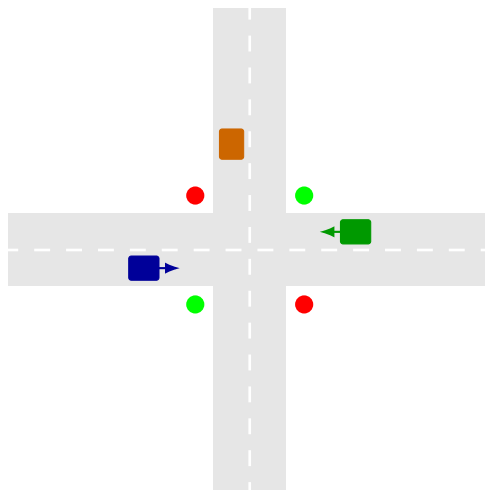


Hochschule Karlsruhe – Technik und Wirtschaft
Fakultät für Informatik und Wirtschaftsinformatik

Multi-Agent Reinforcement Learning für adaptive Ampelsteuerung in SUMO

Bachelorarbeit
im Studiengang Informatik



Vorgelegt von: Sam Weiler
Matrikelnummer: 73640
Studiengang: Informatik

Betreuer: Prof. Dr. rer. nat. Patrick Baier

Karlsruhe, den 21. August 2025

Inhaltsverzeichnis

1	Einleitung	5
1.1	Motivation und Problemstellung	5
1.2	Zielsetzung der Arbeit	5
1.3	Begrenzung des Projektumfangs	6
1.4	Wissenschaftliche und gesellschaftliche Relevanz	7
1.5	Aufbau der Arbeit	7
2	Hintergrund und Stand der Technik	8
2.1	Urbane Verkehrssysteme und Verkehrssteuerung	8
2.2	Simulation urbaner Mobilität mit SUMO	8
2.3	Verstärkendes Lernen (Reinforcement Learning)	9
2.4	SUMO-RL: Architektur und Funktionalität	9
2.4.1	Observation- und Aktionsraum	9
2.4.2	Weitere Eigenschaften	10
2.5	Verwandte Arbeiten	11
3	Datenquellen und Modellierungsgrundlage	11
3.1	OpenStreetMap als Grundlage für das Verkehrsmodell	11
3.2	Verfügbare Verkehrsdaten	14
3.2.1	Öffentliche Datenquellen: LUBW, MobiData BW, Straßenverkehrs- zentrale, BAST	14
3.2.2	Stationäre Zählstellen in Karlsruhe und Umgebung	14
3.2.3	Kommerzielle APIs: TomTom, Google Maps	16
3.3	Modellierung der Ampelschaltungen	16
3.3.1	Verfügbare Daten und Herausforderungen	16
3.3.2	Vereinfachte Modellierung	16
4	Methodik	17
4.1	Untersuchungsregion und Datenbasis	17
4.1.1	Auswahl der Untersuchungsregion	17
4.1.2	Verfügbare Verkehrszähldaten	17
4.2	Aufbau des Simulationsmodells in SUMO	18
4.2.1	Netzgenerierung	18
4.2.2	Erzeugung von Fahrzeugflüssen	18
4.2.3	Identifikation relevanter Zufahrtskanten	18
4.2.4	Automatisierte Generierung von Trips	18
4.2.5	Validierung des Netzmodells	19
4.2.6	Szenarien und Referenzsimulationen	19
4.2.7	Signalsteuerung und Simulationsparameter	19
4.3	Reinforcement-Learning-Konzept	20
4.3.1	Formulierung des RL-Problems	20
4.3.2	Auswahl des RL-Algorithmus	20
4.3.3	Zustände	20
4.3.4	Aktionen	21
4.3.5	Belohnungsfunktionen	21
4.3.6	Hyperparameter-Anpassung	22
4.4	Analyse und Herausforderungen bei der OSM-Netznutzung	22
4.4.1	Grundstruktur von Lichtsignalanlagen in SUMO	22
4.4.2	Typische Fehlerquellen nach OSM-Import	24
4.4.3	Problematik nicht-motorisierter Verkehrswege im OSM-Modell	25

4.4.4	Eingesetzte netconvert -Optionen und deren Grenzen	26
4.4.5	Manuelle Eingriffe und strukturelle Rekonstruktionen	26
4.4.6	Ergebnis: ein realistisches, RL-kompatibles Netz	27
4.5	Netzprüfung, Reparatur und Toolchain	28
4.5.1	Werkzeuge zur Netzprüfung und Reparatur	28
4.5.2	Auswahl eines bereinigten Netzes	32
4.5.3	Vorteil des automatisierten Workflows	32
4.6	Einbindung des SUMO-Netzes in die RL-Umgebung	33
4.6.1	Gesamtsystem und Architektur	33
4.6.2	Agentenzuordnung im Multiagentensystem	35
4.6.3	Vermeidung von Überanpassung	35
4.6.4	Konfiguration der Umgebung	35
4.6.5	Trainingsalgorithmus und Hyperparameter	36
4.6.6	Checkpoints, Monitoring und Logging	37
4.6.7	Systemumgebung und Durchführung	38
4.6.8	Zusammenfassung	40
4.7	Evaluationsstrategie	40
4.7.1	Vergleichsszenarien	40
4.7.2	Bewertete Modelle	40
4.7.3	Reward-Funktionen	40
4.7.4	Simulations- und Umgebungsparameter	41
4.7.5	Ablauf je Episode	41
4.7.6	Seeds und Replikationsdesign	41
4.7.7	Metriken und Logging	42
4.7.8	Reproduzierbarkeit	42
5	Evaluation und Ergebnisse	42
5.1	Reward: Diff-Waiting-Time	42
5.1.1	Mittlere Wartezeiten	43
5.1.2	Anzahl stoppender Fahrzeuge	44
5.1.3	Anzahl ankommender Fahrzeuge	46
5.1.4	Durchschnitt fahrender Fahrzeuge	47
5.1.5	Durchschnittsgeschwindigkeiten	48
5.1.6	Anzahl teleportierender Fahrzeuge	49
5.1.7	Anzahl zurückgehaltener Fahrzeuge	49
5.1.8	Einstufung	50
5.2	Reward: Queue	50
5.2.1	Mittlere Wartezeiten	51
5.2.2	Anzahl stoppender Fahrzeuge	52
5.2.3	Anzahl ankommender Fahrzeuge	54
5.2.4	Durchschnitt fahrender Fahrzeuge	55
5.2.5	Durchschnittsgeschwindigkeiten	56
5.2.6	Anzahl teleportierender Fahrzeuge	57
5.2.7	Anzahl zurückgehaltener Fahrzeuge	57
5.2.8	Einstufung	58
5.3	Reward: Reale Welt	58
5.3.1	Mittlere Wartezeiten	58
5.3.2	Anzahl stoppender Fahrzeuge	59
5.3.3	Anzahl ankommender Fahrzeuge	60
5.3.4	Durchschnitt fahrender Fahrzeuge	61
5.3.5	Durchschnittsgeschwindigkeiten	61

5.3.6	Anzahl teleportierender Fahrzeuge	62
5.3.7	Anzahl zurückgehaltener Fahrzeuge	62
5.3.8	Einstufung	62
5.4	Reward: CO ₂ -Emissionen	62
5.4.1	CO ₂ -Emissionen	63
5.4.2	Mittlere Wartezeiten	64
5.4.3	Anzahl stoppender Fahrzeuge	65
5.4.4	Anzahl ankommender Fahrzeuge	67
5.4.5	Durchschnitt fahrender Fahrzeuge	68
5.4.6	Durchschnittsgeschwindigkeiten	69
5.4.7	Anzahl teleportierender Fahrzeuge	70
5.4.8	Anzahl zurückgehaltener Fahrzeuge	70
5.4.9	Einstufung	71
5.5	Robustheit und Replikationsanalyse	71
5.6	Zusammenfassung	71
6	Herausforderungen und Limitationen	71
6.1	Technische und methodische Hürden	71
6.2	Repräsentativität und Qualität der Daten	71
6.3	Generalisierbarkeit der Ergebnisse	71
7	Fazit und Ausblick	71
7.1	Zusammenfassung der wichtigsten Erkenntnisse	71
7.2	Mögliche Weiterentwicklungen	71
7.3	Relevanz für reale Verkehrsplanung	71
A	Trainings-Skripte	72
A.1	<code>train.py</code> – Trainingsskript für PPO über mehrere Seeds	72
A.2	<code>continuetrain.py</code> – Trainingsskript zum Weitertrainieren	79
B	Belohnungsfunktionen	83
B.1	<code>diff-waiting-time</code>	83
B.2	<code>queue</code>	83
B.3	<code>realworld</code>	83
B.4	<code>emissions</code>	84
C	Evaluierungs-Skripte	85
C.1	<code>evaluate.py</code> – Evaluationsskript für PPO-Modelle und Baselines	85
D	Postprocessing der Evaluationsergebnisse	90
D.1	<code>json2csv.py</code> – Konvertierung und Aggregation von Evaluationsergebnissen	90
E	Netzwerk-Skripte	92
E.1	<code>check_tls_consistency.py</code> – Prüfung inkonsistenter Phasenlängen	92
E.2	<code>check_tls_requests.py</code> – Prüfung ungültiger <code><request></code> -Indizes	93
E.3	<code>fix_requests.py</code> – Automatische Korrektur von Requests und Phasen	93
E.4	<code>repair_net.py</code> – manuelle TLS-Reparatur auf Basis eines Referenz-Dictionaries	95
E.5	<code>statecheck.py</code> – Prüfung auf Ziel-Phasenlänge	96
E.6	<code>find_valid_tls.py</code> – Validierung lauffähiger TLS für SUMO-RL	96
E.7	<code>find_relevant_edges.py</code> – Suche alle relevanten edges	97

F Sumo-Konfiguration	98
F.1 sumoconfig_.sumocfg	98

1 Einleitung

1.1 Motivation und Problemstellung

Städte stehen zunehmend vor der Herausforderung, mit den wachsenden Anforderungen des urbanen Verkehrs zurechtzukommen. Die Zahl der Fahrzeuge im Individualverkehr steigt kontinuierlich [63, 20], was zu einer Verdichtung des Verkehrsaufkommens, insbesondere in städtischen Knotenpunkten, führt. Die daraus resultierenden Konsequenzen sind vielfältig: Verkehrsüberlastungen führen zu erhöhten Reisezeiten, steigenden Emissionen und einer verminderten Lebensqualität für die Bevölkerung. [11] Darüber hinaus verursacht ineffizienter Verkehr einen erheblichen wirtschaftlichen Schaden durch Zeitverluste und Ressourcenverschwendung. [64, 16]

Ein zentraler Hebel zur Verbesserung dieser Situation liegt in der intelligenten Steuerung des Verkehrsflusses, insbesondere an Kreuzungen, an denen mehrere Verkehrsströme aufeinandertreffen. Die Lichtsignalanlagen, die dort zum Einsatz kommen, arbeiten vielerorts noch nach starren, zeitbasierten Schaltplänen, die selten in Echtzeit auf veränderte Verkehrssituationen reagieren. [5] Auch adaptive Verfahren, wie verkehrsabhängige Steuerungen mittels Induktionsschleifen oder Kameras, sind in ihrer Reaktionsfähigkeit beschränkt. Damit bleibt ein enormes Potenzial zur Effizienzsteigerung ungenutzt. [6]

Vor diesem Hintergrund bietet die Kombination moderner Simulationstechniken mit Methoden der künstlichen Intelligenz, insbesondere dem *Reinforcement Learning*, eine vielversprechende Alternative. Reinforcement Learning (RL) ist ein lernbasiertes Verfahren, bei dem ein Agent durch Interaktion mit einer Umgebung eine optimale Strategie zur Maximierung eines definierten Belohnungskriteriums erlernt. Die Anwendung dieses Konzepts auf Ampelsteuerungen erlaubt es, reaktive, datengestützte Systeme zu entwickeln, die dynamisch auf die aktuelle Verkehrssituation reagieren und dabei auf langfristige Effizienz optimiert sind.

Zur Erprobung solcher Verfahren eignet sich die Verkehrssimulationsumgebung *SUMO* (Simulation of Urban MObility) [47], eine quelloffene, modular aufgebaute Plattform, die es ermöglicht, Verkehrsflüsse realitätsnah zu modellieren und zu analysieren. In Kombination mit dem Framework *sumo-r1* [55], das eine Brücke zwischen SUMO und gängigen Machine-Learning-Frameworks wie TensorFlow oder PyTorch schlägt, lassen sich Reinforcement-Learning-Agenten direkt in die Simulationsumgebung einbetten. Diese können dann die Steuerung einzelner Ampelanlagen übernehmen und ihre Strategien durch wiederholte Simulation iterativ verbessern.

1.2 Zielsetzung der Arbeit

Ziel dieser Bachelorarbeit ist es, eine auf Reinforcement Learning basierende Steuerung von Ampelanlagen innerhalb eines realitätsnahen, simulierten städtischen Verkehrsnetzes zu entwickeln, umzusetzen und zu evaluieren. Als Modellregion dient ein ausgewählter, stark befahrener Bereich der Stadt Karlsruhe, dessen Straßennetz mithilfe von OpenStreetMap-Daten und Verkehrsdaten von Institutionen wie LUBW, MobiData BW und der Bundesanstalt für Straßenwesen realitätsnah abgebildet wird. [30, 24, 22]

Die Arbeit verfolgt einen anwendungsorientierten Ansatz: Es wird ein vollständiges System aufgebaut, in dem einzelne Ampelkreuzungen durch RL-Agenten gesteuert werden. Diese erhalten als Eingabe Informationen zur aktuellen Verkehrslage, etwa Fahrzeuganzahl, Wartezeiten oder Stauentwicklungen, und geben als Ausgabe Ampelschaltbefehle zurück. Ziel ist es, durch Training in der Simulation eine Steuerungsstrategie zu entwickeln, die relevante Zielgrößen wie die durchschnittliche Wartezeit, den

Verkehrsfluss oder die Anzahl von Fahrzeugstopps optimiert.

