}$$

$$\mu \frac{x_{seg}}{x_{ray}} + \frac{x_{diff}}{x_{ray}} = \mu \frac{y_{seg}}{y_{ray}} + \frac{y_{diff}}{y_{ray}} \tag{4.20}$$

$$\mu \frac{x_{seg}}{x_{ray}} - \mu \frac{y_{seg}}{y_{ray}} = \frac{y_{diff}}{y_{ray}} - \frac{x_{diff}}{x_{ray}} \tag{4.21}$$

$$\mu \left(\frac{x_{seg} y_{ray}}{x_{ray} y_{ray}} - \frac{x_{ray} y_{seg}}{x_{ray} y_{ray}} \right) = \frac{x_{ray} y_{diff}}{x_{ray} y_{ray}} - \frac{y_{ray} x_{diff}}{x_{ray} y_{ray}} \tag{4.22}$$

$$\mu \frac{x_{seg} y_{ray} - x_{ray} y_{seg}}{x_{ray} y_{ray}} = \frac{x_{ray} y_{diff} - y_{ray} x_{diff}}{x_{ray} y_{ray}} \tag{4.23}$$

$$\mu (x_{seg} y_{ray} - x_{ray} y_{seg}) = x_{ray} y_{diff} - y_{ray} x_{diff} \tag{4.24}$$

$$\mu = \frac{x_{ray} y_{diff} - y_{ray} x_{diff}}{x_{seg} y_{ray} - x_{ray} y_{seg}}, x_{seg} y_{ray} - x_{ray} y_{seg} \neq 0 \tag{4.25}$$

Wenn $x_{seg} y_{ray} - x_{ray} y_{seg} \neq 0$, ergibt sich der Schittpunkt $S = P_1 + \mu v_{seg}$ durch Einsetzen von μ in $M_{obstacle}$. τ kann über $\tau = \frac{\mu x_{seg} + x_{diff}}{x_{ray}}$ oder $\tau = \frac{\mu y_{seg} + y_{diff}}{y_{ray}}$ bestimmt werden. Nun muss noch überprüft werden, ob $0 \leq \mu \leq 1$ und $\tau \geq 0$. Ist dies der Fall, trifft der Strahl das Hindernis nach einer Entfernung von $dist(P_{sensor}, S)$, andernfalls ist die Distanz ∞. Wenn $x_{seg} y_{ray} - x_{ray} y_{seg} = 0$, steht der Strahl parallel zum Hindernis und verfehlt, da Hindernisse keine Tiefe besitzen.

Nachbereitung der berechneten Entfernungen

Nachdem die Kollisionsberechnungen für alle Strahlen und Entitäten durchgeführt wurden, kann die minimale Entfernung d_{ray} pro Strahl bestimmt werden. Entfernungen, die noch auf ∞ initialisiert sind, da keine Kollision vorliegt, werden mit $d_{ray} = min(d_{ray}, s_{max})$ auf die maximale Scanreichweite des Sensors beschränkt.

Nun wird zur bisher perfekten Messung ein leichtes Rauschen hinzugefügt, was vor allem für das Training einer Künstlichen Intelligenz wichtig ist, um Overfitting vorzubeugen. Hierbei gehen Strahlen mit einer gewissen Wahrscheinlichkeit P_{lost} ganz verloren, sodass die gemessene Distanz der maximalen Scanreichweite entspricht. Zudem können auch Messungen mit der Wahrscheinlichkeit $P_{corrupt}$ verfälscht werden, sodass die Entfernung zufällig um einen gleichverteilten Faktor $f \in [0, 1]$ skaliert wird. Empfohlene Wahrscheinlichkeiten sind $P_{lost} = 0.005$ und $P_{corrupt} = 0.002$.

4.2.7 Steuerung der Fußgänger

Bezüglich der Simulation von realistischem Fußgängerverhalten, wird das Social Force Modell aus Abschnitt 2.2 verwendet, um Fußgänger einzeln oder in Gruppen zu einem Zielort zu bewegen. Eine Gruppe ist am Ziel, wenn sich ihr Massenschwerpunkt $\frac{1}{n}\sum_{i=1}^{n} p_i$ zu einem diskreten Zeitpunkt t nah genug am Zielpunkt befindet. O.B.d.A. sind einzelne Fußgänger als eine Gruppe mit nur einem Fußgänger definiert. Da Fußgängerzonen und Gehwege simuliert werden sollen, wird im Folgenden betrachtet, wie ein entsprechendes Verhalten erzielbar ist.

Zur Simulation von Fußgängerzonen werden zunächst Zonen definiert, in denen sich die Fußgänger aufhalten. Um ein Verhalten zu erzielen, bei dem die Fußgänger durcheinander laufen, werden gleichverteilt zufällige Zielpunkte innerhalb der Zone gewählt, zu denen die Fußgänger laufen. Sobald ein Ziel erreicht ist, wird ein neues, zufälliges Ziel bestimmt. Das stationäre Verhalten in Fußgängerzonen wird durch das zielgerichtete Laufen von Routen ergänzt, was eher dem Fußgängerverhalten auf Gehwegen entspricht. Ähnlich wie bei der Globalen Navigation des Fahrzeugs werden die Routen zwischen Start- und Zielzone anhand von groben Wegpunkten vorgegeben. Sobald ein Wegpunkt erreicht ist, wird der nächste angesteuert. Sind die Fußgänger am finalen Ziel innerhalb der Zielzone, erscheinen sie wieder in der Startzone, um die Route erneut zu laufen.

Entsprechend der vorgegebenen Fußgängerdichte, d.h. Fußgänger pro Fläche, werden allen Routen und Fußgängerzonen eine Anzahl an Fußgängern gemäß ihrer Fläche zugewiesen. Die Gehwegfläche wird durch die Multiplikation der jeweiligen Routenlänge mit einer angenommenen Gehwegbreite von 3-4 Metern geschätzt. Wie in Moussaïd et. al [7] beschrieben ist die Größe der Gruppen poissonverteilt. Daher wird vor der Zuteilung der Fußgänger zunächst die Anzahl der Gruppenmitglieder gemäß der Wahr-

scheinlichkeitsverteilung gezogen. Anschließend werden die Massenschwerpunkte der Gruppen gleichverteilt innerhalb der Fußgängerzonen bzw. entlang der Routen bestimmt und deren Gruppenmitglieder normalverteilt um den Schwerpunkt positioniert.

4.3 Technische Umsetzung der Simulationsumgebung

Zum Erlernen autonomer Fahrverhaltensweisen wird ein Simulator anhand der Beschreibung aus Abschnitt 4.2 in Python umgesetzt. Das sehr ähnliche Projekt *RobotSF* der Universität Triest von Caruso et. al [1] wird hierfür als Basis verwendet und entsprechend adaptiert. In der ursprünglichen Version von *RobotSF* steht sowohl ein steuerbares Fahrzeug mit entsprechender Sensorik und Aktuatorik, als auch die Simulation der Fußgänger mittels Social Force durch das Paket *PySocialForce* [30] bereit. Die Implementierung des Simulators bezüglich der Differential Drive Kinematik, des LiDAR-Sensors, der Fußgängersteuerung und der Kollisionslogik ist von Caruso et. al und wird entsprechend der Konzeption aus Abschnitt 4.2 angepasst. Die Schnittstelle der Simulationsumgebung zur Umsetzung der Trainingsalgorithmen von Caruso et. al wird zugunsten einer breiteren Kompatibilität mit Implementierungen gängiger Trainingsalgorithmen wie beispielsweise *Stable Baselines 3* [33] zu einer *OpenAI Gym* Schnittstelle [34] umgestellt. Da es sich bei der Umsetzung der Simulationsumgebung um eine Erweiterung handelt, wird diese im Folgenden *RobotSF* genannt.

4.3.1 Visualisierung der Simulationsumgebung

Da die ursprüngliche Simulationsumgebung über keine Live-Visualisierung verfügt, wird eine entsprechende Komponente zu *RobotSF* hinzugefügt, wie in Abbildung 4.6 zu sehen ist. Die Ansicht verwendet die Visualisierungstechnologie *PyGame*, womit eine ausreichende Performanz für die flüssige Darstellung von Bewegungen erzielt werden kann. Vergleichbare Technologien wie *Tkinter* oder *Turtle* sind zu langsam und kommen daher nicht in Betracht.

Die dargestellten Entitäten umfassen das Fahrzeug (blauer Kreis), Fußgänger (rote Kreise) und Hindernisse (graue Polygone). Zudem werden die Zielzonen der anzusteuernden Wegpunkte (grüne Kreise) und der Dynamikvektor des Fahrzeugs (blaue Linie) augmentiert. Die Kamera blickt senkrecht von oben auf die simulierte 2D-Ebene und zentriert das betrachtete Fahrzeug in der Mitte des Bildschirms. Zudem kann auch der Zoom-Faktor zu Beginn der Simulation passend eingestellt werden.

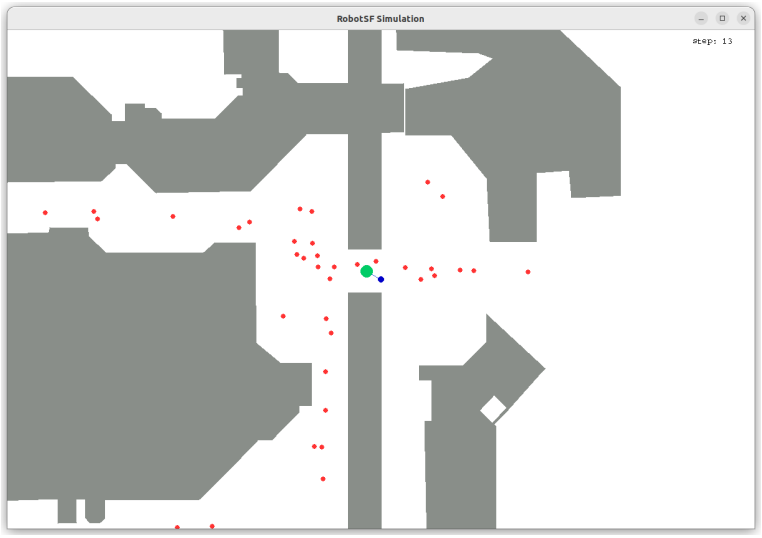

Abbildung 4.6: Simulatoranzeige mit PyGame

4.3.2 Aufbereitung des Kartenmaterials

Zur Bearbeitung des Kartenmaterials wird das Werkzeug *Map Editor* entwickelt. Wie in Abbildung 4.7 zu sehen ist, besteht der *Map Editor* aus einem Texteditor auf der linke Seite und einer Live-Vorschau der Karte auf der rechte Seite, sodass mit vertretbarem Aufwand neue Trainingsszenarien erstellt werden können. Durch ein weiteres Skript können Gebäudeumrisse maßstabsgetreu aus OpenStreetMap geladen werden, was das Importieren von echtem Kartenmaterial auf einfache Weise ermöglicht. Per Linksklick in die Vorschauanzeige werden die Koordinaten an der entsprechenden Stelle auf der Karte ausgegeben, sodass sich Zonen und Routen leicht einpflegen lassen.