Ein positiver Untersuchungsverlauf könnte zeigen, dass bestehende Straßennetze effizienter genutzt werden können, ohne kostspielige Neubauten oder Erweiterungen. Die verbesserte Auslastung bestehender Infrastruktur spart Kosten, reduziert Flächenversiegelung und mindert Umweltbelastung durch Verkehrsvermeidung. Außerdem wäre ein solches adaptive System klimafreundlicher als starre Ampelsteuerungen.

Darüber hinaus soll die Arbeit systematisch untersuchen, wie sich unterschiedliche Modellierungsentscheidungen (z.B. Wahl der Belohnungsfunktion, Anzahl der gesteuerten Agenten, Parametrisierung der Umgebung) auf das Verhalten und die Leistungsfähigkeit der lernenden Agenten auswirken. Die gewonnenen Erkenntnisse sollen kritisch reflektiert und mit konventionellen, nicht-adaptiven Steuerungsstrategien verglichen werden.

1.3 Begrenzung des Projektumfangs

Trotz des Anspruchs auf Realitätsnähe handelt es sich bei der vorliegenden Arbeit um ein simulationsbasiertes Projekt mit bewusst gewähltem Fokus. Die Umsetzung erfolgt ausschließlich in der Simulationsumgebung SUMO und basiert auf öffentlich zugänglichen Geodaten (OpenStreetMap) sowie begrenzt verfügbaren Verkehrsdaten von staatlichen und kommunalen Institutionen. Eine vollständige Abbildung aller Aspekte des realen Straßenverkehrs ist damit weder angestrebt noch möglich. [47]

Inbesondere ergeben sich folgende Einschränkungen:

- **Eingeschränkte Datenverfügbarkeit:** Nicht alle für eine realitätsnahe Verkehrsmodellierung relevanten Daten liegen in ausreichender Qualität oder Auflösung vor. Exakte Ampelschaltzeiten, Fußgängerfrequenzen oder dynamische Verkehrsdaten zu Stoßzeiten sind teilweise nicht öffentlich zugänglich oder nur unvollständig. Dazu kommt, dass Kommunen teilweise bewusst den Verkehr lenken, etwa durch Zufahrtsbeschränkungen oder Verkehrsberuhigungszonen, was oft nicht öffentlich kommuniziert wird. [65]
- **Vereinfachte Modellierung der Umgebung:** In der Simulation wird angenommen, dass alle Verkehrsteilnehmer (Fahrzeuge, Fußgänger, Radfahrer) durch die Agenten präzise erfasst werden können, eine Annahme, die in der Realität durch technische und datenschutzrechtliche Hürden nicht haltbar ist. Moderne Systeme arbeiten hier mit Datenschutz-mechanismen, aber eine flächendeckende, genaue Erfassung ist unerlässlich, aber derzeit technisch und rechtlich nicht umsetzbar. [8] Dies wird in der Arbeit berücksichtigt, vor allem bei realitätsnahe Modelltraining.
- **Städtebauliche Verkehrslenkung:** In der Realität regeln Städte Verkehrsflüsse z.B. durch Low-Traffic-Neighbourhoods, Zufahrtsbeschränkungen oder geregelte Zuflusssteuerung, um bestimmte Stadtbereiche zu entlasten. [23] Solche Maßnahmen sind jedoch in der Simulationsumgebung nicht dynamisch abbildbar, da nur externe Ampelagenten kontrollieren und keine zonale Steuerungslogik abgebildet wird.
- **Begrenzter räumlicher und zeitlicher Umfang:** Simuliert wird lediglich ein ausgewählter Ausschnitt des Karlsruher Straßennetzes und nur für definierte Zeitabschnitte. Eine vollständige Tag-Nacht-Modellierung liegt außerhalb des Umfangs.

- **Trainings- und Evaluierungsgrenzen:** Reinforcement-Learning-Agenten benötigen viele Trainingszyklen. Die in dieser Arbeit verwendete Hardware limitiert Trainingsdauer und Modellkomplexität.

Diese bewusste Eingrenzung ermöglicht es, sich auf die technische Umsetzbarkeit und das methodische Vorgehen zu konzentrieren. Dennoch sind die gewonnenen Erkenntnisse relevant, sie liefern zentrale Einsichten in die Wirksamkeit von KI-basierten Verkehrssteuerungssystemen und können als Grundlage für weiterführende Forschung dienen.

1.4 Wissenschaftliche und gesellschaftliche Relevanz

Die Kombination von KI und Verkehrssteuerung ist nicht nur ein hochaktuelles Forschungsthema, sondern besitzt auch ein erhebliches Potenzial für den realweltlichen Einsatz. [21] Durch die Integration lernfähiger Steuerungssysteme in bestehende Verkehrsmanagementlösungen könnten Städte künftig dynamischer, effizienter und umweltfreundlicher agieren. Die hier behandelte Arbeit leistet einen Beitrag zur Untersuchung der technischen Machbarkeit sowie der Leistungsfähigkeit solcher Systeme unter realitätsnahen Bedingungen.

Gleichzeitig dient die Arbeit als Beispiel für den Einsatz moderner Methoden der Informatik in einem interdisziplinären Anwendungsfeld. Sie schlägt die Brücke zwischen Verkehrsingenieurwesen, Datenanalyse und maschinellem Lernen und eröffnet damit Perspektiven für eine zukunftsweisende Gestaltung urbaner Infrastrukturen.

1.5 Aufbau der Arbeit

Die Arbeit ist in sieben Kapitel unterteilt:

- Kapitel 2 stellt die theoretischen Grundlagen der Arbeit dar. Es werden die Funktionsweise von SUMO, die Prinzipien des Reinforcement Learning sowie die zugrundeliegenden technischen Komponenten erläutert. Auch verwandte Arbeiten werden kritisch betrachtet.
- Kapitel 3 widmet sich den Datenquellen und der Modellierungsgrundlage. Es werden sowohl die verwendeten Geodaten als auch Verkehrszählungen, Ampelschaltpläne und Annahmen beschrieben.
- Kapitel 4 beschreibt die methodische Vorgehensweise bei der Erstellung des Simulationsmodells, der Formulierung des Lernproblems, der Wahl der Trainingsstrategie und der technischen Umsetzung.
- Kapitel 5 präsentiert die Ergebnisse der Simulationen und stellt sie in Bezug zur gewählten Zielsetzung. Es erfolgt eine quantitative und qualitative Auswertung der Agentenleistung.
- Kapitel 6 diskutiert zentrale Herausforderungen und Limitationen der Arbeit, sowohl methodisch als auch datenbezogen.
- Kapitel 7 fasst die wesentlichen Erkenntnisse zusammen und gibt einen Ausblick auf weiterführende Forschungsansätze und Anwendungsoptionen.

2 Hintergrund und Stand der Technik

2.1 Urbane Verkehrssysteme und Verkehrssteuerung

Die urbane Verkehrssteuerung umfasst alle Maßnahmen zur Regelung, Lenkung und Optimierung von Verkehrsflüssen innerhalb städtischer Räume. Ziel ist es, den Verkehrsfluss effizient zu gestalten, Staus zu vermeiden, die Sicherheit aller Verkehrsteilnehmer zu erhöhen sowie Emissionen und Lärm zu reduzieren. Klassische Steuerungsmechanismen basieren häufig auf festen Zeitplänen oder einfachen verkehrsabhängigen Regeln, z. B. durch Induktionsschleifen oder Detektoren gesteuerte Ampelphasen. [62]

Mit dem Aufkommen neuer Technologien und wachsender Mobilitätsdaten entstehen zunehmend datenbasierte und dynamische Steuerungsansätze. Dazu gehören adaptive Lichtsignalsteuerungen, vernetzte Fahrzeuge (V2X-Kommunikation) und erste Pilotprojekte mit KI-gesteuerten Verkehrsmanagementsystemen. [19, 57] Dennoch sind viele Systeme in der Praxis noch unflexibel oder schwer skalierbar. [18, 68]

2.2 Simulation urbaner Mobilität mit SUMO

Simulation of Urban MObility (SUMO) ist ein quelloffener, Verkehrs-Simulator, der ursprünglich vom Deutschen Zentrum für Luft- und Raumfahrt (DLR) entwickelt wurde. [47] SUMO erlaubt die detaillierte Modellierung individueller Fahrzeuge, Straßeninfrastruktur, Ampelschaltungen sowie Fahrverhalten. [47, 49, 52]

Besonders relevant für diese Arbeit sind folgende Merkmale[47]:

- **Mikroskopische Modellierung:** Jedes Fahrzeug wird als individuelles Objekt simuliert. Parameter wie Geschwindigkeit, Abstand oder Spurwechselverhalten sind individuell konfigurierbar.
- **Flexible Netzdefinition:** Verkehrsnetze lassen sich aus OpenStreetMap-Daten sowie aus Shapefiles oder VISUM-Modellen mit dem Tool `netconvert` erzeugen. [27] Netzdateien können auch mit `netedit` visuell editiert werden.
- **Nachfragegenerierung:** Fahrpläne und Routen lassen sich mit Tools wie `activitygen`, `randomTrips`, `od2trips` oder `duarouter` erzeugen, basierend auf statistischen oder echten OD-Matrizen. [39, 10]
- **Multimodalität:** SUMO unterstützt neben Pkw auch Busse, Fahrräder, Fußgänger sowie den öffentlichen Nahverkehr. Ampeln können für alle Verkehrsarten gleichzeitig modelliert werden.
- **Emissionsmodellierung:** Mit Hilfe von integrierten HBEFA-Tabellen (Version 4)[51] kann SUMO CO₂-, NO_x- und Feinstaubemissionen simulieren und ausgeben.
- **Steuerbare Ampelanlagen:** Lichtsignalanlagen können sowohl mit festen Programmen als auch dynamisch über die TraCI-Schnittstelle gesteuert werden.
- **Reproduzierbarkeit und Kontrolle:** SUMO ist vollständig deterministisch, was es ideal für kontrollierte Experimente und das Training von KI-Agenten macht.
- **Visualisierung und Debugging:** Die SUMO-GUI und das Tool `sumo-gui` ermöglichen eine grafische Darstellung von Netz, Fahrzeugen, Ampelphasen und Simulationsergebnissen.

Ein zentrales Element für externe Steuerungsexperimente ist das **Traffic Control Interface (TraCI)**[47]. Dabei handelt es sich um eine TCP-basierte Client-Server-Schnittstelle, über die laufende SUMO-Simulationen in Echtzeit beeinflusst werden können. TraCI erlaubt nicht nur das Abfragen von Zustandsgrößen wie Position, Geschwindigkeit oder Warteschlangenlänge von Fahrzeugen, sondern auch das aktive Eingreifen, etwa durch Phasenwechsel an Lichtsignalanlagen oder die Modifikation von Routen. Dadurch wird SUMO zu einer interaktiven Umgebung, die sich ideal für Reinforcement-Learning-Anwendungen eignet, da Agenten ihre Aktionen direkt auf den Verkehrsfluss auswirken können und unmittelbares Feedback erhalten. [61]

Die SUMO-Toolchain bietet damit alle notwendigen Komponenten für die Entwicklung, Analyse und Auswertung urbaner Verkehrsszenarien und stellt eine erprobte Plattform für KI-gestützte Steuerungsexperimente dar.

2.3 Verstärkendes Lernen (Reinforcement Learning)

Reinforcement Learning (RL) ist ein Teilgebiet des maschinellen Lernens, bei dem ein Agent durch Interaktion mit einer Umgebung lernt, optimale Handlungen auszuführen. Ziel ist es, die kumulative Belohnung über Zeit zu maximieren [58].

Ein RL-Prozess wird typischerweise als Markov Decision Process (MDP) beschrieben und besteht aus folgenden Komponenten:

- **Zustand s (state):** Repräsentation der aktuellen Situation der Umgebung.
- **Aktion a (action):** Entscheidung oder Handlung, die der Agent im Zustand s trifft.
- **Belohnung r (reward):** Numerischer Wert, der die Güte der Aktion bewertet.
- **Policy π :** Strategie, die angibt, welche Aktion in welchem Zustand gewählt wird.

Der Agent interagiert wiederholt mit der Umgebung, beobachtet Zustände, wählt Aktionen, erhält Belohnungen und gelangt in neue Zustände. Durch diese Rückkopplung lernt er, langfristig optimale Entscheidungen zu treffen.

In dieser Arbeit wird ausschließlich **Proximal Policy Optimization (PPO)** [36] eingesetzt – ein policy-basierter RL-Algorithmus, der für seine Stabilität und Effizienz bei kontinuierlichen Interaktionen mit komplexen Umgebungen bekannt ist. Andere RL-Verfahren (z. B. Q-Learning oder DQN) wurden nicht berücksichtigt.