Für den Map Editor wird Tkinter als Visualisierungstechnologie gewählt, da PyGame über keine vorgefertigten Bausteine zur Texteingabe verfügt und somit ausscheidet. Zudem erfordert die Vorschau des Kartenmaterials ohnehin keine hohen Bildraten für eine flüssige Darstellung, sodass stattdessen auf die Canvas-Funktion von Tkinter zurückgegriffen wird. Ein vollkommen visuelles Bedienkonzept ohne die Notwendigkeit einer manuellen Bearbeitung des Kartenmaterials im Texteditor stellt eine Alternative dar, wird jedoch aufgrund des niedrigen Mehrwerts bei gleichzeitigem, erheblichem Mehraufwand für das Projekt verworfen. Es wäre vorgesehen gewesen, die einzelnen Zonen, Wegpunkte und Hindernisse per Mausklick einzufügen. Per Schaltfläche wählt der Bearbeiter die einzufügende Struktur aus. Anschließend setzt er die Umrisse per

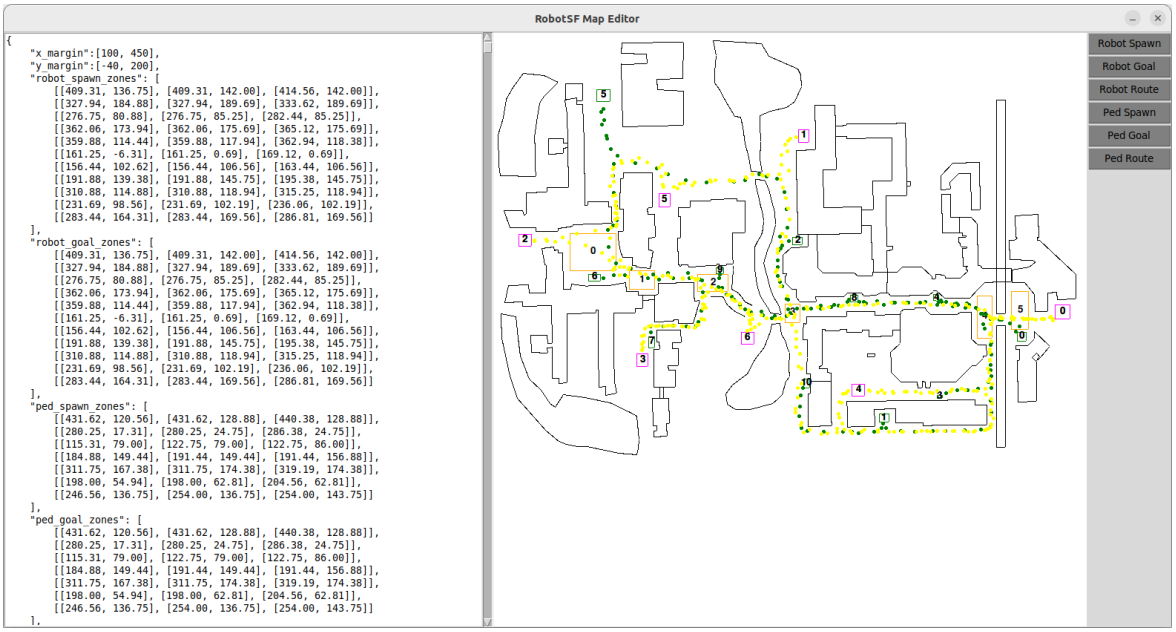

Abbildung 4.7: Map Editor mit Texteingabe auf der linken und Vorschau auf der rechten Seite. Die Vorschau zeigt den Campus der Universität Augsburg. Im Text Editor wird das Kartenmaterial bearbeitet.

Linksklick in die Kartenvorschau. Ein Rechtsklick bricht das Einfügen ab. Ist keine Schaltfläche aktiv, werden die Entitäten beim Bewegen des Mauszeigers über ihre unmittelbare Umgebung hervorgehoben. Wird während der Hervorhebung zusätzlich ein Recktsklick ausgeführt, löscht dies die Entität. Das Setzen der Kartenumrisse löscht immer die vorherigen Umrisse. Per Drehung des Mausrads kann wie gewöhnlich gezoomt werden.

4.3.3 Konzeption der Trainingsumgebung

Der *RobotSF* Simulator des Forscherteams aus Triest sieht bereits vor, zufallsgeneriertes Kartenmaterial mit Hindernissen (Polygone) zu erzeugen und als JSON-Dateien zu speichern. Wie in Abbildung 4.8 zu sehen ist, besteht das Kartenmaterial aus unregelmäßigen, sternförmigen Hindernissen, was mehr einer Mondlandschaft anstatt einem Gehweg oder einer Fußgänerzone entspricht. Die unrealistischen Formen von *RobotSF* werden daher durch echtes, aus OpenStreetMap importiertes Kartenmaterial ersetzt und mithilfe des *Map Editors* aufbereitet. Dies erfordert eine Effizienzoptimierung der Simulationsumgebung, auf die im darauffolgenden Abschnitt 4.4 eingegangen wird.
Als Kartenmaterial für die Trainingsumgebungen werden zwei echte Karten gewählt.

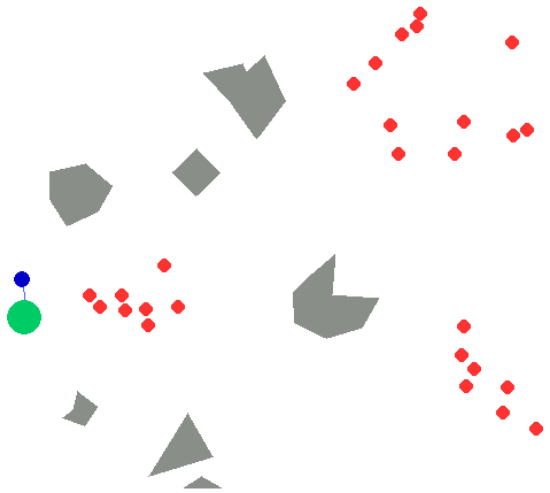

Abbildung 4.8: Beispiel für altes Kartenmaterial des *RobotSF* Simulators

Wie in Abbildung 4.9 zu sehen ist, besteht die erste Trainingsumgebung aus einigen miteinander verbundenen Häuserblocks nahe des Universitätsgeländes, in deren Innenhöfen künstliche, in orange dargestellte Fußgängerzonen angelegt werden, um belebte Plätze zu simulieren. Das Fahrzeug startet in der blau markierten Zone und folgt den durch die grünen Punkte dargestellten Routen bis hin zur jeweiligen Zielzone. Die Routen verlaufen in alle Himmelsrichtungen und führen bewusst durch mindestens eine Menschenmenge.

Dadurch dass der Agent gezwungen wird, durch Menschenmengen zu fahren, soll die sichere Interaktion mit Fußgängern erlernt werden. Die Verkehrsdichte in den orange markierten Fußgängerzonen kann während der Trainingsläufe und Evaluationen variiert werden, um die Robustheit der erlernten Fahrverhaltensweise zu prüfen. Durch das Ein- und Ausschalten der *Ped-Robot Force* wird das Verhalten der Fußgänger dahingehend modifiziert, ob sie das Fahrzeug wahrnehmen können und ihm ausweichen. Durch die Modifikation des Fußgängerverhaltens bezüglich der Wahrnehmung des Fahrzeugs während des Trainings soll gesteuert werden, ob der Agent offensiveres oder defensiveres Fahrverhalten lernt.

Für die zweite Trainingsumgebung wird eine Karte des Campus der Universität Augsburg gewählt, da sich entsprechendes Kartenmaterial als verkehrsberuhigter Bereich mit viel Fußgängerverkehr bestens für die virtuelle Erprobung autonomer Mikromobilitätsahrzeuge anbietet und auch perspektivisch die Evaluationen echter Fahrzeuge in

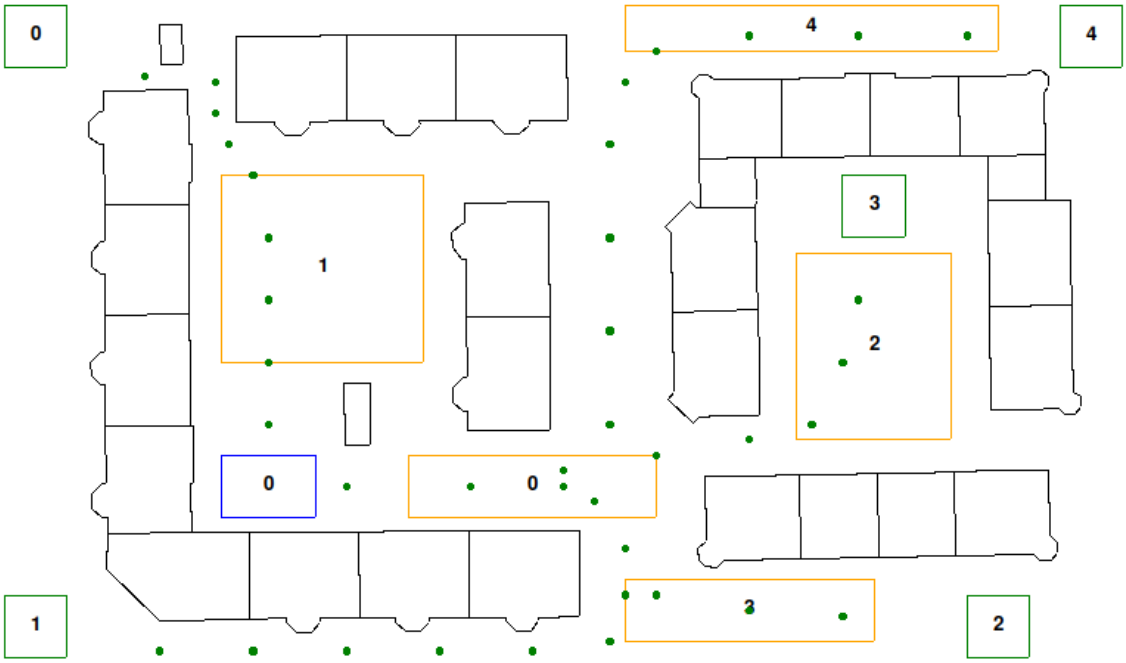

Abbildung 4.9: Erste Trainingsumgebung des überarbeiteten *RobotSF* Simulators

der Realität ermöglicht. Wie in Abbildung 4.10 zu sehen, umfassen die in grün dargestellten Start- und Zielorte des Fahrzeugs das Unikum (0), das Rote Pferd (1), die Alte Cafeteria (2), den Durchgang zwischen N-Gebäude und Mensa (3), das Prüfungsamt (4), den Ausgang zum Messeparkplatz (5), die Juristische Fakultät (6), den Eingang zur Wirtschaftsinformatik (7), den Eingang zum C-Hörsaal (8), den Eingang zur Zentralbibliothek (9) und den Eingang zur mathematisch-naturwissenschaftlichen Bibliothek (10). Am Wasserspiel (0, 1), vor der Zentralbibliothek (2), vor der Brücke (3) und bei den Straßenbahnhaltestellen (4, 5) befinden sich die orange markierten Fußgängerzonen, um dichten Verkehr zu simulieren. Zudem verlaufen auf allen befahrenen Gehwegen die mit gelben Punkten dargestellten Fußgängerrouten in beide Laufrichtungen, um typischen Verkehr zu generieren.

Das Design der Routen sieht vor, verschiedenste Situationen herbeizuführen, die auch in der Realität auftreten können. Auf den Gehwegen wird hauptsächlich getestet, ob der Agent die Bewegung der Fußgänger korrekt einschätzen und Überholmanöver in engen Räumen sicher durchführen kann. Nah an Hauswänden verlaufende 90-Grad Kurven prüfen, ob vor der Kurve abgebremst wird, da der Agent nur eine eingeschränkte Sicht auf den sich hinter der Wand befindlichen Gegenverkehr hat. Hingegen wird auf großen, belebten Plätzen wie bereits bei der ersten Trainingsumgebung die Fähigkeit

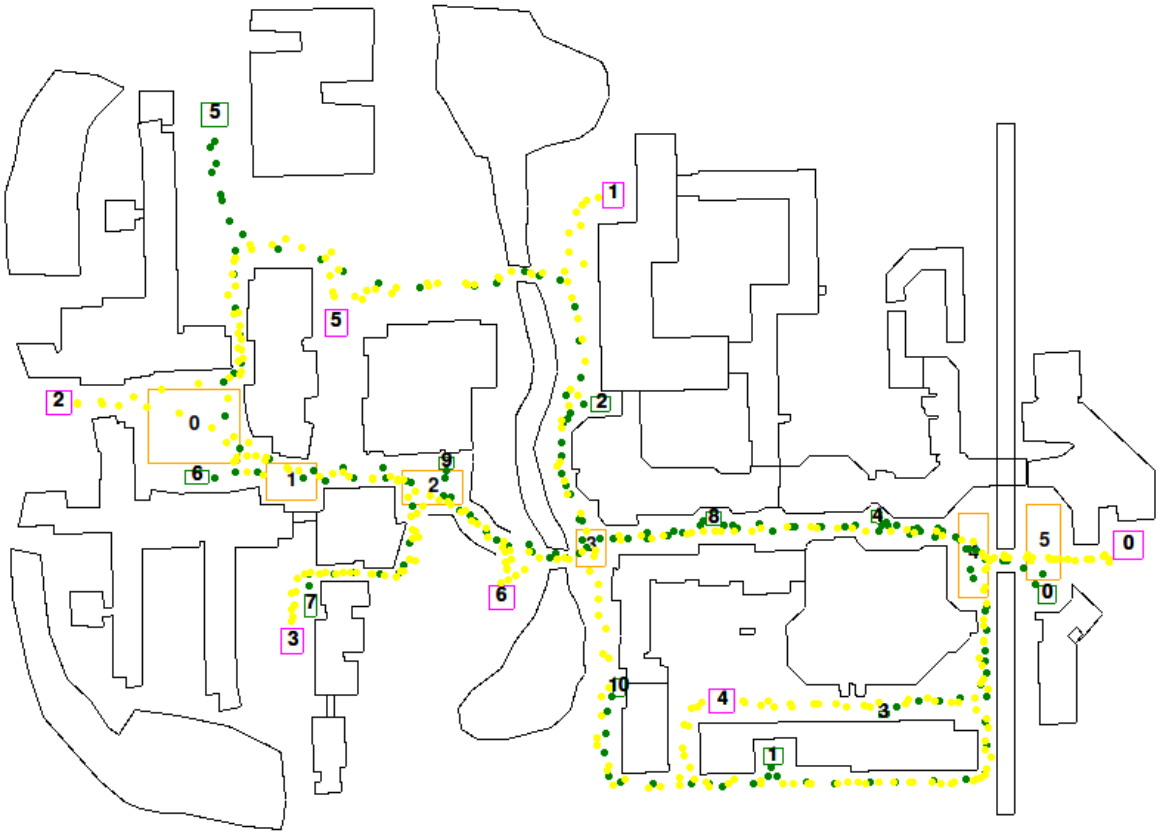

Abbildung 4.10: Zweite Trainingsumgebung des überarbeiteten *RobotSF* Simulators. Das Kartenmaterial zeigt den Campus der Universität Augsburg.

zur Kollisionsvermeidung und Erzielung von Fortschritten in dichten Menschenmengen getestet. Dabei soll der Agent ein möglichst defensives Verhalten aufweisen, damit sich die Fußgänger sicher fühlen können. Durch Engstellen soll der Agent lernen, erst zu fahren, sobald eine befahrbare Passage frei wird. Dies erfordert unter anderem eine Einschätzung, ob die Engstelle lange genug befahrbar bleibt, um sie zu passieren.

4.4 Effizienzoptimierung der Simulationsumgebung

Nach einem Austausch mit dem Forscherteam hinter *RobotSF* wird klar, dass ein effizientes Training von Fahragenten an erheblichen Effizienzproblemen der Simulationsumgebung leidet. Die Forscher geben Trainingszeiten von bis zu einem Monat an. Aufgrund des beschränkten zeitlichen Rahmens dieser Arbeit sind entsprechende Wartezeiten nicht hinnehmbar, weshalb die Umsetzung durch effizientere Algorithmen und Datenstrukturen optimiert wird. Zur Messung der Effizienz dient ein Skript, das 10000 Simulationsschritte ausführt und dabei zufällige Aktionen auswählt. Mithilfe

Tabelle 4.1: Performance-Profile der jeweiligen Simulatorversionen. Es sind die Gesamtrechenzeit in Sekunden und die jeweiligen Anteile der Komponenten an der Gesamtzeit in Prozent zu sehen. Die Spalte „Speedup“ zeigt den jeweiligen Faktor der erzielten Beschleunigung zur vorherigen Simulatorversion.

Performance-Profil vs. Simulatorversion	Total	PySF	Koll.	LiDAR	Sonst.	Speedup
Unoptimized	289s	36%	34%	19%	11%	-
Fast Grid Raycast	132s	76%	13%	5%	6%	2.26x
Fast Grid Map	82s	78%	2%	9%	11%	1.61x
Fast Obstacle Force	61s	74%	3%	12%	11%	1.34x
Continuous Map Algos	59s	72%	10%	6%	12%	1.03x
No Custom Forces	15s	30%	5%	15%	50%	3.93x

von Scalene [35] kann der Beitrag jeder Zeile aus dem Python Quellcode gemessen werden, um kritische Komponenten zu identifizieren. Anhand der Performance-Profile ist ersichtlich, dass hauptsächlich der Kraftberechnung von PySocialForce und die von Caruso et. al umgesetzten Komponenten des LiDAR-Sensors und der Kollisionserkennung zur Rechenzeit beitragen. Alle restlichen Zeiten werden unter dem Punkt „Sonstige“ zusammengefasst, wie in Tabelle 4.1 dargestellt.

Anhand des ersten Profils entfallen ca. 48% auf eine Datei namens „map.py“, die sowohl die Logik zur Kollisionserkennung, als auch die Umsetzung des LiDAR-Sensors enthält. Daher wird die Repräsentation des Kartenmaterials nun genauer betrachtet. Es stellt sich heraus, dass die Karte durch eine Rasterstruktur (2D-Bitmaske) umgesetzt wird, wobei ein Bit pro Zelle angibt, ob sich dort Hindernisse befinden. Zum Aufbau des Rasters werden die Umrisse aller Entitäten berechnet und anschließend per bitweisem OR zu einer Maske (engl. Occupancy Grid) zusammengefügt. Umrisse statischer Entitäten werden zu Beginn der Simulation vorberechet und um die dynamisch berechneten Umrisse aller beweglichen Entitäten zu jedem Simulationsschritt ergänzt.

Vorteile der Datenstruktur bestehen zunächst darin, dass die Komplexität der Algorithmen, die auf einem fertig berechneten Raster arbeiten, lediglich mit der Rastergröße skaliert, nicht aber mit der Anzahl der dargestellten Objekte. Die Kollisionsberechnung des Fahrzeugs kann beispielsweise mittels bitweisem AND der Fahrzeugumrisse mit den Hindernissen auf der Karte umgesetzt werden. Der Vorteil der Skalierbarkeit ist jedoch ein Trugschluss, da die Anzahl an simulierten Objekten für einen tatsächlichen Vorteil größer als die Anzahl der Rasterzellen sein muss. Weitere Nachteile ergeben sich, da

das Sichtfeld des Fahrzeugs mit $O(r^2d^2)$ bzgl. des Sichtradius r und der Rasterdichte d skaliert und somit dem „Curse of Dimensionality“ unterliegt, der sich zudem verstärkt, falls eine feingranularere Rasterisierung der Karte benötigt wird. Für *RobotSF* wird beispielsweise ein quadratisches Raster mit einer Seitenlänge von 40 Metern gewählt, dessen Zellen eine Seitenlänge von 0.1 Meter aufwiesen. Dies entspricht einem Vielfachen von 160000 Rechenzyklen pro Rasteroperation. Abgesehen von einem erheblichen Verlust an Genauigkeit sind Algorithmen mit Bitmasken auch sehr ineffizient, da die restlichen Programmteile von *RobotSF* ohnehin mit Vektorgrafiken arbeiten und daher zu jedem Zeitschritt die Konvertierung aller beweglichen Objekte in Bitmasken erfordern. Auch die Umsetzung eines LiDAR-Sensors bzgl. einer Rasterstruktur ist eher umständlich, da die Strahlen als Pixelgeraden, z.B. mit Bresenham's Line [36], berechnet werden müssen, um dann einen Abgleich der Bitmasken durchführen zu können. Um die Kompatibilität zur alten Grid-Struktur von Caruso et. al beibehalten zu können, wird zunächst der LiDAR-Sensor und anschließend die Kollisionserkennung mit Numba optimiert, indem die Algorithmen dahingehend verbessert werden, möglichst viele Befehle in generierten C Code auszulagern. Dies führt zu ersten Erfolgen, sodass die Rechenzeiten von vormals 158 Sekunden auf 9 Sekunden gesenkt werden können, was im Bezug auf die Gesamtrechenzeit eine 3.6-fache Beschleunigung erlaubt.

Nun stellen die Kraftberechnungen von PySocialForce mit 78% den Hauptanteil am Profil, sodass diese im Folgenden näher betrachtet werden. Um den kritischen Pfad von PySocialForce im Performance-Profil zu analysieren, wird dessen Quelltext von GitHub als Untermodul registriert. Untersuchungen ergeben, dass 99% der Rechenzeit für die Abstoßungskraft der Fußgänger von Hindernissen aufgewendet werden, damit Fußgänger nicht durch diese hindurch laufen können. Nach genauerer Betrachtung des Codes ergibt sich, wie in Abbildung 4.11 dargestellt, dass PySocialForce Hindernisse nicht als Liniensegmente, sondern als einzelne, zwischen den beiden Endpunkten im Abstand von 0.1 Meter angeordnete Punkte betrachtet, die zum Simulationsstart vorberechnet werden. Die abstoßende Kraft wirkt um jeden einzelnen dieser Punkte, wobei die Kraft exponentiell mit wachsender Entfernung abnimmt. Die Repräsentation von Liniensegmenten durch viele Punkte ist bereits für kleine Karten sehr ineffizient, da der Algorithmus nicht nur mit der Anzahl der Hindernisse, sondern zusätzlich mit der Länge der Hindernisse skaliert. Daher wird die Kraftberechnung durch ein virtuelles Potentialfeld ersetzt, für das die reziproge, quadratische Entfernung zwischen Fußgängern und Hindernissen aus Abschnitt 2.2 herangezogen wird. Die Formel für die Abstoßungskraft $F = -\frac{\partial}{\partial p} dist(o,p)^{-2}$ entspricht den partiellen Ableitungen nach der

x- und y-Koordinate der Fußgängerposition p. Zur Berechnung der Distanz zwischen Fußgänger und Hindernis kann ein durch p verlaufendes Lot auf das Liniensegment gefällt werden. Trifft das Lot das Liniensegment nicht, wird stattdessen die Distanz zum nähergelegenen Endpunkt des Segments herangezogen. Durch diese einfache Änderung der Berechnungsformel und einige weitere Hardware-Optimierungen mittels Numba kann die Effizienz des Simulators um Faktor 1.3 gesteigert werden. Des weiteren fällt auf, dass für einige Kräfte zur Umsetzung von Gruppendynamiken eigene Implementierungen von Caruso et. al existieren, die einen sehr großen Anteil am Performance-Profil einnehmen. Durch das Ersetzen der Kräfte durch ihr standardmäßig in PySocialForce mitgeliefertes Pendant kann die Simulationszeit nochmals 4-fach beschleunigt werden.