2.4 SUMO-RL: Architektur und Funktionalität

`sumo-rl` ist ein Python-Framework, das die Verkehrssimulation mit SUMO und Reinforcement Learning kombiniert. Es bietet eine stabile Schnittstelle zu Gymnasium und PettingZoo, wodurch RL-Agenten problemlos in SUMO-Simulationen integriert werden können. Die zentrale Implementierung bildet die Klasse `SumoEnvironment`, die sowohl Single-Agent- als auch Multi-Agent-Setups unterstützt. Über Parameter wie `use_gui`, `num_seconds`, `min_green`, `yellow_time`, `delta_time` usw. lässt sich das Verhalten der Umgebung sehr flexibel steuern. [53]

2.4.1 Observation- und Aktionsraum

Beobachtung (Observation) Die Standard-Beobachtung für jeden Ampelagenten ist ein Vektor: [53]

obs = [phase_one_hot, min_green, lane₁_density, ..., lane_n_density, lane₁_queue, ..., lane_n_queue]

Diese Komponenten bedeuten im Detail:

- **phase_one_hot**: One-Hot-Kodierung der aktuell aktiven Grünphase.
- **min_green**: Binärwert, der anzeigt, ob die Minstdauer (**min_green**) der aktuellen Phase bereits erreicht wurde.
- **lane_i_density**: Verhältnis der Fahrzeuge auf Spur *i* zur maximal möglichen Kapazität der Spur.
- **lane_i_queue**: Anteil an Fahrzeugen mit Geschwindigkeit $< 0,1$ m/s auf Spur *i* im Verhältnis zur Kapazität.

Falls eine eigene Beobachtungsfunktion gewünscht ist, kann man eine Klasse von `ObservationFunction` ableiten und beim Initialisieren der Umgebung übergeben.

Aktionsraum (Action Space) Der Aktionsraum ist diskret. Agents dürfen alle `delta_time` Sekunden eine neue Grünphase wählen. Jeder Aktionswert repräsentiert dabei eine bestimmte, zulässige Grünkonfiguration für die Kreuzung. Ein Phasenwechsel wird stets durch eine automatisch hinzugefügte Gelbphase (`yellow_time` Sekunden) eingeleitet.

2.4.2 Weitere Eigenschaften

- **TraCI-Integration**: Echtzeit-Kommunikation mit SUMO, u. a. zum Abruf von Zustandsdaten und Auslösen von Phasenwechseln.
- **Multi-Agent-Fähigkeit**: Unterstützt Einzel- und Mehragentenumgebungen; bei Verwendung von `parallel_env` über PettingZoo werden alle Agenten synchron gesteuert.
- **Systemmetriken (Infos)**: Die Umgebung kann global relevante Kennzahlen wie Wartezeit, Gesamtemissionen, Backlog etc. über das `infos`-Objekt bereitstellen – steuerbar über Parameter wie `add_system_info` oder `add_per_agent_info`.
- **Kompatibilität mit RL-Bibliotheken**: Nativ kompatibel mit Stable-Baselines3, PyTorch, TensorFlow, RLlib etc., was eine einfache Einbettung in bestehende RL-Pipelines erlaubt.
- **Konfigurationsmöglichkeiten**: Belohnungsfunktionen (`reward_fn`), Beobachtungsfunktion (`observation_class`) und diverse Simulationsparameter können flexibel angepasst werden.

[53]

Mit diesen Eigenschaften bietet `sumo-rl` eine robuste Grundlage zur Integration von Reinforcement Learning in SUMO-basierte Verkehrssteuerungssysteme und bildet das methodische Rückgrat dieser Arbeit.

2.5 Verwandte Arbeiten

In den letzten Jahren wurden zahlreiche wissenschaftliche Arbeiten veröffentlicht, die KI-basierte Methoden zur Optimierung der Verkehrslichtsteuerung mithilfe der SUMO-RL-Bibliothek vorstellen. Im Folgenden eine Auswahl repräsentativer Ansätze, die sich durch innovative RL-Methoden und vielfältige Anwendungsfelder auszeichnen:

- **Alegre et al. (2021):** Untersuchung des Einflusses von Nicht-Stationarität auf lernbasierte Ampelsteuerungsstrategien. Die Arbeit beleuchtet, wie sich veränderte Verkehrsbedingungen auf die Stabilität und Leistung von Reinforcement-Learning-Agenten auswirken. [2]
- **Hwang et al. (2023):** Entwicklung eines informations-theoretischen Zustandsraummodells für Multi-View Reinforcement Learning. Dieser Ansatz zielt auf die verbesserte Integration heterogener Sensordaten zur präziseren Verkehrssteuerung. [15]
- **Reza et al. (2023):** Einsatz einer TD-Learning-basierten Methode zur umfassenden stadtweiten Ampelregelung unter besonderem Fokus auf autonome Fahrzeuge. Die Performance wird mittels SUMO-Simulationen evaluiert. [41]
- **Almeida et al. (2022):** Kombination von Multiagenten-Reinforcement Learning mit k-Nearest-Neighbors-Technik zur koordinierten Steuerung von Ampeln in Netzwerken mit dichtem Verkehrsaufkommen. [3]
- **Zheng et al. (2022):** Vorschlag eines Curriculum-Learning-Ansatzes, der schrittweise von lokalen zu globalen Steuerstrategien führt, mit dem Ziel, RL-Agenten robuster und lernfähiger zu machen. [70]
- **Weitere erwähnenswerte Arbeiten:** Benchmarking verschiedener RL-Algorithmen für Ampelsteuerung (Ault & Sharon, 2021), ein Framework zur Belohnungsgestaltung („EcoLight“, Agand et al., 2021) sowie ontologiegestützte Lernmodelle, die eine robuste Anwendung von Steuerungsstrategien in unbekannten Situationen ermöglichen (Ghanadbashi et al., 2022). [4, 1, 12]

Diese Arbeiten zeigen, dass RL-basierte Methoden das Potenzial haben, bestehende Systeme zu übertreffen, sowohl bei einfachen als auch bei komplexeren Szenarien. Die vorliegende Arbeit knüpft an diesen Forschungsstand an und erweitert ihn um eine Anwendung auf reale Geodaten aus Karlsruhe sowie eine methodische Evaluation.

3 Datenquellen und Modellierungsgrundlage

3.1 OpenStreetMap als Grundlage für das Verkehrsmodell

Das Verkehrsnetz für die Simulation basiert auf öffentlich verfügbaren Geodaten der Plattform OpenStreetMap (OSM). OSM bietet eine frei zugängliche, kollaborativ gepflegte Datenbank, die detaillierte Informationen zu Straßenverläufen, Kreuzungen, Fahrspuren, Tempolimits und teilweise zu Ampelanlagen enthält. Diese Eigenschaften machen OSM zu einer geeigneten Grundlage für Verkehrssimulationen mit SUMO. [30, 33]

Zur Erstellung des Netzes wurde ein Ausschnitt des Straßennetzes der Stadt Karlsruhe exportiert, der einen stark frequentierten urbanen Bereich mit mehreren signalgesteuerten Kreuzungen umfasst. Der betrachtete Bereich liegt zwischen 49,00738,°N und

49,01523,°N sowie 8,38589,°E und 8,40050,°E und deckt unter anderem die Reinhold-Frank-Straße, das Mühlburger Tor und angrenzende Hauptverkehrsachsen ab. Der Export erfolgte als `.osm`-Datei über den Geofabrik-Downloaddienst bzw. mit dem Tool JOSM. [31, 17] Die anschließende Konvertierung in das SUMO-Format erfolgte mit dem Programm `netconvert` (Version 1.19.0), einem Teil der SUMO-Toolchain. [48] Hierbei wurden relevante Parameter wie Straßentypen, Fahrspuren, Prioritäten und erlaubte Abbiegevorgänge berücksichtigt. Als Typemap kam `osmNetconvert.typ.xml` [27] zum Einsatz, um realitätsnahe Geschwindigkeiten und Fahrspuren zuzuweisen.

Das resultierende Verkehrsnetz umfasst 1.379 definierte Knotenpunkte (*junctions*), 1.919 Straßenkanten (*edges*) sowie insgesamt 5.310 modellierte Fahrstreifen (*lanes*). [28] Darüber hinaus konnten 17 signalgesteuerte Kreuzungen mit Lichtsignalanlagen (*traffic lights*) identifiziert (siehe Algorithmus 6) werden, die als Steuerungspunkte für das spätere Training der Reinforcement-Learning-Agenten dienen.

Zusätzliche Informationen wie Ampeldefinitionen und Vorfahrtsregeln manuell über das Tool `netedit` [28] ergänzt oder angepasst, um die Netzrealität weiter zu verfeinern. Dabei wurden insbesondere fehlerhafte Knotenbeziehungen bereinigt sowie isolierte Netzteile entfernt. Die finale `.net.xml`-Datei bildet die topologische und funktionale Grundlage für alle weiteren Simulationsschritte.



Abbildung 1: Visualisierung des aus OSM generierten SUMO-Netzes (sumo-gui [50]).

Fazit: Insgesamt erlaubt OSM trotz dieser Limitationen den Aufbau eines funktionalen Verkehrsnetzes für mikroskopische Simulationen, sofern der Import sorgfältig konfiguriert und die resultierenden Daten kritisch hinterfragt und gegebenenfalls manuell nachbearbeitet werden.

3.2 Verfügbare Verkehrsdaten

Zur Kalibrierung und Validierung der Simulation sind verlässliche Verkehrsdaten unerlässlich. In Baden-Württemberg stehen hierfür mehrere öffentliche sowie kommerzielle Quellen zur Verfügung. Diese umfassen Informationen über Verkehrsstärken, Fahrzeugzusammensetzung, Reisezeiten und Störungen im Straßenverkehr. Im Folgenden werden die wichtigsten Quellen sowie die für das vorliegende Projekt relevanten Verkehrszählungen zusammengefasst.

3.2.1 Öffentliche Datenquellen: LUBW, MobiData BW, Straßenverkehrszentrale, BAST

Die Landesanstalt für Umwelt Baden-Württemberg (LUBW) stellt aggregierte Verkehrszählungen im Rahmen automatischer Straßenverkehrszählungen bereit. Diese umfassen Tagesmittelwerte sowie jahreszeitliche Schwankungen für verschiedene Fahrzeugkategorien. Die Daten der Straßenverkehrszentrale Baden-Württemberg (SVZ-BW) liefern zudem Echtzeitinformationen zu Störungen, Baustellen und Verkehrsfluss.[69, 7, 22, 5, 44]

Über die Plattform MobiData BW werden offene Mobilitätsdaten gebündelt bereitgestellt, darunter auch historische Detektordaten und OpenTraffic-Feeds. Die Bundesanstalt für Straßenwesen (BAST) wiederum veröffentlicht bundesweite Zählungen, insbesondere für überörtliche Straßen. [7]

Diese öffentlichen Quellen bilden eine solide Grundlage für die realitätsnahe Modellierung des Verkehrsaufkommens, sind jedoch teilweise nur in aggregierter Form oder mit begrenzter räumlicher Auflösung verfügbar.

3.2.2 Stationäre Zählstellen in Karlsruhe und Umgebung

Eine besonders wertvolle Datenquelle zur realitätsnahen Modellierung des Verkehrsaufkommens stellen die stationären Zählstellen des Landes Baden-Württemberg dar. Diese liefern standardisierte Tagesverkehrswerte (DTV¹), getrennt nach Fahrzeugklassen.

Im direkten Untersuchungsgebiet, der Reinhold-Frank-Straße in Karlsruhe, befindet sich eine automatische Dauerzählstelle. Die dort erfassten Werte für den Zeitraum vom 1.1. bis 20.6.2025 lauten: [69]

- **Kraftfahrzeuge (KFZ):** 21.300 Fahrzeuge/Tag
- **Personenkraftwagen (PKW):** 20.500 Fahrzeuge/Tag
- **Schwere Nutzfahrzeuge (sNfz):** 120 Fahrzeuge/Tag

Diese Messwerte stimmen gut mit den aus den äußeren Zufahrtsachsen abgeleiteten Schätzungen überein. Um das Verkehrsaufkommen plausibel zu quantifizieren, wurden zusätzlich acht zentrale Zählstellen aus dem Jahr 2023 entlang wichtiger Ein- und

¹DTV steht für *Durchschnittlicher Tagesverkehr* und bezeichnet die mittlere Anzahl an Fahrzeugen, die einen bestimmten Straßenabschnitt pro Tag passieren, typischerweise gemittelt über einen längeren Zeitraum.

Ausfallstraßen berücksichtigt. Sie bilden die Grundlage für die Annahmen über den täglichen Verkehr, der potenziell durch das untersuchte innerstädtische Netz fließt:

Tabelle 1: Verkehrszählungen in und um Karlsruhe (DTV, Jahr 2023) [9]

Zufahrt	Zählstellenbeschreibung	KFZ/Tag	SV/Tag	Gesamt
B10 West	Rheinbrücke / Entenfang	62.102	6.159	68.261
B36 Neureut	Neureuter Str. / Ausfahrt Neureut Süd	35.165	1.712	36.877
B36 Nord	Eggenstein / Neureut	28.595	1.361	29.956
L605 Nord	Weißes Haus / Eggenstein	14.563	220	14.783
B36 Süd	Rheinstetten / Innenstadt	24.239	1.487	25.726
B36 Mörsch	Mörsch / Forchheim	26.841	1.531	28.372
L605 Süd	Ettlingen / Bulacher Kreuz	65.816	3.474	69.290
B10 Ost	Durlach (A5) / Innenstadt	28.555	913	29.468

Hinweis: KFZ = Leichtverkehr (Pkw, Lieferwagen, Motorräder); SV = Schwerverkehr (Lkw, Busse, schwere Nutzfahrzeuge); Gesamt = Summe aus KFZ und SV.



Legende: ● Temporäre Zählstellen ● Dauerzählstellen ● Manuelle Zählstellen

Abbildung 3: Lage der Dauerzählstellen im Raum Karlsruhe (Quelle: MobiData BW [25]).