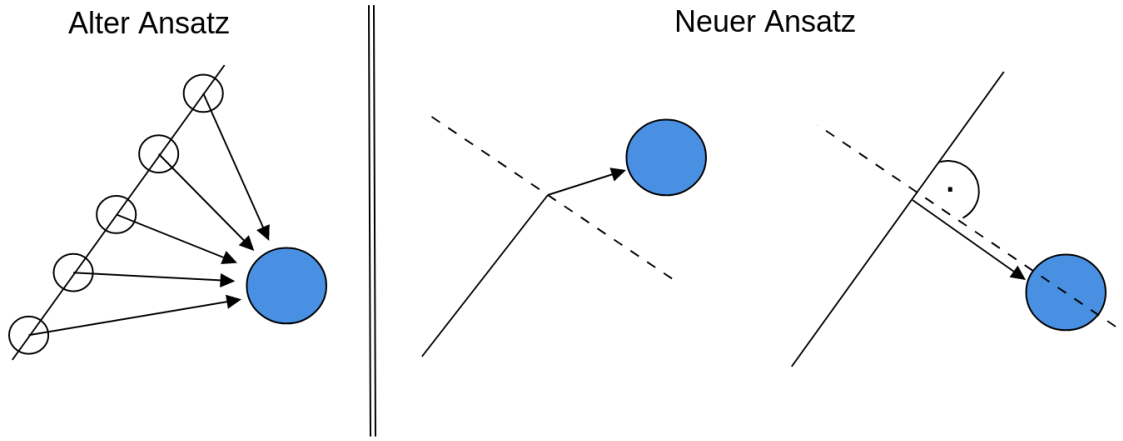

Abbildung 4.11: Umsetzung der *Obstacle Force*. Der alte Ansatz repräsentiert Hindernisse als viele Punkte, von denen der Fußgänger jeweils einzeln abgestoßen wird. Der neue Ansatz fällt ein Lot auf das Hindernis und stößt den Fußgänger orthogonal ab. Befindet sich der Fußgänger neben dem Hindernis, wird die Distanz zum nähergelegenen Endpunkt des Hindernisses herangezogen.

Wie bereits angesprochen, ist die Umsetzung der Grid-Struktur sehr unflexibel und verhindert einfachere Algorithmen mit Vektorgrafiken. Außerdem kann der LiDAR-Sensor aufgrund der Rundungseffekte durch die Rasterisierung nicht hinreichend getestet werden, um die Funktionstüchtigkeit der Sensorik sicherzustellen. Aus besagten Gründen wird die Repräsentation von Hindernissen, Fußgängern, Fahrzeugen und Laserstrahlen auf einfache Vektorgrafiken umgestellt, wie in Abschnitt 4.2 beschrieben. Dies hat zwar keine nennenswerten Auswirkungen auf das Performance-Profil, aber ermöglicht eine Umstrukturierung und Vereinfachung der Simulationslogik. Da *RobotSF* nun im

Vergleich zur ursprünglichen Simulationsumgebung von Caruso et. al ungefähr 19-fach beschleunigt ist, können die Trainingsexperimente mit einer vertretretbaren Wartezeit durchgeführt werden. Es werden noch einige kleinere Verbesserungen eingeführt, um die Berechnung von Kollisionen und Kräften weit voneinander entfernter Entitäten zu filtern. Dies soll in diesem Rahmen jedoch nicht weiter thematisiert werden.

4.5 Mögliche Verbesserungen bezüglich Effizienz

Wie der *Mulit-Robot* Ansatz von Fan et. al [4] demonstriert, können gleichzeitig mehrere Fahrzeuge innerhalb derselben Simulationsumgebung trainieren, sodass die Umgebung vektorisiert wird. Im Fall von *RobotSF* würde dies zudem eine Effizienzsteigerung bezüglich der Rechenzeit pro Simulationsschritt ermöglichen, da die Berechnung der Kinematik und Sensorik pro Fahrzeug nur etwa 12% Anteil am gesamten Performance-Profil hat. Je nach Kartenmaterial kann in jeder Startzone je ein Fahrzeug seine Route starten. Beispielsweise können Fahrzeuge auf dem Universitätscampus an 11 verschiedenen Orten starten, was die während des Trainings verwendeten 64 Simulationsumgebungen auf 6 reduziert. Im konkreten Fall ist in etwa eine 5-fache Effizienzsteigerung möglich.

$$\eta = \left(\frac{0.88 \cdot 6 + 0.12 \cdot 64}{64} \right)^{-1} \approx 5$$

Durch eine Umstellung auf lokalitätsaffine Algorithmen im Zusammenspiel mit passenden Datenstrukturen, z.B. Quad-Trees, wie es in der Grafikprogrammierung üblich ist, sind weitere Effizienzsteigerungen bei der Umsetzung der Simulationsumgebung möglich. Insbesondere kann die Dimensionierung des simulierten Kartenmaterials aufgrund der lokalitätsaffinen Algorithmen nochmals deutlich vergrößert werden, sodass mehr Startplätze und folglich auch weitere Fahrzeuge pro Umgebung denkbar sind. Eine entsprechende Anpassung der Simulationsumgebung geht über den Umfang dieser Arbeit deutlich hinaus, kann aber in Anbetracht der möglichen Verbesserungen durchaus lohnenswert sein, zumal die Ergebnisse von Fan et. al [4] ohnehin eine Qualitätssteigerung der erlernten Fahrverhaltensweisen in Aussicht stellen.

Auch die Umsetzung des Trainingsalgorithmus mit Stable Baselines 3 hat noch Verbesserungspotential. Während eines Trainingslaufs mit PPO werden ca. 30 Gigabyte Arbeitsspeicher allokiert. Anhand der 64 parallel betriebenen Simulationen und einem für das Sammeln der Trainingsdaten verwendeten Zeitintervall von 2048 Simulations-

schritten wären bei optimaler Speichernutzung aber nur ca. 450 Megabyte notwendig. Eine eigene Implementierung von PPO mit TensorFlow 2 kann erfolgreich die Spiele Pong und CartPole erlernen, ist jedoch nicht nennenswert effizienter. Um den Speicher adäquat kontrollieren zu können, wäre eine Portierung hin zu einer hardwarenahen Sprache sinnvoll. Um den Großteil der Simulationslogik behalten zu können, wird eine graduelle Umstellung auf Mojo empfohlen. Neben der Unterstützung von eigenem Python Code sind auch alle verwendeten Abhängigkeiten weiterhin verfügbar. Zusätzlich profitieren in Mojo geschriebene Programme von einer Hyperparameteroptimierung der Cachegrößen. Hierzu kompiliert Mojo zur Laufzeit mehrere Versionen des Codes und unterzieht diese anschließend einigen Benchmarks, sodass eine optimale Performance mit der verwendeten Maschine ohne tiefe Kenntnisse der Hardwareprogrammierung erzielt werden kann.

5 Lernexperimente und Ergebnisse

Dieses Kapitel beschäftigt sich mit dem Erlernen autonomer Fahrverhaltensweisen. Zunächst werden die durchgeführten Trainingsläufe vorgestellt. Daraufhin folgt eine Evaluation der daraus resultierenden Agenten anhand von Unfallmetriken. Anschließend werden die Ergebnisse interpretiert und mit den Ergebnissen anderer Ansätze verglichen. Zum Schluss wird noch auf die Minimierung der Trainingszeiten durch eine Verbesserung der Sample Efficiency anhand einer Hyperparameteroptimierung und der Anwendung von Modellbasiertem Lernen eingegangen.

5.1 Durchführung der Trainingsläufe

Im Laufe dieser wissenschaftlichen Arbeit werden eine Vielzahl an Trainingsläufen durchgeführt. Die erfolgreichsten Agenten sind in Tabelle 5.1 zu sehen. Außerdem stehen die Modelle und Videos des beobachteten Fahrverhaltens im Appendix zur Verfügung. Im Folgenden werden die Versuche beschrieben und die vorgenommenen Adaptionen an den Trainingsbedingungen begründet. Experiment 01 bezieht sich auf den Ansatz der ursprünglichen Umsetzung von *RobotSF* wie in [1] und dient als Grundlinie. Alle weiteren Experimente verwenden die Trainingsumgebungen aus Abschnitt 4.3.3.

Tabelle 5.1: Auflistung der Experimente und verwendeter Parameter

Training	Karte	Reward	Verkehr	Kinematik	Aktionen	Δt	FE	PRF
Exp. 01	orig.	komplex	mittel	Diff. Drive	wenig	1	nein	ja
Exp. 02	klein	einfach	wenig	Diff. Drive	wenig	1	nein	nein
Exp. 03	klein	einfach	viel	Diff. Drive	wenig	1	nein	nein
Exp. 04	klein	einfach	viel	Diff. Drive	wenig	1	nein	ja
Exp. 05	klein	einfach	sehr viel	Diff. Drive	wenig	1	nein	ja
Exp. 06	groß	einfach	mittel	Bicycle	viel	3	ja	ja
Exp. 07	groß	einfach	mittel	Bicycle	viel	3	ja	ja
Exp. 08	groß	einfach	mittel	Bicycle	viel	3	ja	ja

Zunächst liegt der Fokus bis einschließlich Experiment 05 auf einem Fahrzeug mit

Differential Drive Kinematik. Das Kartenmaterial besteht aus einigen miteinander verbundenen Häuserblocks nahe des Universitätsgeländes, in deren Innenhöfen künstliche Fußgängerzonen angelegt werden, um belebte Plätze zu simulieren. Das Fahrzeug startet in der Mitte der Karte und folgt Routen in alle Himmelsrichtungen, die bewusst durch mindestens eine Menschenmenge verlaufen. Durch die Wahl eines Zielradius von 1 Meter wird erzwungen, dass der Fahragent lernt, sicher in die Menschenmenge einzutauchen und wieder herauszufahren. Die *Ped-Robot Force* (PRF) ist hierbei zunächst deaktiviert, sodass die Fußgänger das Fahrzeug nicht wahrnehmen und demnach auch nicht ihrerseits ausweichen können. Zudem liefert der LiDAR-Sensor nur die Entfernungen eines Zeitschritts (Standbild), woraus für den Agent keine Fußgängerdynamiken ableitbar sind. Des weiteren verfügt das Neuronale Netz des Agenten über keinen Feature Extractor (FE), sondern verarbeitet flache, konkatenierte Feature-Vektoren aller Sensoren. Die Fußgängerdichte wird während des Trainings auf 0.01 Fußgänger pro m^2 eingestellt, was einer relativ lückenhaften Menschenmenge entspricht und im Laufe der Versuchsreihe hin zu moderaten Menschenmassen mit Dichte 0.04 erhöht. Anschließend wird das Fahrverhalten der trainierten Agenten jeweils im Live Debugging begutachtet, wobei schrittweise die Fußgängerdichte von 0.01 Fußgänger pro m^2 auf 0.08 angehoben wird.

Es kann bei den zugehörigen Experimenten 02 und 03 beobachtet werden, dass der Fahragent erwartungsgemäß eine Strategie erlernt, die im fußgängerfreien Bereich geradlinig zum Ziel fährt und beim Eintauchen in Menschenmengen ebenfalls zielstrebig eine Lücke findet, um den jeweiligen Wegpunkt zu erreichen. Bei höheren Fußgängerdichten wird anstatt des schnellen Eintauchens in die Menschenmenge ein abwartendes Verhalten beobachtet, wobei der Agent so lange neben der Menge stehen bleibt, bis sich eine befahrbare Lücke öffnet. Die gelernten Verhaltensweisen erfüllen zwar die von der Trainingsumgebung gestellte Aufgabe, sind jedoch für praktische Erprobungen eher ungeeignet, da oftmals Situationen vorliegen, bei denen kein Fortschritt zu erzielen ist, wenn nur außerhalb der Menschenmenge gewartet wird.

Um praxistauglichere Verhaltensweisen zu erhalten, wird eine weitere Versuchsreihe anhand der Experimente 04 und 05 mit moderaten bis hohen Fußgängerdichten und aktivierter *Ped-Robot Force* durchgeführt, sodass die Fußgänger nun das Fahrzeug sehen können und eigenständig ausweichen. Die Kraft wird so gewichtet, dass ein stehendes Fahrzeug relativ eng von den Fußgängern umlaufen wird. Dadurch muss der Agent keine Zusammenstöße befürchten, ist aber zugleich zum Abwarten gezwungen, bis die Passage frei ist. Bei einem entsprechend trainierten Agent wird ein Fahrverhalten beobachtet,

bei dem das Fahrzeug frontal in die Menschenmenge eintaucht und sich langsam je nach Freiraum vor dem Fahrzeug vorwärts zum Wegpunkt bewegt. Wie in Abbildung 5.1 zu sehen ist, bleibt das Fahrzeug stehen und lässt die Fußgänger passieren, solange es komplett von Fußgängern umgeben ist und wartet ab, bis erneut Fortschritte zu erzielen sind. Diese Verhaltensweise liefert zufriedenstellende Ergebnisse, jedoch lenkt der Agent nach dem Erreichen von Wegpunkten teilweise abrupt, da ihm die Peilung des nächsten Wegpunkts nicht bekannt ist und er zudem durch eine niedrige Aktionsfrequenz von nur 2.5 Aktionen pro Sekunde keine Möglichkeit für exaktere Reaktionen hat.

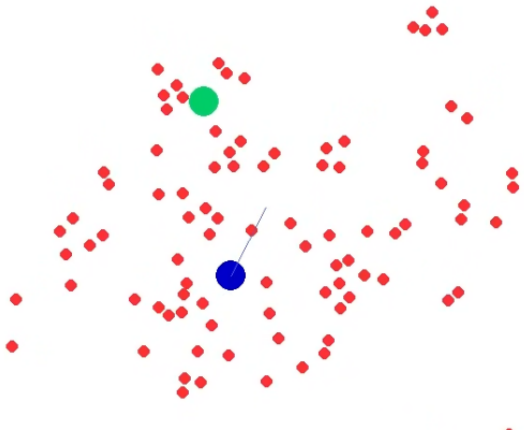

Abbildung 5.1: Sicheres Durchqueren einer dichten Menschenmasse

Um dynamischere Trajektorien fahren zu können, wird dem Agent eine zusätzliche Zielpeilung des nächsten Wegpunkts bezüglich seiner aktuellen Orientierung bereitgestellt. Zudem wird die Aktionsfrequenz auf 10 Aktionen pro Sekunde erhöht, um eine exaktere Fahrweise zu ermöglichen. Der anschließende Trainingslauf bringt jedoch keine wesentliche Verbesserung und wird deshalb verworfen. Es wird experimentell nachgewiesen, dass die bei einer niedrigen Aktionsfrequenz trainierten Agenten in einer Simulationsumgebung mit hoher Aktionsfrequenz gute Ergebnisse erzielen. Dies ist vermutlich auf die sehr defensive Fahrweise der Agenten zurückzuführen, wofür nicht so viele Aktionen pro Sekunde notwendig sind.

Da mit den bisherigen Trainingsläufen bereits sehr gute Verhaltensweisen in dichten Menschenmengen erzielbar sind, wird nun eine größere Karte des Universitätscampus als Trainingsumgebung verwendet, die Fußgängerrouten auf allen befahrenen Gehwegen und einige Fußgängerzonen an Engstellen aufweist. Um die Erlernbarkeit der Steuerung eines selbstfahrenden E-Scooters zu demonstrieren, wird eine zusätzliche Fahrzeugkine-

matik anhand des Fahrradmodells erprobt. Zur Modellierung von Fußgängerdynamiken stehen dem Agent die Sensordaten der letzten 3 Simulationsschritte zur Verfügung, welche mit einem Feature Extractor vorverarbeitet werden.

Die aus den folgenden Experimente 06, 07 und 08 resultierenden Agenten sind durch die neue Kinematik anhand des Fahrradmodells in der Lage, rückwärts zu fahren, wodurch sich die erlernte Verhaltensweise stark ändert. Durch die zudem verbesserte Erkennung der Fußgängerdynamiken können gefährlichere Fahrmanöver beobachtet werden, bei denen der Agent auf einem Gehweg frontal auf einen entgegenkommenden Fußgänger zu fährt, im letzten Moment stark rückwärts beschleunigt und dann eine ausweichende Kreisbahn einschlägt. Eine entsprechende Verhaltensweise scheint für den echten Verkehr jedoch ungeeignet, weshalb in der Fahrzeugkinematik das Rückwärtsfahren anschließend verboten wird. Eine darauffolgende Auswertung ohne Rückwärtsfahren bestätigt die Annahme, dass der Agent das Rückwärtsfahren aktiv in seinen Fahrmanövern einsetzt, da der Agent nun beim Ausweichen rückwärts fahren möchte, aber nicht mehr kann und daraufhin mit dem Fußgänger kollidiert.

Weitere Experimente können aufgrund des beschränkten zeitlichen Rahmens dieser Arbeit nicht mehr durchgeführt werden. Offene Fragen bleiben, wie gut ein Agent mit Differential Drive Kinematik bzw. Fahrradmodell ohne Rückwärtsfahren in der großen Trainingsumgebung des Universitätscampus lernt.

5.2 Evaluation mittels Unfallmetriken

Zur Auswertung der erlernten Fahragenten werden die in Abschnitt 4.1.10 definierten Unfallmetriken herangezogen. Als Kartenmaterial dient der Campus der Universität Ausgburg. Die Agenten steuern ein Fahrzeug mit denselben kinematischen Eigenschaften wie zur Trainingszeit.

Um das erlernte Verhalten bereits während des Trainings beurteilen zu können, werden entsprechende Metriken zudem auch mittels gleitendem Durchschnitt über die letzten 10 gefahrenen Routen pro Simulationsumgebung erhoben. Wie in Abbildung 5.2 dargestellt zeigt sich allgemein, dass die Agenten zu Beginn eines Trainingslaufs zunächst lernen, Kollisionen zu vermeiden, jedoch oftmals den anzusteuernden Wegpunkt nicht erreichen, was sich in einer erhöhten *Timeout Rate* bzw. Episodendauer äußert. Anschließend erfolgt die Erprobung von Fahrmanövern, wobei die dafür benötigte, simulierte Zeit

minimiert wird, was durch eine langsam sinkende, durchschnittliche Episodendauer messbar ist. Da der Agent nun riskantere Fahrmanöver auswählt, erhöhen sich in der Regel die Kollisionsraten. Stationären Hindernissen kann leicht ausgewichen werden, weshalb es hauptsächlich zu Kollisionen mit Fußgängern kommt. Gegen Ende des Trainings lernt der Agent schließlich, die Bewegung der Fußgänger zu interpretieren und ihnen effektiv auszuweichen.

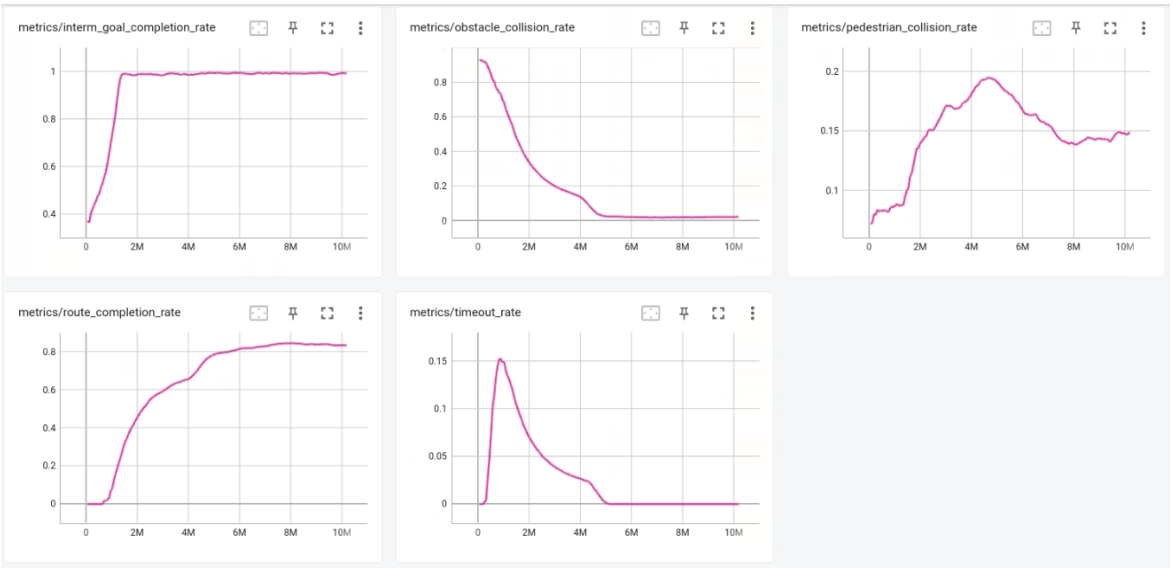

Abbildung 5.2: Typischer Verlauf der Unfallmetriken während des Trainings. Zu sehen sind zeilenweise von rechts nach links die Komplettierungsrate von Wegpunkt zu Wegpunkt, die Kollisionsraten mit Fußgängern bzw. Hindernissen, die Rate komplettierter Routen und die Timeout-Rate. Auf x- und y-Achse sind jeweils die Zeit und die Rate zwischen 0 und 1 aufgetragen.

Aufgrund der Stochastizität der erlernten Strategien unterliegen die Unfallmetriken starken Schwankungen. Es genügen oftmals schon wenige, fehlerhafte Vorhersagen, um einen Unfall herbeizuführen, der bei einer deterministischen Schätzung nicht geschehen wäre. Daher werden zur Evaluation immer die geschätzten Mittelwerte für die Aktuatoren statt der während des Trainings üblichen, normalverteilten Schätzungen verwendet. Aufgrund der Notwendigkeit, die für die Modellaktualisierung benötigten Trainingsdaten mit einer stochastischen Strategie zu sammeln, haben zur Trainingszeit erhobene Unfallmetriken nur eine geringe Aussagekraft und dienen lediglich als grobe Anhaltspunkte. Die folgenden Ergebnisse werden deshalb mit deterministisch vorhergesagten Aktionen austrainierter Agenten ausgewertet. Je Auswertung fährt der Agent 100 Routen, sodass die resultierenden Ergebnisse auf 1% genau sind. Insbesondere

für die Bereiche $[0.00, 0.01]$ und $[0.99, 1.00]$ besteht daher keine Aussagekraft. Dies stellt jedoch kein Problem dar, da die zu vergleichenden Ergebnisse ohnehin keine entsprechende Qualität aufweisen.

Zur Evaluation werden alle erfolgreichen Trainingsläufe aus Tablelle 5.1 betrachtet. Experiment 01 dient als Grundlinie für die ursprüngliche Umsetzung der Trainingsumgebung durch Caruso et. al [1] bezüglich Kartenmaterial und Rewardstruktur.

Tabelle 5.2: Auswertung der Unfallzahlen ohne Fußgänger

Unfallmetrik vs. Trainingslauf	Completion	Obst. Coll.	Ped. Coll.	Timeout
Experiment 01	0.08	0.00	0.00	0.92
Experiment 02	0.81	0.19	0.00	0.00
Experiment 03	0.05	0.00	0.00	0.95
Experiment 04	0.30	0.00	0.00	0.70
Experiment 05	0.14	0.00	0.00	0.86
Experiment 06	0.73	0.27	0.00	0.00
Experiment 07	1.00	0.00	0.00	0.00
Experiment 08	1.00	0.00	0.00	0.00

Wie in Tabelle 5.2 zu sehen ist, wird zunächst zur Kontrolle eine Auswertung ohne Fußgänger durchgeführt, um zu prüfen, ob die jeweilige Fahrsoftware statischen Hindernissen ausweichen kann und die einfachere Aufgabe ohne bewegliche Hindernisse einwandfrei lösen kann. Hier wäre eine Rate von 100% zu erwarten, da die Problemstellung deutlich einfacher als mit Fußgängern ist. Es zeigt sich aber, dass einige Agenten die Problemstellung nur unzureichend lösen, was vielfältige Gründe haben kann. Beispielsweise ist in der Live-Ansicht beobachtbar, dass die Agenten aus den Experimenten 02 bis 05 die Route bis ans Ziel fahren, aber den letzten Wegpunkt meiden, weil sich dieser zu nah an einem Gebäude befindet. Vermutlich ist ein entsprechendes Verhalten auf den Mangel an Engstellen im Kartenmaterial zurückzuführen. Interessanterweise schneidet der mit einer komplexen Rewardstruktur trainierte Agent in einer Umgebung ohne Fußgänger erstaunlich schlecht ab, was nur schwer nachvollziehbar ist. Hingegen können die aus den Experimenten 07 und 08 resultierenden Agenten überzeugen, was vermutlich daran liegt, dass die Agenten mit demselben Kartenmaterial trainiert wurden.