Diese externen Zuflüsse bilden die Grundlage für realistische Eingangsströme in der Simulation. Sie versorgen das Untersuchungsgebiet direkt und ergeben ein plausibles Verkehrsaufkommen von etwa 20.000 bis 40.000 Fahrzeugen pro Tag, was mit den Messungen in der Reinhold-Frank-Straße übereinstimmt.

Die Zähldaten erlauben es, die Fahrzeugströme in SUMO proportional zu den realen Verhältnissen abzubilden und unterstützen zugleich die spätere Kalibrierung und Validierung der Szenarien.

3.2.3 Kommerzielle APIs: TomTom, Google Maps

[13, 60]

Ergänzend zu den öffentlichen Datenquellen bieten kommerzielle Anbieter wie TomTom und Google über Programmierschnittstellen (APIs) hochaufgelöste Echtzeit- und Historikdaten an. Diese umfassen unter anderem:

- Durchschnittliche Fahrgeschwindigkeiten nach Wochentag und Uhrzeit,
- Verkehrsdichte und Stauinformationen,
- Prognosen basierend auf anonymisierten Bewegungsdaten.

Der Zugriff auf diese APIs ist in der Regel kostenpflichtig oder durch Nutzungsbeschränkungen limitiert. Sie ermöglichen eine deutlich feinere zeitliche und räumliche Auflösung, was für die Modellierung und spätere Optimierung des Verkehrsflusses mittels KI von Vorteil sein könnte.

Für die vorliegende Arbeit wurden diese kommerziellen Angebote nicht genutzt. Die Modellierung basiert ausschließlich auf offenen Datenquellen wie OSM sowie auf Google Maps für einzelne Standortrecherchen.

3.3 Modellierung der Ampelschaltungen

Für eine realitätsnahe Simulation spielt die Modellierung der Lichtsignalsteuerung eine zentrale Rolle. Ampelanlagen beeinflussen maßgeblich den Verkehrsfluss an Knotenpunkten und sind daher ein zentraler Bestandteil der Simulationslogik. [46]

3.3.1 Verfügbare Daten und Herausforderungen

In den öffentlich zugänglichen OSM-Daten sind Ampelanlagen in der Regel lediglich als Punktobjekte an Kreuzungen vermerkt. Informationen zu Phasenplänen, Umlaufzeiten oder koordinierter Schaltung fehlen vollständig. Auch von Seiten der Stadt Karlsruhe oder anderer kommunaler Stellen liegen keine detaillierten Steuerungsdaten vor, da diese in der Regel nicht öffentlich zugänglich sind. [45]

Eine eigene systematische Erfassung der Schaltzeiten wäre zwar prinzipiell möglich, hätte jedoch einen erheblichen Zeitaufwand bedeutet und wäre aufgrund der dynamischen, nicht-statischen Signalsteuerungen (z. B. verkehrsabhängige Phasen) methodisch schwer zuverlässig umzusetzen gewesen.

3.3.2 Vereinfachte Modellierung

Aus diesen Gründen wurde anfangs auf eine synthetische Modellierung zurückgegriffen. Mittels netgenerate [29] wurde ein synthetisches Netz generiert und abenfalls testweise Modelle trainiert. Dies erwies sich als sehr simpel, wegen geringer Komplexität. SUMO bietet hierfür die Möglichkeit, sogenannte `tlLogic`-Blöcke manuell oder automatisch zu definieren, die verschiedene Phasenfolgen und Zeitparameter enthalten. In der vorliegenden Arbeit wurde auf Standardampelprogramme zurückgegriffen, wie sie in SUMO generisch verwendet werden, um testweise eine vereinfachte Lichtsignalsteuerung zu modellieren. Diese erlaubt die spätere Umsetzung des realen karlsruher Netzes. [46]

4 Methodik

4.1 Untersuchungsregion und Datenbasis

4.1.1 Auswahl der Untersuchungsregion

Für die Anwendung und Evaluation der KI-basierten Verkehrssteuerung wurde ein Ausschnitt des innerstädtischen Straßennetzes von Karlsruhe gewählt. Die Auswahl fiel auf ein [Gebiet](#) rund um die Reinhold-Frank-Straße und das Mühlburger Tor, das durch hohe Verkehrsdichte, komplexe Knotenpunkte und mehrere signalgesteuerte Kreuzungen gekennzeichnet ist. Der gewählte Bereich liegt geografisch zwischen 49,006947°N und 49,015602°N sowie 8,380176°E und 8,403887°E und deckt mehrere stark frequentierte Hauptachsen ab.

Die Entscheidung für diese Region basiert auf folgenden Kriterien (siehe Kapitel 3):

- **Hohe Verkehrsbedeutung:** Das Gebiet stellt einen wichtigen innerstädtischen Verkehrsraum dar.
- **Bekanntes Stauaufkommen:** Die Reinhold-Frank-Straße ist in der Stadtbevölkerung für regelmäßige Verkehrsstaus bekannt, insbesondere zu Stoßzeiten. [\[43\]](#)
- **Verfügbarkeit realer Verkehrszählzeiten:** Eine automatische Dauerzählstelle erhebt dort täglich Verkehrsdaten. Für den Zeitraum vom 01.01.2025 bis 20.6.2025 wurden durchschnittlich 21.300 Kfz/Tag erfasst. [\[69, 9\]](#)
- **Zusätzliche Zählzeiten angrenzender Hauptverkehrsstraßen:** Zählstellen an der B10, B36, L605 und in Durlach liefern ergänzende Werte zur Plausibilisierung des Gesamtverkehrsflusses.
- **Vorhandensein mehrerer Ampelanlagen:** Im Netz befinden sich 17 signalgesteuerte Kreuzungen, geeignet für RL-gesteuerte Steuerungsexperimente (siehe Kapitel 6).
- **Gute Abgrenzbarkeit:** Das Gebiet ist topologisch geschlossen und in SUMO sauber simulierbar.
- **Verfügbarkeit von Geodaten:** Die Region ist in OpenStreetMap detailliert kartiert. [\[31\]](#)

4.1.2 Verfügbare Verkehrszählzeiten

Für die Kalibrierung und Validierung der Verkehrssimulation wurden verschiedene reale Zählzeitenquellen aus dem Raum Karlsruhe herangezogen. Hauptquelle war dabei die offene Mobilitätsdatenplattform des Landes Baden-Württemberg [\[26\]](#). Dort werden automatisiert erfasste Stundenwerte stationärer Dauerzählstellen veröffentlicht, die eine fein aufgelöste Analyse von Verkehrsverläufen ermöglichen.

Konkret wurden folgende Datensätze ausgewertet:

- **Dauerzählstelle Reinhold-Frank-Straße:** Erfasst täglich die Anzahl der Kraftfahrzeuge (Kfz), aufgeschlüsselt nach Fahrzeugklassen (KFZ, PKW, sNfz). Für den Zeitraum 01.01.–20.06.2025 lag der durchschnittliche Tagesverkehr bei ca. 21.300 Kfz/Tag.

- **Historische Jahresmittelwerte:** Langzeitdatenreihen von 2008-2024 aus Mobi-Data BW ermöglichen eine Kontextualisierung der aktuellen Verkehrsbelastung. [25]
- **Zählstellen an äußeren Zufahrtsachsen:** Ergänzende Zählraten aus dem Jahr 2023 an acht stark befahrenen Einfallstraßen (u. a. B10, B36, L605) [25] liefern Anhaltspunkte zur Verkehrsstärke an den Netzzändern.

Die Kombination dieser Quellen ermöglicht eine robuste, datenbasierte Schätzung realistischer Flussverteilungen für die Simulation, sowohl zeitlich (z. B. Spitzenlasten) als auch räumlich (Zufahrtsverteilung).

4.2 Aufbau des Simulationsmodells in SUMO

4.2.1 Netzgenerierung

Zur Modellierung des realen Straßennetzes wurde ein Kartenausschnitt des Untersuchungsgebiets über die Exportfunktion von OpenStreetMap [31] heruntergeladen und anschließend mit JOSM [17] bereinigt. Der Ausschnitt umfasst die Reinhold-Frank-Straße sowie angrenzende Hauptverkehrsachsen im Bereich des Mühlburger Tors. Der bearbeitete Kartenausschnitt wurde mit dem SUMO-Werkzeug `netconvert` [48] in ein XML-Netzwerkformat überführt. Dabei kamen Optionen zur Einsatz um das Netz stabiler zu machen (z. B. `-junctions.join`). [47]

4.2.2 Erzeugung von Fahrzeugflüssen

Für die Simulation wurden zwei Ansätze genutzt. Zunächst entstanden mit dem SUMO-Skript `randomTrips.py` [39] einfache Testflüsse, die zur Validierung des Netzmodells dienten. Anschließend wurden auf Basis realer Verkehrszählraten (siehe Kapitel 3.2.2) Flussprofile definiert, welche die beobachteten DTV-Werte proportional auf die äußeren Zufahrtskanten verteilten. Hauptachsen wie B10 oder B36 erhielten dabei ein höheres Gewicht. Die daraus erzeugten Routendateien wurden mit `duarouter` [10] zu konfliktfreien Fahrten verarbeitet. Unterschiedliche Szenarien (niedrige Last, normale Last, hohe Last) erlaubten die Abbildung verschiedener Verkehrslagen.

4.2.3 Identifikation relevanter Zufahrtskanten

Die Auswahl geeigneter Zufahrtskanten basierte auf den äußeren Hauptverkehrsachsen (u. a. B10, B36, L605, Durlacher Allee). Hierzu wurde ein [Python-Skript](#) eingesetzt, das Kanten mit einem `name`-Attribut automatisch durchsucht und auf relevante Straßennamen prüfte (z. B. "B10", "Reinhold-Frank-Straße"). Die Ergebnisse wurden manuell überprüft und in `netedit` [28] ergänzt. Die so extrahierten Kanten wurden je Verkehrsachse gruppiert und dienten als Grundlage für die segmentierte Trip-Erzeugung.

4.2.4 Automatisierte Generierung von Trips

Die DTV-Werte wurden auf die Zufahrtsgruppen skaliert und auf eine Simulationsdauer von 5000s verteilt. Ein eigens entwickeltes Python-Skript erzeugte daraus Fahrzeugeinträge (`<trip>`), die über die ermittelten Zufahrtskanten ins Netz eingespeist wurden. Die Trips wurden im XML-Format gespeichert und mit `duarouter` in vollständige, konfliktfreie Routen (`<route>`) überführt. Dabei wurde die Gesamtanzahl stündlich extrapoliert und stark belastete Achsen (z. B. B10, L605) mit entsprechend höherem Anteil berücksichtigt. Die erzeugten Flüsse wurden durch Detektor-Ausgaben (u. a. `laneAreaDetector`) [47] überprüft und bei Bedarf angepasst.

4.2.5 Validierung des Netzmodells

Vor dem Einsatz des Modells erfolgte eine mehrstufige Validierung:

- **Netzprüfung:** Einsatz von `netconvert -check-lane-geometry` und `netcheck` zur Überprüfung der topologischen Konsistenz. [27]
- **Visuelle Kontrolle:** Inspektion kritischer Knoten in der SUMO-GUI und in `netedit`, um Fehler wie unverbundene Spuren oder falsche Richtungen zu erkennen. [28]
- **TLS-Prüfung:** Alle 17 Lichtsignalanlagen wurden in `netedit` geöffnet. Phasenpläne, gesteuerte Verbindungen und Zustandslängen (`state`) wurden geprüft und bei Bedarf korrigiert.

Die manuelle Nachbearbeitung war zeitaufwändig, da fehlerhafte TLS nicht automatisch erkannt werden. In mehreren Fällen mussten Phasenpläne neu erstellt oder angepasst werden.

4.2.6 Szenarien und Referenzsimulationen

Zur Validierung der Verkehrsflüsse wurden Szenarien mit unterschiedlicher Lastintensität definiert:

- **Niedrige Last:** geringes Verkehrsaufkommen mit überwiegend freiem Fluss,
- **Normale Last:** repräsentatives Durchschnittsaufkommen im Tagesverlauf,
- **Hohe Last:** stark verdichteter Verkehr zur Abbildung von Spitzenbelastungen.

Diese Szenarien ermöglichen eine Überprüfung der Netzdurchlässigkeit und der Leistungsfähigkeit der Knotenpunkte unter verschiedenen Belastungsniveaus.

4.2.7 Signalsteuerung und Simulationsparameter

Für das Reinforcement-Learning-Setup wurden die TLS so konfiguriert, dass sie in SUMO als „aktuiert“² [47] initialisiert und anschließend direkt durch das RL-Modul über TraCI gesteuert werden konnten [53].

Die Simulation wurde mit folgenden Parametern durchgeführt:

- **Simulationszeitraum:** 5000 Sekunden
- **Zeitschritt (step-length):** 1,0 s
- **Routengenerierung:** deterministisch mit fixer seed zur Reproduzierbarkeit
- **Simulationstyp:** meso-Modus für Training, default-Modus für Evaluation
- **Verkehrsverteilung:** definiert über `flows.xml` und über Randkanten eingeleitet

²„Aktuiert“ bedeutet in SUMO, dass die Signalanlagen nicht strikt einem festen Schaltplan folgen, sondern dynamisch mit Hilfe von Detektoren oder externen Eingaben reagieren können. Im Rahmen dieser Arbeit erfolgt die Steuerung ausschließlich über das TraCI-Interface.