In der zweiten Evaluation, deren Ergebnisse in Tabelle 5.3 zu sehen sind, werden die trainierten Agenten in einer Umgebung mit einer geringen Fußgängerdichte getestet. Hierbei wird geprüft, ob die jeweilige Fahrsoftware vereinzelten, beweglichen Hinder-

Tabelle 5.3: Auswertung der Unfallzahlen mit Fußgängerdichte 0.02

Unfallmetrik vs. Trainingslauf	Completion	Obst. Coll.	Ped. Coll.	Timeout
Experiment 01	0.89	0.00	0.00	0.11
Experiment 02	0.59	0.13	0.28	0.00
Experiment 03	0.07	0.00	0.60	0.33
Experiment 04	0.38	0.06	0.02	0.54
Experiment 05	0.64	0.00	0.00	0.36
Experiment 06	0.18	0.25	0.57	0.00
Experiment 07	0.95	0.02	0.03	0.00
Experiment 08	0.93	0.01	0.06	0.00

nissen ausweichen kann und dabei nicht mit statischen Hindernissen Kollidiert. Es zeigt sich, dass die Agenten mit kinematischem Fahrradmodell und einer verbesserten Sensorik besonders gut abschneiden, da sie die Bewegungsdynamiken aus den Sensordaten extrahieren und somit deren Bewegung einschätzen können, um adäquat auszuweichen. Hingegen weisen die Experimente mit Differential Drive Kinematik und einer aus Standbildern bestehenden Sensorik deutlich defensivere Verhaltensweisen auf, was auch im Live Debugging zu beobachten ist. Der Ansatz mit alter Belohnungsstruktur und Kartenmaterial aus Experiment 01 schneidet gut ab, was möglicherweise auf den Belohnungsterm zur Annäherungung an den nächsten Wegpunkt zurückzuführen ist.

Tabelle 5.4: Auswertung der Unfallzahlen mit Fußgängerdichte 0.08

Unfallmetrik vs. Trainingslauf	Completion	Obst. Coll.	Ped. Coll.	Timeout
Experiment 01	0.20	0.08	0.07	0.65
Experiment 02	0.16	0.11	0.73	0.00
Experiment 03	0.02	0.04	0.76	0.18
Experiment 04	0.59	0.07	0.11	0.23
Experiment 05	0.61	0.04	0.00	0.35
Experiment 06	0.01	0.20	0.79	0.00
Experiment 07	0.60	0.09	0.30	0.01
Experiment 08	0.51	0.13	0.35	0.01

In den letzten beiden Evaluationen, deren Ergebnisse in den Tabellen 5.4 und 5.5 zu sehen sind, werden die trainierten Agenten in einer Umgebung mit sehr dichtem Verkehr getestet. Hierbei steht die Unfallvermeidung, ohne Schäden an Personen, Hindernissen und dem Fahrzeug zu verursachen im Vordergrund. Um die Aufgabe anspruchsvoller zu

gestalten, müssen die Agenten bewusst durch belebte Zonen hindurch fahren, was durch entsprechend positionierte Wegpunkte erreicht wird. Die Schwierigkeit der Aufgabe besteht nun darin, selbst bei dichtem Verkehr noch sicher Fortschritte erzielen zu können. Um die Aufgabe überhaupt lösbar zu gestalten, weichen die Fußgänger dem Fahrzeug mittels *Ped-Robot Force* aus. Besonders hervorzuheben sind die Ergebnisse aus Experiment 05, wobei keine einzige Kollision mit Fußgängern festzustellen ist. Der Agent kommt mit einer Ankunftsrate von über 60% am Routenziel an und wartet in fast allen weiteren Fällen ab. Die Ansätze mit kinematischem Fahrradmodell und verbesserter Sensorik erreichen auch noch nennenswert oft ihr Ziel, weisen jedoch eine offensive Fahrweise auf. Dies kann zum einen an der Durchführung der Trainingsläufe bei weitaus geringerer Verkehrsdichte liegen. Zum anderen ist auch denkbar, dass sich die Agenten zu sicher sind, die Dynamiken der Fußgänger korrekt einschätzen zu können, was anschließend zu Unfällen führt. Der mit komplexer Rewardstruktur trainierte Agent aus Experiment 01 kann nur schwer Fortschritte erzielen, verursacht jedoch auch keine erhebliche Anzahl an Unfällen.

Tabelle 5.5: Auswertung der Unfallzahlen mit Fußgängerdichte 0.10

Unfallmetrik vs. Trainingslauf	Completion	Obst. Coll.	Ped. Coll.	Timeout
Experiment 01	0.07	0.10	0.12	0.71
Experiment 02	0.10	0.18	0.72	0.00
Experiment 03	0.02	0.00	0.79	0.19
Experiment 04	0.39	0.10	0.20	0.31
Experiment 05	0.62	0.03	0.00	0.35
Experiment 06	0.00	0.17	0.83	0.00
Experiment 07	0.50	0.15	0.32	0.03
Experiment 08	0.33	0.14	0.52	0.01

5.3 Vergleich der Ergebnisse

Für einen fairen Vergleich werden die veröffentlichten Ergebnisse von Caruso et. al [1] herangezogen, da hier die meisten fachlichen und konzeptionellen Überschneidungen bestehen und dieselben Unfallmetriken zur Auswertung verwendet werden. Ihre Ergebnisse werden in Tabelle 5.6 zusammegefasst.

Es zeigt sich, dass der Ansatz mit *RobotSF* in seiner ursprünglichen Form zuverlässig in Szenarien mit wenigen Fußgängern funktioniert und in etwa mit Experiment 07

Tabelle 5.6: Auswertung der Unfallzahlen von Caruso et. al [1] mit A3C

Unfallmetrik vs. Fußgängerdichte	Completion	Obst. Coll.	Ped. Coll.	Timeout
0.00 / m^2	1.00	0.00	0.00	0.00
0.02 / m^2	0.87	0.00	0.13	0.00
0.08 / m^2	0.67	0.00	0.32	0.01
0.10 / m^2	0.52	0.01	0.47	0.00

vergleichbar ist. Bei dichtem Verkehr weist die Fahrsoftware starke Schächen auf, da hohe Kollisionsraten mit Fußgängern bestehen. Die defensive Strategie aus Experiment 05 erreicht ähnlich gute Ankunftsraten, verursacht jedoch keine Kollisionen mit Fußgängern. Es kann also von einer klaren Qualitätsverbesserung bei der Fahrsicherheit gesprochen werden. Der Trainingsansatz, Agenten gezielt durch große Menschenmengen fahren zu lassen, scheint vielversprechend. Ähnlich wie in Fan et. al [4] könnte eine Kombination einer defensiven und offensiven Fahrweise der Agenten aus Experiment 05 und 07 das Fahrverhalten nochmals deutlich verbessern, indem bei wenig Verkehr die offensive und bei viel Verkehr die defensive Fahrweise zum Einsatz kommt.

Da die Umsetzung einer Simulationsumgebung mit mehreren gleichzeitig agierenden Fahrzeugen wie im Ansatz von *Multi-Robot* zeitlich nicht mehr möglich war, ist ein Vergleich mit Fan et. al [4] nicht sinnvoll.

5.4 Optimierung der Sample Efficiency

Dieser Abschnitt befasst sich mit der Verbesserung der Trainingsdateneffizienz, auch engl. Sample Efficiency genannt. Wie in vorherigen Abschnitten demonstriert wurde, sind für qualitativ hochwertige Fahragenten keine rechenintensiven Neuronalen Netze notwendig. Dementsprechend wenden Trainingsalgorithmen verhältnismäßig viel Zeit für das Sammeln der Trainingsdaten auf. Ist eine hinreichende Performanz der Simulationsumgebung hinsichtlich der inhärenten Berechnungskomplexität gegeben, muss das gewählte Lernverfahren dahingehend optimiert werden, bereits gute Ergebnisse mit möglichst wenigen Trainingsdaten zu erzielen. Die Durchführbarkeit von Lerntechniken des Bestärkenden Lernens in komplexen Simulationsumgebungen hängt demnach stark von der Wahl eines geeigneten Lernverfahrens und dessen Parametrisierung ab. In den folgenden Abschnitten wird daher untersucht, wie die Trainingsdateneffizienz beim Erlernen von Fahrverhaltensweisen gesteigert werden kann.

5.4.1 Optimierung der Lernparameter

Wie bereits im Abschnitt über PPO 2.5.2 erläutert, kann die Trainingsdateneffizienz erheblich durch die Wahl eines geeigneten Lernverfahrens gesteigert werden. Steht ein geeigneter Lernalgorithmus fest, können dessen Parameter an die jeweilige Problemstellung angepasst werden, um weitere Effizienzsteigerungen zu erreichen.

In diesem Zusammenhang stellt sich Effizienz als das Erlernen von Fahrqualität pro Trainingszeit dar. Hierbei wird die Qualität durch die Rate unfallfrei gefahrener Routen (*Route Completion Rate*) und die Trainingszeit durch die simulierten Trainingsschritte bemessen. Um diskrete Zeitpunkte bei der Erreichung einer gewissen Qualität zu bestimmen, werden Qualitätsstufen in Schritten von jeweils 1% definiert. Die Anregung einer guten Qualität soll doppelt so hoch als deren möglichst schnelle Erreichung gewichtet werden, sodass zunächst Parametrisierungen gefunden werden, die eine ausreichende Qualität ermöglichen. Anschließend wird die benötigte Trainingszeit für deren Erreichung minimiert. Es ergibt sich folgende zwischen 0 und 1 normierte Bewertungsfunktion U für die erreichten Qualitätsstufen $Q \subseteq \{q_1, q_2, ..., q_{100}\}$ und die für deren erstmalige Erreichung aufgewendeten Simulationsschritte $s(q)$ mit maximalen Schritten s_{max}.

$$U(Q) = \frac{\sum_{q_i \epsilon Q} q_i \cdot (2 + \frac{s_{max} - s(q_i)}{s_{max}})}{3 \cdot \sum_{q_i \epsilon Q} q_i} \tag{5.1}$$

Da die gleichzeitige Optimierung aller Parameter sehr zeitaufwendig und kostspielig ist, wurden die optimierbaren Parameter in 3 Gruppen unterteilt und nacheinander mit *Optuna* [37] verbessert. Es handelt sich um Parameter der Simulationsumgebung, des Lernverfahrens PPO und der Belohnungsfunktion.

Optimierung des Lernverfahrens

Bei der Optimierung des Lernverfahrens wurde zum einen die Auswirkung verschiedener Modellstrukturen und zum anderen die Wahl der Lernparameter von PPO untersucht. Als Modellstrukturen steht zur Wahl, ob ein Feature Extractor verwendet werden soll. Falls ein Feature Extractor zum Einsatz kommt, können für jede Faltungsschicht die Anzahl der Filter, die Größe des Faltungskerns und die Dropout-Rate gewählt werden. Zudem können die Anzahl der Simulationsschritte bis zur nächsten Modellaktualisierung, die während der Modellaktualisierung durchgeführten PPO Updates mit denselben

Tabelle 5.7: Zu optimierende Parameter der Modellstruktur und des Lernverfahrens

Feature Extractor	{ ja, nein }
Anzahl der Faltungen	{ 8, 16, 32, 64, 128, 256 }
Größe des Faltungskerns	{ 3, 5, 7, 9 }
Dropout-Rate	(0, 1)
Anzahl Environments	{ 32, 40, 48, 56, 64 }
Schritte zum nächsten Update	{ 128, 256, 512, 1024, 2048 }
Anzahl PPO Epochs	{ 2, 3, ..., 20 }

Trainingsdaten und die Anzahl der parallel laufenden Simulationsumgebungen gesteuert werden. Folgende Tabelle zeigt die möglichen Parametrisierungen je Parameter.

Es zeigt sich, dass eine Modellstruktur mit Feature Extractor und möglichst vielen Filtern mit Kerngröße 5 in den ersten Faltungsschichten bevorzugt wird. Des weiteren kann die von Stable Baselines 3 voreingestellte Anzahl der Trainingsepochen mit denselben Trainingsdaten von 10 empirisch bestätigt werden. Als Dropout-Raten wählt Optuna für die ersten Faltungsschichten eine etwas niedrigere Rate von 15%, bestätigt jedoch die 30% der folgenden Schichten. Die Simulationsschritte bis zur nächsten Modellaktualisierung können mit 1024 etwas niedriger als zuvor mit 2048 gewählt werden. Da trainierte Agenten ca. 3-5 Sekunden simulierte Zeit benötigen, um von Wegpunkt zu Wegpunkt zu navigieren, was je nach Aktionsfrequenz maximal 50 Simulationsschritten entspricht, kann der Parameter ggf. noch weiter gesenkt werden. Damit die für eine Modellaktualisierung verwendete Trainingsdatenmenge gleich groß bleibt, kann im Gegenzug der Parallelisierungsgrad durch mehr Simulationsumgebungen erhöht werden.

Optimierung der Belohnungsstruktur

Wie bereits erwähnt spielt die Wahl einer geeigneten Belohnungsfunktion eine wichtige Rolle beim Bestärkenden Lernen. Relativ zu der auf 1 normierten Belohnung für das Erreichen des Wegpunkts experimentiert Optuna mit der Höhe des Step Discounts und den Strafen für Kollisionen mit Fußgängern und Hindernissen. Folgende Tabelle listet die möglichen Parametrisierungen je Parameter auf.

Eine entsprechend durchgeführte Optimierung bestätigt größtenteils die initiale Parameterbelegung aus der Konzeption in Abschnitt 4.1.9. Die besten Trainingsläufe weisen einen Step Discount von 0.15 auf, was sehr nah am ursprünglich gewählten Wert

Tabelle 5.8: Zu optimierende Parameter der Belohnungsstruktur

Erreichen des Wegpunkts	{ 1 }
Kollision mit Fußgänger	{ -10, -9, ..., -1 }
Kollision mit Hindernis	{ -10, -9, ..., -1 }
Step Discount	[-1, 0]

von 0.1 liegt. Auch bei den Strafen für Kollisionen mit Fußgängern und Hindernissen werden ähnliche Werte mit 2 und 5 gewählt. Hintergrund für die ursprüngliche Wahl derselben Bestrafung für Kollisionen mit Fußgängern und Hindernissen ist, dass der Agent auf einem Standbild die verschiedenen Entitäten anhand der Entfernungen des LiDAR-Sensors nicht unterscheiden kann. Während der Optimierung stehen dem Agent jedoch die Sensordaten der letzten 3 Zeitschritte zur Verfügung, sodass sehr wohl die Dynamiken der Fußgänger extrahiert werden können und folglich eine höhere Bestrafung für Kollisionen mit Personenschäden durchaus sinnvoll erscheint.

Optimierung der Simulationseinstellungen

Sinnvolle Parameter für die Optimierung der Simulationsumgebung betreffen vor allem die Sensorik, die dem Agent während des Trainings zu Verfügung steht. Dies umfasst die Anzahl der gemessenen LiDAR-Strahlen, die Anzahl der zu Verfügung stehenden Zeitschritte, sowie die Präsenz der Peilung des nächsten Ziels. Zusätzlich kann die zwischen den Zeitschritten vergangene, simulierte Zeit Δt bezüglich der Aktionsrate des Fahragenten gewählt werden. Folgende Tabelle gibt die möglichen Parametrisierungen je Parameter an.

Tabelle 5.9: Zu optimierende Parameter der Simulationsumgebung

Anzahl LiDAR-Strahlen	{ 144, 176, 208, 272 }
Anzahl Stacked Steps	{ 1, 2, 3, 4, 5 }
Peilung nächstes Ziel	{ ja, nein }
Simulationsschritt Δt	[0.1, 0.2, 0.3, 0.4]

Da eine entsprechende Versuchsreihe mehr als eine Woche gedauert hätte, kann die Optimierung der Simulationseinstellungen nicht mehr rechtzeitig fertiggestellt werden. Es wäre jedoch interessant zu erfahren, ob die gewählten Simulationseinstellungen während der Lernexperimente bereits optimal sind.

5.4.2 Effizienzsteigerung durch Modellbasiertes Lernen

Einen weiteren Ansatz zur Steigerung der Trainingsdateneffizienz stellt eine Anwendung des Modellbasierten Lernens aus Abschnitt 2.7 dar. In den Dreamer Veröffentlichungen werden Beschleunigungen der Trainingszeiten um 2-3 Magnituden in Aussicht gestellt, was die damit verbundenen Anstrengungen durchaus motiviert. Die folgenden Experimente und deren Ergebnisse liegen im Appendix als Jupyter Notebooks bereit.

Zur Anwendung von Dreamer auf *RobotSF* wird zunächst versucht, die Atari-Spiele Pong und MsPacman nachzustellen, um anschließend die Erkenntnisse auf *RobotSF* zu übertragen. Hierfür dient die Dreamer-Architektur (Version 2) [24] als Vorlage. Jedoch zeigt sich schnell, dass mit den zur Verfügung stehenden Rechenkapazitäten keine Ergebnisse innerhalb einer vertretbaren Zeit zu erwarten sind. Als Grund für die Fehlschläge wird die Umsetzung des VAE identifiziert, die die Ausgabe des Encoders mit den Dynamiken vorheriger Standbilder konkateniert und anschließend die kategorische Repräsentation per Softmax Aktivierung und Straight-Through Gradients [38] vorhersagt. Anhand der Rekonstruktion der Standbilder mit dem Decoder zeigt sich, dass nach mehr als einem Tag Trainingszeit keinerlei Informationen über die in den Standbildern wahrnehmbaren, beweglichen Objekte vorhanden sind. Dies kann auch durch einen viel zu hohen Rekonstruktionsfehler gemessen werden.

Es wird daher auf einen Vector-Quantized Variational Autoencoder (VQ-VAE) [39] zurückgegriffen, der sich dem Konzept der Vektorquantisierung aus der Physik bedient. Der VQ-VAE lernt eine Embedding-Tabelle mit quantisierten Vektoren und ersetzt die einzelnen Teilvektoren der Ausgabe des Encoders durch den ähnlichsten Vektor aus der Embedding-Tabelle. Anhand der Indizes der gewählten Vektoren aus der Tabelle ergeben sich automatisch die Kategorien der latenten Repräsentation. Durch den Einsatz von Straight-Through Gradients und dem Codebook Loss wird der eigentlich nicht differenzierbare Quantisierungsschritt dennoch differenzierbar und somit kompatibel zur Backpropagation mit Neuronalen Netzen. Folgende Experimente ergeben, dass der VQ-VAE innerhalb weniger Stunden eine brauchbare, latente Repräsentation erlernt.

Jedoch liefert die Dreamer-Architektur nach der Integration des VQ-VAE weiterhin unbrauchbare Ergebnisse. Ursache hierfür ist das Hinzufügen der Dynamiken mit anschließender Verarbeitung durch eine vollvermaschte Neuronenschicht, da der VQ-VAE nur gut funktioniert, wenn er die unveränderte Ausgabe des Encoders als Eingabe in die latente Vektorquantisierung erhält. Das Konkatenieren der Dynamiken hat keinen

Einfluss, was experimentell durch Hinzufügen von Rauschen statt echter Dynamiken gezeigt werden kann. Die Architektur wird daher entsprechend umgestaltet, indem ein nicht adaptierter VQ-VAE latente Repräsentationen von Standbildern lernt. Anschließend werden die mittels Encoder gewonnenen, latenten Repräsentationen als Trainingsdaten für das Dynamikmodell verwendet. Vorteil dieser Architektur ist, dass eine gesonderte Validierung der einzelnen Komponenten möglich ist, um die Fehlersuche zu erleichtern. Nun kann ein bereits vortrainierter VQ-VAE verwendet werden, um anschließend das Dynamikmodell zu trainieren. Es zeigt sich, dass die verwendete GRU-Zelle des Dynamikmodells als Schwachstelle ausgeschlossen werden kann, da die Zustandsübergänge zwischen den latenten Repräsentationen mit über 99% Genauigkeit innerhalb von wenigen Minuten erlernbar sind. Die damit generierten Videosequenzen, sog. Träume, weisen jedoch nur eine sehr niedrige Qualität auf, was vermutlich auf die Qualität der Repräsentationen zurückzuführen ist.

Sofern es gelingt, einen brauchbaren VQ-VAE zu trainieren, könnte die Anwendung der Dreamer-Architektur auf *RobotSF* vielversprechende Ergebnisse liefern. Die Rede ist von Effizienzsteigerungen um 2-3 Magnituden. Da gerade die Stärke von Dreamer darin besteht, aus Zeitreihen hochdimensionaler Sensordaten zu lernen, könnte eine Übertragung der Ergebnisse von *RobotSF* auf deutlich rechenintensivere, fotorealistische Simulationsumgebungen wie beispielsweise CARLA ein Training innerhalb vertretbarer Wartezeiten ermöglichen. Sobald gute Repräsentations- und Dynamikmodelle von der Simulationsumgebung existieren, kann mittels aufgezeichneter Startsequenzen aus dem echten Simulator beispielsweise eine szenarienbasierte Falsifizierung von Fahragenten effizient umgesetzt werden.

6 Zusammenfassung und Ausblick

6.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde eine Fahrsoftware zur Lokalen Navigation von Mikromobilitätsfahrzeugen mittels Deep Reinforcement Learning entwickelt. Dabei fokussierten sich die Untersuchungen auf die sichere Interaktion des Fahrzeugs mit Fußgängern in Fußgängerzonen und auf Gehwegen, was durch die Erweiterung eines existierenden Simulators *RobotSF* umgesetzt werden konnte. Die 19-fache Beschleunigung der Simulationszeit ermöglichte die Durchführbarkeit zahlreicher Lernexperimente auf dem virtuellen Universitätscampus. Zusätzlich konnte durch das gewählte Lernverfahren der Proximal Policy Optimization eine weitere Effizienzsteigerung erzielt werden. In Kombination ermöglichten die Verbesserungen bezüglich der Effizienz, dass Trainingsläufe nun nicht mehr länger als einen Tag daueren, was im Vergleich zur vorherigen Umsetzung von Caruso et. al mit Trainingszeiten von bis zu einem Monat einen großen Fortschritt darstellt.