4.3 Reinforcement-Learning-Konzept

4.3.1 Formulierung des RL-Problems

Das Problem der Verkehrssteuerung wird als sequentielles Entscheidungsproblem [58] modelliert und mit Hilfe von Reinforcement Learning (RL) gelöst. Ein Agent interagiert dabei mit der SUMO-Simulationsumgebung, welche den Verkehrsfluss in diskreten Zeitschritten abbildet. Der Agent beobachtet den aktuellen Zustand s_t , wählt eine Aktion a_t , erhält eine Belohnung r_t und beeinflusst dadurch den Folgezustand s_{t+1} . Ziel ist es, eine Policy $\pi(a|s)$ zu erlernen, die langfristig die erwartete kumulierte Belohnung maximiert.

Im Trainingsskript sind die verschiedenen Belohnungsfunktionen jeweils als eigene Python-Funktion implementiert und werden über den Parameter `reward_fn` in der SUMO-Umgebung ausgewählt. Dadurch können unterschiedliche Optimierungsziele systematisch untersucht werden. Die Anbindung zwischen RL-Agent und Simulation erfolgt über das **Traffic Control Interface (TraCI)**, das eine Laufzeitsteuerung von SUMO ermöglicht (siehe Kapitel 2.2). Über TraCI werden sowohl Zustandsgrößen wie Rückstaulängen oder aktuelle Ampelphasen abgefragt als auch die Entscheidungen des Agenten direkt in die Simulation zurückgeschrieben. Dies bildet die Rückkopplungsschleife, die für den Trainingsprozess im Reinforcement Learning erforderlich ist.

4.3.2 Auswahl des RL-Algorithmus

Für das Training wurde der Algorithmus *Proximal Policy Optimization* (PPO) eingesetzt. PPO ist ein *on-policy* Policy-Gradient-Verfahren, das durch eine Clip-Funktion für Policy-Updates stabile Lernkurven und gute Sample-Effizienz ermöglicht [37]. Der Algorithmus hat sich in zahlreichen Studien zur Verkehrsoptimierung mit SUMO als robust und leistungsfähig erwiesen, insbesondere im Multiagenten-Setting [37, 70].

Begründung der Algorithmuswahl Alternative Verfahren wie *Deep Q-Networks* (DQN) oder *Advantage Actor-Critic* (A2C) wurden verworfen, da sie bei kontinuierlichen Zustandsräumen und in Multiagentenszenarien oft instabile Lernverläufe zeigen. *Soft Actor-Critic* (SAC) gilt zwar als state-of-the-art für kontinuierliche Steuerungsprobleme, erfordert aber große Replay-Puffer und hohe Rechenressourcen. PPO stellt somit einen sinnvollen Kompromiss zwischen Stabilität, Implementationsaufwand und empirischer Leistungsfähigkeit dar und integriert sich nahtlos in die bestehende `sumo-rl`-Umgebung.

4.3.3 Zustände

Der Zustand s_t beschreibt die Verkehrssituation an einer Kreuzung zum Zeitpunkt t . Die Zustandsrepräsentation umfasst: [47]

- Fahrzeuganzahl pro Spur (mittels `laneAreaDetector`),
- Länge der Warteschlange (Anzahl Fahrzeuge mit $v < 0.1$ m/s),
- Durchschnittsgeschwindigkeit pro Spur,
- aktuelle Ampelphase (TLS state),
- Dauer der aktuellen Phase,

- Binärmasken für die Phasenwechselbarkeit³,
- optional aggregierte Zustände benachbarter Kreuzungen (Multiagentensetting).

Die Merkmale werden normiert und als Eingabevektor an das neuronale Netz des Agenten übergeben.

4.3.4 Aktionen

Die Aktionsmenge A beschreibt die Eingriffsmöglichkeiten in die Steuerung der Lichtsignalanlage:

1. **Phasenwechsel-Modell:** Der Agent entscheidet binär, ob die aktuelle Phase fortgesetzt oder gewechselt wird ($A = \{\text{keep}, \text{switch}\}$).
2. **Direktwahl-Modell:** Der Agent wählt direkt eine der möglichen Phasen ($A = \{\text{phase}_0, \dots, \text{phase}_n\}$).

Einschränkungen sind u. a. Mindestgrünzeiten, automatisch generierte Zwischenphasen (Gelb, Räumzeit) und die Synchronität mehrerer Agentenentscheidungen. Zur Vermeidung hektischer Schaltungen wird ein Action-Interval (z. B. 5 s) definiert.

4.3.5 Belohnungsfunktionen

Im Rahmen dieser Arbeit wurden vier unterschiedliche Belohnungsfunktionen entwickelt und getestet. Eine Übersicht der Varianten, ihrer Optimierungsziele und Bezüge zur Literatur zeigt Tabelle 2.

Reward-Variante	Optimierungsziel	Bezug zur Literatur
diff-waiting-time	Reduktion der kumulierten Wartezeit durch Minimierung der Wartezeitdifferenz zwischen Zeitschritten	[54]
queue	Minimierung von Staus, gemessen über die Anzahl gestoppter Fahrzeuge	[54]
realworld	Kombination mehrerer in der Praxis messbarer Größen (Geschwindigkeit, Queues, Wartezeit)	Eigene Erweiterung
emissions	Erweiterung um CO ₂ -Emissionen zur Berücksichtigung von Nachhaltigkeit	[70, 40], Eigene Erweiterung

Tabelle 2: Übersicht der implementierten Reward-Funktionen und deren theoretische Motivation.

Für das Haupttraining wurde die Funktion **realworld** verwendet, da sie ausschließlich auf Metriken basiert, die in realen Städten mit vergleichsweise einfacher Sensorik (z. B. Induktionsschleifen, Kameras) erhoben werden können. Die übrigen Varianten wurden für Vergleichsstudien eingesetzt (siehe Anhang B).

³Die Phasenwechselbarkeit gibt an, ob ein Wechsel zu einer bestimmten nächsten Phase im aktuellen Zeitpunkt zulässig ist. Dies hängt von internen Restriktionen wie Mindestgrün- oder Gelbzeiten ab. Die Binärmaske verhindert damit ungültige Agentenaktionen.

Beschreibung der implementierten Funktionen

- **diff-waiting-time** (siehe Anhang B.1) Diese Funktion misst die Differenz der akkumulierten Wartezeit zwischen zwei Zeitschritten. Eine Abnahme der Gesamtwartezeit wird positiv belohnt.
- **queue** (siehe Anhang B.2) Hier wird die Anzahl der gestoppten Fahrzeuge direkt als negativer Reward verwendet. Weniger Fahrzeuge im Stau führen zu höherem Reward.
- **realworld** (siehe Anhang B.3) Diese Belohnungsfunktion kombiniert drei Metriken: durchschnittliche Geschwindigkeit, Länge der Warteschlangen und mittlere Wartezeit. Sie bildet einen Kompromiss zwischen Verkehrsfluss und Staureduktion.
- **emissions** (siehe Anhang B.4) Neben den Metriken aus **realworld** wird zusätzlich ein Term für CO₂-Emissionen berücksichtigt. Damit soll eine Balance zwischen Verkehrsleistung und Umweltverträglichkeit gefunden werden.

4.3.6 Hyperparameter-Anpassung

Für PPO wurden neben festen Werten auch dynamische *Schedules* eingesetzt, um Lernrate, Clip-Range oder Entropiekoeffizient während des Trainings anzupassen. Implementiert wurden u. a. Cosine-Warmup mit anschließendem Absenken, lineare Absenkung und adaptive Entropie-Steuerung.

Motivation der gewählten Schedules Diese Schedules vermeiden typische Probleme wie frühes Overfitting oder stagnierende Exploration. Das Cosine-Warmup erlaubt eine initiale Erkundung bei hoher Lernrate, während der anschließende Cosine-Decay den Lernprozess stabilisiert. Die adaptive Entropiesteuerung sorgt dafür, dass zu Beginn stark exploriert wird und im weiteren Verlauf zunehmend exploitation-basiert gelernt wird, ein Vorgehen, das sich in verwandten Arbeiten bewährt hat [38].

4.4 Analyse und Herausforderungen bei der OSM-Netznutzung

4.4.1 Grundstruktur von Lichtsignalanlagen in SUMO

Bevor die Probleme beim OSM-Import analysiert werden, ist es hilfreich, den Aufbau und die Abhängigkeiten der relevanten XML-Elemente in SUMO [47] zu verstehen, insbesondere im Zusammenhang mit der Steuerung von Lichtsignalanlagen.

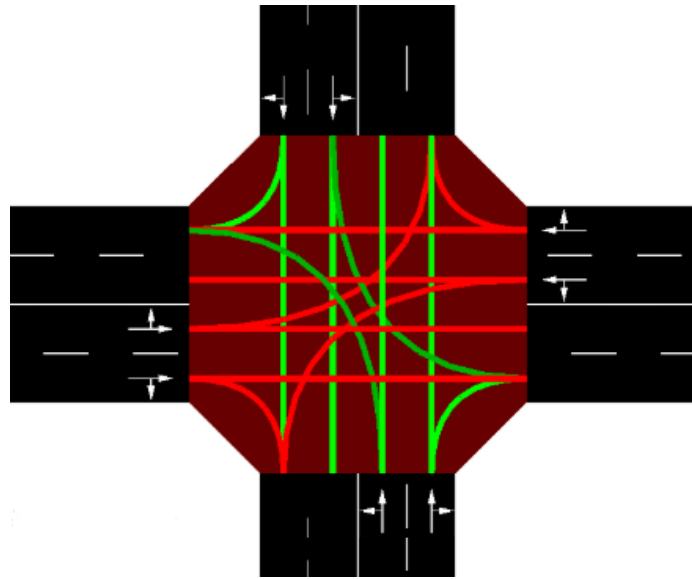


Abbildung 4: Visualisierung einer TLS-Kreuzung (Quelle nededit[28]).

- **<junction>** Definiert Knotenpunkte im Netz. Falls eine Ampel gesteuert wird, ist der Typ `type="traffic_light"`. Die ID entspricht in der Regel der TLS-ID.
- **<connection>** Verbindet zwei Fahrstreifen (von `from` nach `to`). Wenn diese Verbindung durch eine Ampel kontrolliert wird, enthält sie die Attribute `tl=tl_id` und `linkIndex`. Die Reihenfolge der `linkIndex`-Werte bestimmt die Position im Phasen-String.
- **Controlled Link** Jede `<connection>` mit einem `tl`-Attribut zählt als „gesteuerte Verbindung“. Die Anzahl solcher Verbindungen bestimmt die Länge des Phasenstrings (`state`).
- **<tlLogic>** Enthält die Steuerungslogik einer TLS. Jede `<tlLogic>` hat eine eindeutige ID (i.d.R. identisch zur `junction-ID`) und eine Liste von `<phase>`-Elementen.
- **<phase>** Jede Phase ist ein String (z.B. "grGr"), der den Zustand aller `linkIndex`-Verbindungen kodiert. Jeder Buchstabe (z.B. G = MajorGrün, g = MinorGrün, r = Rot) steht für den Status eines bestimmten kontrollierten Links.
- **<request>** Optionale Anforderungen einzelner Signalgruppen, meist bei aktuierten oder adaptiven TLS. Jeder Eintrag verweist über `index=` auf einen gesteuerten Link.

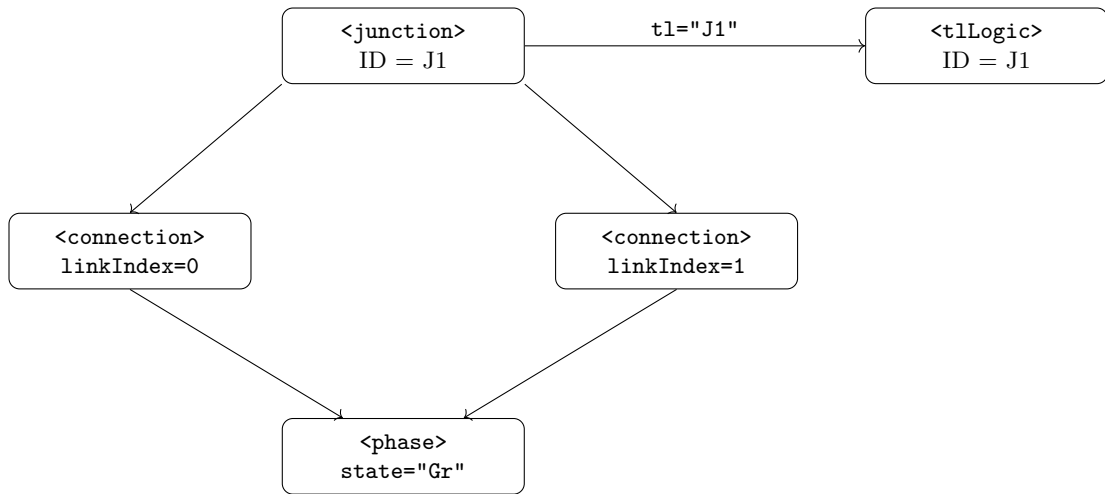


Abbildung 5: Zusammenspiel von Kreuzung, Verbindungen und Ampellogik in SUMO

4.4.2 Typische Fehlerquellen nach OSM-Import

Die automatische Ableitung von Ampelsteuerungen aus OSM ist unvollständig und fehleranfällig. Im Zusammenspiel mit `sumo-r1` ergeben sich daraus mehrere konkrete Probleme:

- **Fehlende oder unvollständige TLS-Definitionen:** In OSM sind Ampelanlagen in der Regel lediglich als Punktknoten mit dem Tag `highway=traffic_signals` erfasst. Die genaue Schaltlogik (`tlLogic`), also die Phasen und Zustände, fehlt vollständig. SUMO generiert daher beim Netzimport mit `-tls.guess-signals` heuristische Ampeldefinitionen, die jedoch oft lückenhaft oder unbrauchbar sind. [32, 45]
- **TLS mit nur einer Phase:** Viele der generierten Ampeln besitzen lediglich eine einzige definierte Phase. Dies entspricht keinem realen Verhalten und führt zu Fehlern beim Training mit `sumo-r1`, da das Framework mindestens zwei steuerbare Phasen voraussetzt. Die betroffenen Knoten müssen daher identifiziert und aus der Simulation ausgeschlossen oder manuell korrigiert werden. [53, 47, 45]
- **Unstimmige Phasenlängen:** Jede Phase in SUMO ist ein Zeichenstring (`state`), dessen Länge der Anzahl der gesteuerten Verbindungen (sogenannte *controlled links*) entsprechen muss. Bei fehlerhafter Generierung ist diese Bedingung oft verletzt, beispielsweise wenn der `state` zu kurz oder zu lang ist. Dies führt in `sumo-r1` zu Indexfehlern oder undefiniertem Verhalten. [53]
- **Fehlerhafte oder überzählige <request>-Einträge:** Jede TLS enthält in der Netzdatei zusätzliche Steuerinformationen über `request`-Elemente. Diese verweisen auf spezifische Signale mittels eines Index. Häufig verweisen diese Einträge jedoch auf nicht vorhandene Verbindungen, da `netconvert` Signalverknüpfungen nicht korrekt zuordnet. SUMO ignoriert solche Fehler teilweise still, während `sumo-r1` hingegen bricht mit Ausnahmen ab. [53]
- **Mehrdeutige oder verschachtelte Kreuzungen:** In komplexeren innerstädtischen Kreuzungen fasst SUMO mehrere OSM-Knoten zu einem „cluster“

zusammen, um den Verkehrsfluss abzubilden. Dies kann zu sehr großen TLS mit dutzenden Ein- und Ausfahrten führen, die übermäßig viele Phasen oder extrem lange Zustandsdefinitionen erzeugen. Solche TLS sind schwer zu debuggen und häufig inkompatibel mit den Erwartungen von `sumo-rl`. [53, 47]

Folgen für `sumo-rl` Das Framework `sumo-rl` erwartet für jede zu steuernde TLS: [53]

- mindestens zwei valide Phasen,
- konsistente Phasenzustände (`state`) mit korrekter Länge,
- vollständige Verbindungen zu kontrollierten Links,
- eindeutig identifizierbare TLS-IDs.

Sind diese Anforderungen nicht erfüllt, führt dies typischerweise zu einer der folgenden Fehlermeldungen:

- `IndexError: string index out of range`
- `ValueError: Invalid phase length`
- `KeyError: TLS not found`

Da diese Probleme nicht durch SUMO selbst gemeldet, sondern erst zur Laufzeit in `sumo-rl` sichtbar werden, ist ein systematischer Debugging- und Reparaturprozess zwingend notwendig. Die Komplexität steigt dabei exponentiell mit der Anzahl der TLS im Netz.

Erkenntnis Der direkte Import von OSM-Daten in SUMO erzeugt ein formal nutzbares Verkehrsnetz, jedoch nicht automatisch ein für Reinforcement Learning (RL) geeignetes. Ohne zusätzliche Aufbereitung ist ein stabiler Trainingsbetrieb in `sumo-rl` nicht möglich. Im Rahmen dieser Arbeit wurde das reale OSM-Netz von Karlsruhe daher gezielt analysiert, bereinigt und überarbeitet, sodass es nun erfolgreich und stabil im RL-Kontext eingesetzt werden kann. Dazu wurden eigene Werkzeuge zur automatisierten Strukturprüfung und Reparatur entwickelt, die im einem folgenden Abschnitt näher beschrieben werden.

4.4.3 Problematik nicht-motorisierter Verkehrswege im OSM-Modell

Ein zentrales Problem beim ursprünglichen OSM-Import stellten die Strukturen nicht-motorisierter Verkehrsträger dar, insbesondere Fußwege, Überwege und Fahrradtrassen.. Diese sind im OSM-Modell zwar detailliert erfasst, führen aber in SUMO häufig zu problematischen Simulationseffekten: [45]

- **Separate Fahrspuren für Radverkehr:** Zusätzliche Radstreifen erzeugen neue Kanten mit eigenen Abbiegebeziehungen, die von SUMO automatisch als TLS-relevant eingestuft werden, was häufig zu übermäßig vielen Signalgruppen führt.
- **Fußgängerüberwege mit Konfliktzonen:** `highway=crossing`-Elemente erzeugen automatisch Übergänge mit Konfliktzonen, die eine Ampelregelung erfordern, selbst wenn sie im Originalnetz nur symbolisch vorhanden sind.

- **Komplexität beim Entfernen:** Die gezielte Entfernung solcher Elemente führte häufig zu inkonsistenten Junctions und Netzfragmentierung. Ein manuelles Vorgehen wäre fehleranfällig und kaum skalierbar gewesen.

Diese Herausforderungen machten eine rein automatische Nutzung des OSM-Imports zunächst unmöglich. Erst durch gezielte algorithmische Nachbearbeitung konnte das Karlsruher Netz so transformiert werden, dass es für die RL-Simulation zuverlässig nutzbar wurde.

4.4.4 Eingesetzte `netconvert`-Optionen und deren Grenzen

Zur automatisierten Aufbereitung kamen zahlreiche Optionen von `netconvert` zum Einsatz, um das aus OSM exportierte Netz anzupassen. Dabei zeigte sich jedoch, dass viele dieser Optionen nicht auf die hohen Anforderungen von RL-Umgebungen zugeschnitten sind: [27]

- `-tls.guess-signals`: Erzeugt Ampeln auf Basis der Netzstruktur, allerdings oft mit unrealistischen oder unbrauchbaren Phasen.
- `-tls.join` und `-junctions.join`: Reduzieren Komplexität, erzeugen jedoch teils unübersichtliche Cluster, die schwer manuell kontrollierbar sind.
- `-ramps.guess`: Für urbane Netze weitgehend irrelevant oder sogar kontraproduktiv.
- `-remove-edges.isolated`, `-keep-edges.by-vclass` und `-discard-simple`: Dienen der Netzvereinfachung, führen aber oft zu strukturellen Problemen oder fehlenden funktionalen Verbindungen.

Obwohl diese Optionen wichtige Vorarbeiten leisteten, war ihre Wirkung für das RL-Zielmodell begrenzt. Erst durch zusätzliche Werkzeuge und maßgeschneiderte Filterlogik konnte das Netz gezielt bereinigt und optimiert werden.

4.4.5 Manuelle Eingriffe und strukturelle Rekonstruktionen

Neben automatisierten Bereinigungen waren auch gezielte manuelle Anpassungen notwendig. Insbesondere wurden mithilfe von `netedit` [28] einzelne Kreuzungen vollständig neu aufgebaut, um **Deadlocks zu vermeiden**, die zwar in der realen Verkehrsführung nicht auftreten, jedoch in SUMO durch implizite Abbiegelogiken und Vorrangregeln entstehen können.

Diese Rekonstruktionen erfolgten unter Beibehaltung der realweltlichen Topologie, jedoch mit einer technisch sauberen Definition aller Fahrbeziehungen und Signalisierungen. Damit konnte sichergestellt werden, dass auch komplexere Kreuzungen reproduzierbar, konfliktfrei und steuerbar bleiben.

Beseitigung von Deadlocks Während der initialen Simulationen traten in SUMO wiederholt Deadlocks an komplexen Kreuzungen auf, die in der realen Verkehrsführung nicht vorkommen. Ursache waren insbesondere von OSM importierte Rampen- und Abbiegebeziehungen, die zu konfliktbehafteten Fahrbeziehungen führten. Diese Knoten wurden in `netedit` gezielt angepasst:

- Anpassung der Geometrie, um SUMO-konforme Fahrspuren und eindeutige Vorrangregeln zu gewährleisten,

- Entfernung redundanter oder fehlerhafter Rampenverbindungen,
- Sicherstellung, dass jede Fahrbeziehung durch ein passendes Signal geregelt wird.

Die Änderungen wurden so umgesetzt, dass die realweltliche Topologie beibehalten wurde. Durch diese Eingriffe konnte die Simulation ohne Deadlocks und mit stabilen Verkehrsflüssen betrieben werden.

Visualisierung manueller Anpassungen Abbildung 6 zeigt exemplarisch eine Kreuzung vor und nach der manuellen Anpassung in **netedit**. Links ist die fehlerhafte Geometrie mit konfliktbehafteten Rampenverbindungen zu sehen, die in SUMO zu Deadlocks führten. Rechts ist die angepasste Version dargestellt, bei der alle Fahrbeziehungen eindeutig definiert und den Signalgruppen korrekt zugeordnet sind. Die Topologie entspricht der realen Verkehrsführung, wurde jedoch so modelliert, dass SUMO sie fehlerfrei simulieren kann.

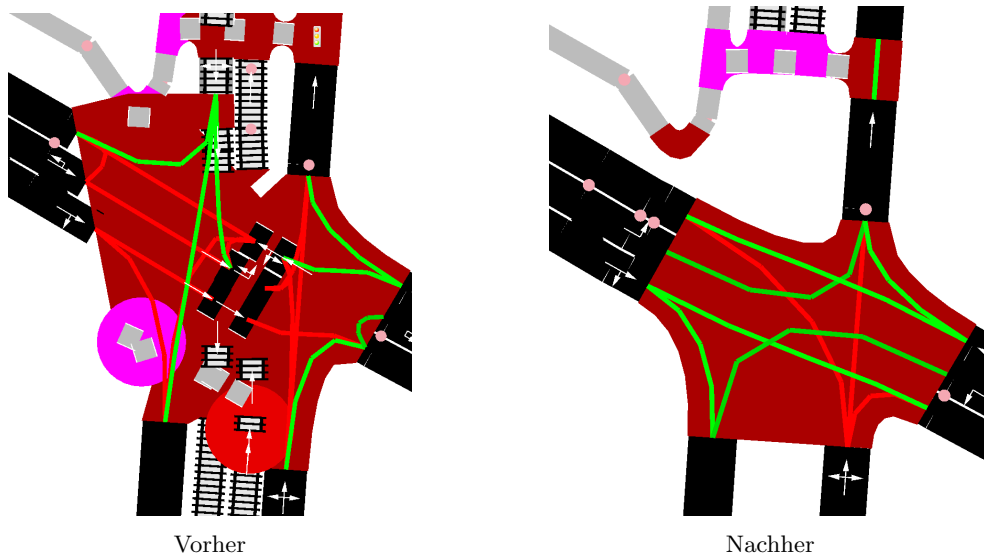


Abbildung 6: Vorher-Nachher-Vergleich einer angepassten TLS-Kreuzung (netedit [28])

4.4.6 Ergebnis: ein realistisches, RL-kompatibles Netz

Im Gegensatz zu einer synthetischen Umgebung basiert das nun eingesetzte Trainingsnetz auf realen topologischen Daten, wurde jedoch gezielt für den Einsatz mit **sumo-rl** überarbeitet. Es erfüllt folgende Eigenschaften:

- **Hohe Realitätsnähe bei kontrollierter Komplexität:** Das Netz bildet reale Strukturen ab, wurde jedoch so bereinigt, dass es RL-kompatibel bleibt.
- **Stabile TLS-Struktur:** Alle Kreuzungen mit Lichtsignalanlagen enthalten reproduzierbare und sinnvoll steuerbare Phasen.
- **Fehlerminimierung und Modularität:** Durch gezielte Reduktion und Nachbearbeitung sind Trainingsläufe wiederholbar und ohne strukturelle Störungen durchführbar.

- **Deadlock-Vermeidung durch gezielte Rekonstruktion:** Kritische Junctions wurden manuell so modelliert, dass sie SUMO-spezifische Blockadesituationen vermeiden, ohne die Realität zu verzerren.

Der Einsatz dieses verbesserten Karlsruher Netzes stellt einen zentralen methodischen Beitrag dieser Arbeit dar, da er demonstriert, wie reale OSM-Daten erfolgreich für das Reinforcement Learning nutzbar gemacht werden können, trotz ihrer ursprünglichen Limitierungen.

4.5 Netzprüfung, Reparatur und Toolchain

Aufgrund der oben beschriebenen strukturellen Schwächen im importierten OSM-Netz (siehe Kapitel 4.4.2) war eine manuelle Nachbearbeitung ineffizient und fehleranfällig. Daher wurden eigene Werkzeuge entwickelt, um eine systematische und automatisierte Reparatur zu ermöglichen.

4.5.1 Werkzeuge zur Netzprüfung und Reparatur

Um die Kompatibilität des aus OpenStreetMap abgeleiteten Verkehrsnetzes mit `sumo-r1` sicherzustellen, wurde eine Reihe eigenentwickelter Python-Skripte implementiert. Diese Werkzeuge automatisieren die Analyse, Validierung und Korrektur der Netzstruktur mit Fokus auf Lichtsignalanlagen (TLS). Der modulare Aufbau erlaubt es, problematische Netzbestandteile zu identifizieren und gezielt zu bereinigen.

Prüfung der Signalverknüpfungen und Zustandslängen Zwei zentrale Tools wurden entwickelt, um die Konsistenz zwischen kontrollierten Verbindungen (*controlled links*) und Phasenzuständen (*state*) der TLS zu überprüfen:

- `check_tls_consistency.py` (siehe Anhang E.1) prüft, ob die Länge jedes `state`-Strings in den `<phase>`-Elementen exakt der Anzahl der gesteuerten Signalindizes entspricht. Abweichungen werden detailliert gelistet, inklusive betroffener Phase und TLS-ID.