Die im adaptierten *RobotSF* Simulator durchgeführten Lernexperimente ergaben wertvolle Erkenntnisse bezüglich der Erlernbarkeit unfallsicherer Fahrverhaltensweisen. Durch Anpassungen im Kartenmaterial konnte sehr robustes, sicheres Verhalten in dichten Menschenmassen erlernt werden. Es wurde zudem deutlich, dass die Gestaltung der Trainingsumgebung anhand der Zusammenstellung des Kartenmaterials, der Konfiguration der Social Forces und der gewählten Belohnungsstruktur großen Einfluss auf die erlernten Fahrverhaltensweisen hat. Unter anderem konnte die Verkehrssicherheit bei dichtem Verkehr durch eine defensive Strategie im Vergleich zu vorherigen Ansätzen erheblich verbessert werden. Die Kollisionsrate mit Fußgängern wurde von vormals 47% auf 0% gesenkt. Hierbei mussten keinerlei Kompromisse bei der Ankunftsrate hingenommen werden, da die defensive Strategie eine ähnlich hohe Ankunftsrate wie vergleichbare Agenten von Caruso et. al aufwies.

Die Repräsentation guter Agenten mittels kleiner Modelle mit weniger als einer Million

Gewichten demonstrierte eindrücklich, dass die Trainingszeiten aufgrund der effizient berechenbaren Modellvorhersagen durch das Sammeln der Trainingsdaten dominiert werden. Hinsichtlich der Erhöhung der Trainingsdateneffizienz wurde die Optimierung der Trainingseinstellungen untersucht, wobei die meisten Parameter aus der Konzeption der Modell- und Belohnungsstrukturen bestätigt und einige weitere Parameter verbessert werden konnten. Vor allem die einfachere Belohnungsstruktur ermöglichte das Erlernen vielfältiger Verhaltensweisen und greift nicht zu stark in die Lösungsfindung des Agenten ein, was eine Verbesserung gegenüber vergleichbaren Arbeiten darstellt.

Zusammenfassend kann gesagt werden, dass die Ziele dieser Arbeit für den Anwendungsfall der sicheren Lokalen Navigation in Fußgängerzonen und auf Gehwegen erreicht wurden. Außerdem wurden einige interessante Ansätze bezüglich der Entwicklung sicherer Fahrsoftware mittels Neuronaler Netze präsentiert, die weiter verfolgt werden können. Es ist klar, dass es bis zur Entwicklung einer Fahrsoftware, die im normalen Straßenverkehr eingesetzt werden kann, noch ein weiter Weg ist.

6.2 Ausblick

Aufgrund der guten Eigenschaften der erlernten Fahrsoftware ist mittelfristig eine Erprobung in der Realität mit einem echten Fahrzeug denkbar. Hierfür bietet sich der Campus der Universität Augsburg an, da die Evaluation der trainierten Fahrsoftware ohnehin auf dem virtuellen Modell des Campus durchgeführt wurde.

Durch das Hinzufügen zusätzlicher Verkehrsteilnehmer, Fahrzeugkinematiken und Entitäten zur Simulationsumgebung kann die Fahrsoftware in noch realistischeren Szenarien außerhalb von Gehwegen und Fußgängerzonen erprobt werden. Möglichkeiten für die Erweiterung wurden aufgezeigt. Zudem kann auch eine weitere Effizienzsteigerung der Simulationslogik mittels lokalitätsaffiner Datenstrukturen und Algorithmen interessant sein, um noch mehr Experimente durchführen zu können. Auch die gleichzeitige Simulation mehrerer Fahrzeuge in derselben Umgebung verspricht zusätzliche Effizienzsteigerungen und eine Verbesserung der erlernten Fahrqualität.

Durch die Auslegung von Simulationsumgebung und Fahrsoftware auf Falsifizierbarkeit können nun in folgenden Arbeiten die Schwachstellen der trainierten Modelle identifiziert und behoben werden, um eine sichere Erprobung der Fahrsoftware in der Realität voranzutreiben.

Abbildungsverzeichnis

Tabellenverzeichnis

Literatur

[1] Matteo Caruso et al. „Robot Navigation in Crowded Environments: A Reinforcement Learning Approach“. In: *Machines* 11.2 (2023). ISSN: 2075-1702. DOI: 10.3390/machines11020268 (S. I, 2, 6, 30, 32, 36, 40, 51, 62, 67, 69, 70).

[2] *Verordnung über die Teilnahme von Elektrokleinstfahrzeugen am Straßenverkehr*. 2019. URL: https://www.gesetze-im-internet.de/ekfv/eKFV.pdf (S. 1).

[3] *Zukunftsfeld Mikromobile: E-Tretroller & Co.* 2019. URL: https://www.ivm-rheinmain.de/wp-content/uploads/2019/05/20190523_Zukunftsfeld-Mikromobile_Brosch%C3%BCre_final.pdf (S. 1).

[4] Tingxiang Fan et al. „Fully Distributed multi-robot collision avoidance via deep reinforcement learning for navigation in complex scenarios“. In: *The International Journal of Robotics Research* (2020) (S. 2, 30, 32, 38, 39, 60, 70).

[5] Shunyi Yao et al. „Multi-Robot Collision Avoidance with Map-based Deep Reinforcement Learning“. In: *2020 IEEE 32nd International Conference on Tools with Artificial Intelligence (ICTAI)*. 2020, S. 532–539. DOI: 10.1109/ICTAI50040.2020.00088 (S. 2, 30, 32).

[6] Dirk Helbing und Péter Molnár. „Social force model for pedestrian dynamics“. In: *Phys. Rev. E* 51 (5 Mai 1995), S. 4282–4286. DOI: 10.1103/PhysRevE.51.4282 (S. 2, 6).

[7] Mehdi Moussaïd et al. „The Walking Behaviour of Pedestrian Social Groups and Its Impact on Crowd Dynamics“. In: *PLOS ONE* 5.4 (Apr. 2010), S. 1–7. DOI: 10.1371/journal.pone.0010047 (S. 2, 6, 7, 36, 50).

[8] John Schulman et al. „Proximal Policy Optimization Algorithms“. In: *CoRR* abs/1707.06347 (2017) (S. 2, 19, 22).

[9] Gregor Klančar et al. „Motion Modeling for Mobile Robots“. In: *Wheeled Mobile Robotics from fundamentals towards Autonomous Systems*. Butterworth-Heinemann, 2017, S. 13–23 (S. 3).

[10] Thomas Fermi. *Algorithms for Automated Driving*. 2023. URL: https://thomasfermi.github.io/Algorithms-for-Automated-Driving/_images/BicycleModel_x_y_theta.svg (S. 4).

[11] Mr.Doctor.No. *Differential Drive Kinematics of a Wheeled Mobile Robot*. 2022. URL: https://commons.wikimedia.org/wiki/File:Differential_Drive_Kinematics_of_a_Wheeled_Mobile_Robot.svg (S. 5).

[12] Yen-Chen Liu et al. „Dynamic Modeling and Simulation of Electric Scooter Interactions With a Pedestrian Crowd Using a Social Force Model". In: *IEEE Transactions on Intelligent Transportation Systems* 23.9 (Sep. 2022), S. 16448–16461. ISSN: 1558-0016. DOI: 10.1109/TITS.2022.3150282 (S. 9).

[13] Nguyen Huynh Duc et al. „Modelling Mixed Traffic Flow at Signalized Intersection Using Social Force Model". In: *EASTS* 10 (März 2014) (S. 9).

[14] Richard S. Sutton und Andrew G. Barto. *Reinforcement Learning: An Introduction*. Second. The MIT Press, 2018 (S. 9, 11, 20).

[15] Claude E. Shannon. „Programming a Computer for Playing Chess". In: *Computer Chess Compendium*. Hrsg. von David Levy. New York, NY: Springer New York, 1988, S. 2–13. ISBN: 978-1-4757-1968-0. DOI: 10.1007/978-1-4757-1968-0_1 (S. 12).

[16] Volodymyr Mnih et al. *Playing Atari with Deep Reinforcement Learning*. 2013 (S. 15).

[17] John Schulman et al. *High-Dimensional Continuous Control Using Generalized Advantage Estimation*. 2018 (S. 19, 21).

[18] John Schulman et al. *Trust Region Policy Optimization*. 2017 (S. 19, 22).

[19] Ronald J. Williams. „Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning". In: *Machine Learning* 8 (2004), S. 229–256 (S. 27).

[20] Thomas Rückstieß, Martin Felder und Jürgen Schmidhuber. „State-Dependent Exploration for Policy Gradient Methods". In: Sep. 2008, S. 234–249. ISBN: 978-3-540-87480-5. DOI: 10.1007/978-3-540-87481-2_16 (S. 27).

[21] Antonin Raffin, Jens Kober und Freek Stulp. *Smooth Exploration for Robotic Reinforcement Learning*. 2021 (S. 27).

[22] David Ha und Jürgen Schmidhuber. „Recurrent World Models Facilitate Policy Evolution“. In: *Advances in Neural Information Processing Systems*. Hrsg. von S. Bengio et al. Bd. 31. Curran Associates, Inc., 2018 (S. 27, 29).

[23] Danijar Hafner et al. *Dream to Control: Learning Behaviors by Latent Imagination*. 2020 (S. 27).

[24] Danijar Hafner et al. *Mastering Atari with Discrete World Models*. 2022 (S. 27, 74).

[25] Danijar Hafner et al. *Mastering Diverse Domains through World Models*. 2023 (S. 27).

[26] Diederik P Kingma und Max Welling. *Auto-Encoding Variational Bayes*. 2022 (S. 28).

[27] David Ha und Jürgen Schmidhuber. *World Models*. 2018. URL: https://worldmodels.github.io/ (S. 29).

[28] B Ravi Kiran et al. „Deep Reinforcement Learning for Autonomous Driving: A Survey“. In: *IEEE Transactions on Intelligent Transportation Systems* 23.6 (2022), S. 4909–4926. DOI: 10.1109/TITS.2021.3054625 (S. 30–32, 36).

[29] Alexey Dosovitskiy et al. „CARLA: An Open Urban Driving Simulator“. In: *Proceedings of the 1st Annual Conference on Robot Learning*. 2017, S. 1–16 (S. 31).

[30] Yuxiang Gao. *Py Social Force*. 2020. URL: https://github.com/yuxiang-gao/PySocialForce (S. 33, 51).

[31] Enrico Regolin Matteo Caruso. *RobotSF*. 2021. URL: https://github.com/EnricoReg/robot-sf (S. 33).

[32] Winston H. *KinematicBicycleModel*. 2021. URL: https://github.com/winstxnhdw/KinematicBicycleModel (S. 34).

[33] Antonin Raffin et al. „Stable-Baselines3: Reliable Reinforcement Learning Implementations“. In: *Journal of Machine Learning Research* 22.268 (2021), S. 1–8 (S. 51).

[34] Greg Brockman et al. *OpenAI Gym*. 2016 (S. 51).

[35] Emery D. Berger, Sam Stern und Juan Altmayer Pizzorno. *Triangulating Python Performance Issues with Scalene*. 2022 (S. 57).

[36] Jack Bresenham. „Algorithm for Computer Control of a Digital Plotter“. In: *IBM Syst. J.* 4.1 (1965), S. 25–30. DOI: 10.1147/sj.41.0025 (S. 58).

[37] Takuya Akiba et al. „Optuna: A Next-generation Hyperparameter Optimization Framework“. In: *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2019 (S. 71).

[38] Yoshua Bengio, Nicholas Léonard und Aaron Courville. *Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation*. 2013 (S. 74).

[39] Aaron van den Oord, Oriol Vinyals und Koray Kavukcuoglu. *Neural Discrete Representation Learning*. 2018 (S. 74).

Erklärung

Die vorliegende Arbeit habe ich selbstständig ohne Benutzung anderer als der angegebenen Quellen angefertigt. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten Quellen entnommen wurden, sind als solche kenntlich gemacht. Die Arbeit ist in gleicher oder ähnlicher Form oder auszugsweise im Rahmen einer oder anderer Prüfungen noch nicht vorgelegt worden.

Augsburg, den 31. 07. 2023 Marco Tröster