Algorithm 1 CheckTLSELengths – Prüfung inkonsistenter Phasenlängen

```
1: function CHECKTLSELENGTHS(net.xml)
2:   Lade XML-Baum und extrahiere <connection>-Elemente
3:   Erstelle Dictionary tls_controlled_links mit Anzahl gesteuerter Links pro
   TLS
4:   for all tlLogic-Elemente im Netz do
5:     expectedLen  $\leftarrow$  Anzahl controlledLinks aus Dictionary
6:     if expectedLen = 0 then
7:       Gib Warnung: TLS hat keine gesteuerten Verbindungen
8:       continue
9:     end if
10:    for all Phasen  $i$  in tlLogic do
11:      actualLen  $\leftarrow$  Länge des state-Strings
12:      if actualLen  $\neq$  expectedLen then
13:        Gib Warnung mit TLS-ID, Phase und state-Inhalt aus
14:      end if
15:    end for
16:  end for
17:  if keine Abweichungen gefunden then
18:    Gib Erfolgsmeldung aus
19:  end if
20: end function
```

- `check_tls_requests.py` (siehe Anhang E.2) validiert, ob alle <request>-Indizes innerhalb zulässiger Grenzen liegen. Falsch verknüpfte Einträge – z.B. `index > max(signalIndex)` – werden gemeldet.

Algorithm 2 CheckTLSRequests – Prüfung ungültiger request-Indizes

```
1: function CHECKTLSREQUESTS(net.xml)
2:   Lade XML-Datei und parse Wurzelknoten
3:   Erzeuge Dictionary tls_signal_indices mit Signalindizes je TLS aus
   <connection>-Elementen
4:   for all junction-Elemente im Netz do
5:     tls_id  $\leftarrow$  ID der Junction
6:     if tls_id in tls_signal_indices then
7:       expected_max  $\leftarrow$  Länge der Signalindizes für dieses TLS
8:       for all request-Elemente in Junction do
9:         index  $\leftarrow$  Wert des index-Attributs
10:        if index  $\geq$  expected_max then
11:          Gib Warnung mit tls_id und index aus
12:        end if
13:      end for
14:    end if
15:  end for
16:  if keine Warnungen ausgegeben then
17:    Gib Erfolgsmeldung aus
18:  end if
19: end function
```

Automatische Reparaturwerkzeuge Die folgenden Programme wurden zur strukturellen Korrektur entwickelt:

- **fix_requests.py** (siehe Anhang E.3) entfernt überzählige `<request>`-Einträge und kürzt `state`-Strings in Phasen auf die zulässige Länge. Die Bereinigung erfolgt anhand der tatsächlichen Anzahl gesteuerter Signalverbindungen (`linkIndex`).

Algorithm 3 FixRequests – Bereinigung ungültiger `<request>`-Einträge und Anpassung der Phasen

```

1: function FIXREQUESTS(net.xml)
2:   Lade XML-Baum mit Netzstruktur
3:   Initialisiere Dictionary tls_max_index für maximale Signalindices
4:   for all connection-Elemente do
5:     if TLS-ID und linkIndex vorhanden then
6:       Aktualisiere tls_max_index[t1] mit höchstem Index
7:     end if
8:   end for
9:   for all junction-Elemente do
10:    Hole TLS-ID
11:    if TLS nicht in tls_max_index then
12:      continue
13:    end if
14:    Bestimme erlaubten Maximalindex (max_idx)
15:    for all request-Einträge do
16:      if Index > max_idx then
17:        Entferne ungültigen request
18:      end if
19:    end for
20:    for all tlLogic-Elemente mit passender TLS-ID do
21:      for all Phasen do
22:        if state-String ist zu lang then
23:          Kürze state auf max_idx + 1
24:        end if
25:      end for
26:    end for
27:  end for
28:  Speichere modifizierte XML-Datei
29:  Gib Statistiken zu entfernten Requests und angepassten Phasen aus
30: end function

```

- **repair-net.py** (siehe Anhang E.4) nutzt ein manuell gepflegtes Dictionary mit TLS-IDs und deren erwarteter Phasenlänge (Anzahl kontrollierter Verbindungen). Alle Phasen, deren Länge abweicht, werden automatisch gekürzt oder aufgefüllt.

Algorithm 4 RepairTLSStates – Korrektur der Phasenlängen anhand manuell gepflegter Referenz

```

1: function REPAIRTLSSTATES(net.xml, referenz_dictionary)
2:   Lade Netzstruktur aus net.xml
3:   for all tlLogic-Elemente im Netz do
4:     tls_id  $\leftarrow$  ID des Ampelknotens
5:     if tls_id nicht in referenz_dictionary then
6:       continue
7:     end if
8:     correctLen  $\leftarrow$  erwartete Zustandslänge aus Referenz
9:     for all Phasen des Knotens do
10:      state  $\leftarrow$  Zeichenkette der Phase
11:      if Länge(state)  $\neq$  correctLen then
12:        Kürze oder ergänze state auf correctLen
13:        Markiere Netz als geändert
14:      end if
15:    end for
16:  end for
17:  if Netz wurde geändert then
18:    Speichere bereinigte Netzdatei als karlsruhe_fixed.net.xml
19:  else
20:    Gib Hinweis: Alle Phasen bereits korrekt
21:  end if
22: end function

```

- `statecheck.py` (siehe Anhang E.5) gibt eine Liste aller TLS-Phasen mit ungewöhnlichen Längen aus. Dieses Tool wurde verwendet, um bei vereinheitlichten Netzen auf eine Ziel-Zustandslänge zu prüfen.

Algorithm 5 StateCheck – Prüfung auf einheitliche Phasenlängen

```

1: function STATECHECK(net.xml)
2:   Lade XML-Baum aus der Netzdatei
3:   for all tlLogic-Elemente im Netz do
4:     tl_id  $\leftarrow$  ID des aktuellen TLS
5:     for all Phasen  $i$  in tlLogic do
6:       state  $\leftarrow$  Zustand der Phase
7:       if len(state)  $\neq$  57 then
8:         Gib Warnung mit tl_id, Phasenindex und tatsächlicher Länge aus
9:       end if
10:    end for
11:  end for
12: end function

```

Gültigkeitsprüfung für SUMO-RL Zur Vorbereitung des Trainings wurden weitere Programme zur Identifikation funktionaler TLS entwickelt:

- `find_valid_tls.py` (siehe Anhang E.6) iteriert über alle TLS im Netz und testet jede einzeln in einem minimalen `sumo-rl`-Lauf. TLS, bei denen die Umgebung erfolgreich initialisiert werden kann, gelten als kompatibel.

Algorithm 6 FindValidTLS – Gültigkeitsprüfung aller TLS im Netz

```
1: function TESTTLS(tls_id)
2:   Initialisiere SumoEnvironment
3:   Setze ts_ids auf [tls_id]
4:   Versuche: env.reset()
5:   if kein Fehler then
6:     env.close()
7:     return True
8:   else
9:     Gib Fehlermeldung aus
10:    return False
11:  end if
12: end function

13: Initialisiere leere Liste all_tls
14: Versuche: Umgebung mit SumoEnvironment zu starten
15: if erfolgreich then
16:   Lese alle ts_ids
17:   Schließe Umgebung
18: else
19:   Gib Fehler aus
20: end if

21: Initialisiere leere Liste valid_tls
22: for all tls_id in all_tls do
23:   if TESTTLS(tls_id) then
24:     Füge tls_id zu valid_tls hinzu
25:   end if
26: end for
27: Gib alle gültigen TLS aus
```

4.5.2 Auswahl eines bereinigten Netzes

Nach mehrfacher Iteration und Debugging wurde ein final bereinigtes Netz erzeugt: `network.net.xml`. Dieses enthält ausschließlich überprüfte TLS mit konsistenten Phasenlängen und steuerbaren Verbindungen. Es bildet die Grundlage für alle nachfolgenden Reinforcement-Learning-Experimente.

4.5.3 Vorteil des automatisierten Workflows

Die entwickelte Toolchain ermöglicht:

- eine strukturierte Diagnose typischer OSM-bedingter Netzprobleme,
- reproduzierbare Netzreparaturen ohne reines manuelles Editieren in `netedit` [28],
- gezielte Selektion steuerbarer TLS für das Experiment.

Der Einsatz dieser Werkzeuge war unerlässlich, um ein funktionales, kompatibles und robusteres Simulationsnetz auf Basis realer OSM-Daten zu etablieren.

4.6 Einbindung des SUMO-Netzes in die RL-Umgebung

Nach Abschluss der Netzbereinigung, der strukturellen Validierung und der Identifikation steuerbarer Lichtsignalanlagen (TLS) wurde das finale Verkehrsnetz in eine auf `sumo-rl` basierende Reinforcement-Learning-Umgebung integriert. Ziel war die Realisierung einer robusten, modularen Multiagentenumgebung, die eine lernbasierte Optimierung der Verkehrssteuerung unter realitätsnahen Bedingungen erlaubt.

4.6.1 Gesamtsystem und Architektur

Die Architektur der Lernumgebung ist als verteiltes Multiagentensystem ausgelegt, bei dem jede signalgesteuerte Kreuzung durch einen eigenständigen Agenten repräsentiert wird. Die Interaktion erfolgt über das TraCI-Protokoll [61] von SUMO [47], das eine Echtzeitkommunikation zwischen Simulator und RL-Agenten ermöglicht. Die zentrale Steuerung und das Training der Agenten basiert auf der RL-Bibliothek `Stable-Baselines3` [42], konkret dem Algorithmus `Proximal Policy Optimization` (PPO) [37]. [53]

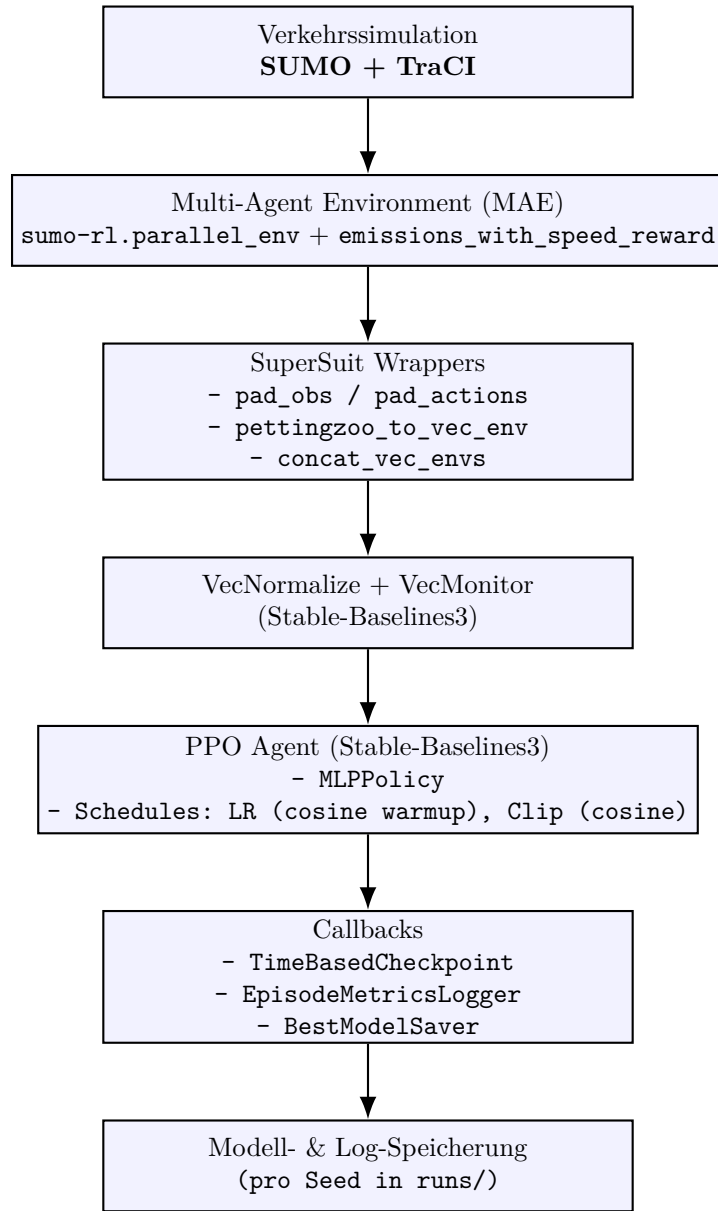


Abbildung 7: Architektur der RL-Trainingspipeline mit SUMO, MAE und Stable-Baselines3

Zur Vereinheitlichung der Multiagentenumgebung kamen die Bibliotheken **PettingZoo** [35] und **SuperSuit** [56] zum Einsatz. **PettingZoo** stellt ein standardisiertes API für Multiagenten-Umgebungen bereit, analog zu **Gymnasium** [14], jedoch speziell für Szenarien mit mehreren Agenten. **SuperSuit** erweitert diese Umgebungen durch eine Vielzahl an Wrappers, beispielsweise zur Vereinheitlichung von Beobachtungs- und Aktionsräumen (z. B. durch Padding) oder zur Umwandlung in vektorisierte Formate. Für parallele Ausführung ist die Umgebung mittels `concat_vec_envs_v1` auf Mehrkernbetrieb vorbereitet (`num_cpus = 8`); im vorliegenden Skript (siehe Anhang A.1) wird eine Instanz (`num_vec_envs = 1`) betrieben, sodass die Skalierung ohne

Codeänderungen möglich ist.

4.6.2 Agentenzuordnung im Multiagentensystem

Jede steuerbare Lichtsignalanlage (TLS) wird einem eindeutigen Agenten zugeordnet. Die Zuordnung erfolgt deterministisch anhand der in der SUMO-Netzdatei vergebenen TLS-IDs. Die Agenten handeln vollständig lokal, es findet kein direkter Informationsaustausch zwischen ihnen statt. Jede Beobachtung wird vor der Weitergabe an die Policy mittels `VecNormalize` standardisiert, sodass alle Eingaben über verschiedene Kreuzungen hinweg vergleichbar sind. Die Beobachtungsvektoren werden mit `pad_observations_v0` auf eine feste Länge gebracht, die sich aus der maximalen Spuranzahl einer Kreuzung multipliziert mit der Anzahl Spurfeatures (Queue, Geschwindigkeit, Dichte) zuzüglich der Phasenfeatures (Phase, Phasendauer) ergibt.

4.6.3 Vermeidung von Überanpassung

Um zu verhindern, dass die trainierten Agenten lediglich auf wiederkehrende Verkehrsmuster optimiert werden, wurden die Routenflüsse mit `randomTrips.py` generiert. Dadurch variiert die Nutzung einzelner Strecken, und es entstehen unterschiedliche Verkehrsverteilungen innerhalb der Episoden. Zusätzlich wurden während des Trainings mit `TensorBoard` zentrale Leistungsmetriken (mittlere Geschwindigkeit, Wartezeit, CO₂-Emissionen, Durchsatz) in Echtzeit überwacht, um sicherzustellen, dass sich die Leistungsverbesserungen nicht nur auf die Trainingsdaten beziehen. Evaluationsläufe mit zuvor nicht genutzten Zufallsseeds dienen der abschließenden Validierung (siehe Kapitel 5).

4.6.4 Konfiguration der Umgebung

Die Trainingsumgebung basiert auf der Klasse `parallel_env` aus dem `sumo-rl`-Framework, wurde jedoch stark erweitert und in eine Stable-Baselines3-kompatible Vektor-Umgebung eingebettet. Dabei wurden die folgenden Aspekte berücksichtigt:

- **Verkehrsnetz:** Grundlage ist das bereinigte Netz `map.net.xml`.
- **Routenprofile:** Drei Routendateien (`flows_low.rou.xml`, `flows_medium.rou.xml`, `flows_high.rou.xml`) werden zyklisch per Reset geladen, sodass das Modell im Training auf verschiedene Lastprofile trifft.
- **Seed-Handling:** Vier unterschiedliche Seeds (`{143534, 456, 635768, 13755}`) sorgen für Reproduzierbarkeit und Variation.
- **Episode-Länge:** Jede Episode läuft für 4096 Simulationssekunden.
- **Entscheidungsintervall:** Ampelaktionen werden alle 5 s (`delta_time`) neu gewählt.
- **Belohnungsfunktionen:** Durch die direkte Erweiterung des `sumo-rl`-Frameworks können unterschiedliche Reward-Funktionen flexibel eingebunden werden, wodurch eine einheitliche Schnittstelle für Experimente mit verschiedenen Zielfunktionen entsteht (siehe Anhang B).
- **Systemmetriken:** `add_system_info=True` aktiviert globale Kennzahlen wie mittlere Wartezeit oder Gesamtemissionen.

- **Multi-Agent-Setup:** Jede steuerbare Ampel im Netz wird als eigenständiger Agent behandelt, was der PettingZoo-Schnittstelle entspricht.
- **Integration in SB3:** Über `SuperSuit` (Padding, Konvertierung, Vektorisierung) wird die Umgebung an `Stable-Baselines3` angebunden.
- **Normalisierung:** Rewards und Beobachtungen werden mit `VecNormalize` z-standardisiert, um das Training zu stabilisieren.

Zusätzlich sind mehrere Callback-Mechanismen implementiert:

- **Checkpoints:** Zeitbasiertes Speichern der Modelle (stündlich).
- **Logging:** Aggregierte Episodenmetriken (Wartezeiten, Queues, Fluss) werden in TensorBoard und CSV-Dateien geschrieben.
- **Bestes Modell:** Automatisches Sichern des jeweils besten Modells nach mittlerem Episoden-Reward.

Das Training erfolgt mit PPO (`stable-baselines3`) und nutzt eine Cosine-Schedule für Lernrate und Clip-Range, sowie eine zweischichtige MLP-Policy ([128, 128]). Jede Run wird im Ordner `runs/` mit Zeitstempel protokolliert.

Die Umgebung ist vollständig kompatibel mit `Gymnasium`, `PettingZoo` sowie den Wrapper-Bibliotheken `SuperSuit` und `VecEnv`, wodurch ein standardisiertes Interfacing mit RL-Algorithmen ermöglicht wird.

4.6.5 Trainingsalgorithmus und Hyperparameter

Das Training der Agenten erfolgte mittels PPO, wobei folgende Hyperparameter eingesetzt wurden (siehe Anhang A.1):

- **Policy-Architektur:** Zwei Hidden-Layer mit jeweils 128 Neuronen; getrennte Netze für `pi` und `vf` (`policy_kwargs`).
- **Batchgröße:** 512.
- **Rollout-Länge (`n_steps`):** 2048.
- **Lernrate:** Cosine-Warmup über 5% der Trainingszeit mit anschließendem Cosine-Decay auf 10% des Startwerts ($3 \cdot 10^{-4}$).
- **Clip-Range:** Cosine-Interpoliert von 0.2 auf 0.1.
- **Entropiekoeffizient:** 0.01 (Exploration).
- **Discount-Faktor:** $\gamma = 0.99$.
- **GAE-Lambda:** 0.95.
- **Device:** CPU (`torch.set_num_threads` auf Standard; SUMO-Simulation single-threaded).

Beobachtungen und Rewards wurden über `VecNormalize` [67] normalisiert (`clip_obs = 10.0`, `clip_reward = 10.0`, `norm_obs = True`, `norm_reward = True`), die Umgebung zusätzlich mit `VecMonitor` [66] überwacht (`monitor.csv`). Pro Seed wurden $2 \cdot 10^6$ Trainingsschritte durchgeführt.

Die Trainingsdauer pro Seed betrug im Mittel 2 Stunden für $2 \cdot 10^6$ Schritte bei einer Episodenlänge von `num_seconds = 5000`, inklusive periodischer Checkpoint-Speicherung und Logging. Dies variiert mit der eingesetzten Belohnungsfunktion. Kürzere Vergleichsläufe mit $5 \cdot 10^5$ Schritten wurden in etwa 35 Minuten abgeschlossen.

Die Wahl der Hyperparameter erfolgte auf Basis einer Kombination aus Literaturrecherche und empirischen Vorversuchen auf einem synthetischen Verkehrsnetz, das strukturell an das finale Zielnetz angelehnt war. Die Netzarchitektur mit zwei Hidden-Layern à 128 Neuronen wurde gewählt, da sie in vergleichbaren SUMO-RL-Szenarien eine gute Balance zwischen Modellkapazität und Generalisierungsfähigkeit bietet [42, 37].

Die Batchgröße von 512 und die Rollout-Länge von 2048 Schritten stellen sicher, dass pro Policy-Update ausreichend diversifizierte Datenpunkte vorliegen, ohne den Speicherbedarf zu stark zu erhöhen. Der Cosine-Schedule für Lernrate und Clip-Range dient dazu, zu Beginn eine hohe Exploration zuzulassen und gegen Ende des Trainings die Updates zu stabilisieren. Der Entropiekoeffizient von 0,01 wurde so gewählt, dass auch in späten Trainingsphasen eine gewisse Exploration erhalten bleibt.

4.6.6 Checkpoints, Monitoring und Logging

Zur Sicherstellung eines robusten und reproduzierbaren Trainingsprozesses wurde ein umfassendes Callback- und Logging-System integriert (siehe Anhang A.1). Dieses setzt direkt auf den Mechanismen von Stable-Baselines3 auf und umfasst:

- **TimeBasedCheckpointCallback:** Zeitbasierte Sicherung des aktuellen Modells und der Normalisierungsdaten in festen Intervallen (standardmäßig stündlich). Die Dateien werden mit Schrittzähler im Namen versioniert und ermöglichen die Wiederaufnahme ab beliebigen Zwischenständen.
- **EpisodeMetricsLoggerCallback:** Aggregiert Kennzahlen direkt aus dem `infos`-Dictionary der SUMO-Umgebung und schreibt sie in das Stable-Baselines3-Logger-Interface. Damit wird ein kontinuierliches Monitoring pro Episode gewährleistet.
- **BestModelSaverCallback:** Automatische Speicherung des jeweils besten Modells, gemessen am mittleren Episodenreward (`ep_info_buffer`). Neben den Modellparametern werden auch die Normalisierungsdaten gesichert, sodass eine konsistente Reproduktion möglich ist.

Geloggte Metriken. Alle relevanten Metriken werden zusätzlich im CSV-Format und über TensorBoard [59] erfasst. Die wichtigsten überwachten Größen sind:

- **Szenario und Methode:** `scenario`, `method`, `run_dir`,
- **Systemmetriken:** Anzahl fahrender Fahrzeuge (`system_total_running`), gestoppter Fahrzeuge (`system_total_stopped`), eingetroffener Fahrzeuge (`system_total_arrived`), abgefertigter Fahrzeuge (`system_total_departed`), backlogged Fahrzeuge (`system_total_backlogged`), Teleports (`system_total_teleported`),
- **Leistungsmetriken:** mittlere Wartezeit (`system_mean_waiting_time`), gesamte Wartezeit (`system_total_waiting_time`), mittlere Geschwindigkeit (`system_mean_speed`),
- **Trainingsmetriken:** mittlerer Episodenreward (`ep_rew_mean`), Episodenlänge (`ep_len_mean`), verwendeter Seed (`ep_seed_mean`), sowie aggregierte Episodenstatistiken (`episode_mean`).

TensorBoard diene als zentrales Werkzeug, um den Verlauf relevanter Kennzahlen in Echtzeit zu überwachen und Trainingstendenzen (z. B. Reward-Stabilität, Auswirkungen von Teleports oder Backlog-Aufbau) nachvollziehbar zu machen.

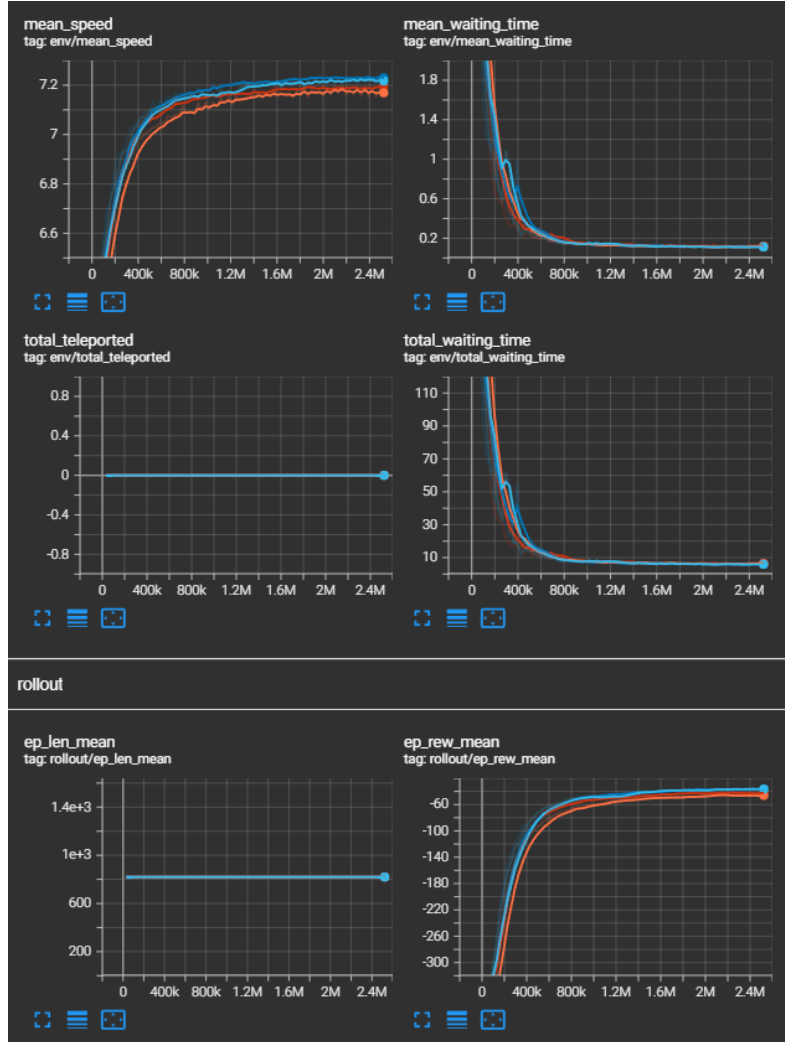


Abbildung 8: Beispielhafte Lernkurve eines Trainingslaufs, visualisiert mit TensorBoard. Der Reward stabilisiert sich nach einer Explorationsphase.

Artefakte und Struktur. Alle Modellartefakte (.zip, vecnormalize.pkl), sowie die zugehörigen monitor.csv-Dateien und TensorBoard-Logs werden pro Seed in einem Run-spezifischen Ordner gespeichert (runs/ppo_sumo_<SEED>_<YYYY-MM-DD_HH-MM-SS>). TensorBoard diene als zentrales Werkzeug, um den Verlauf relevanter Kennzahlen in Echtzeit zu überwachen und Trainingstendenzen (z. B. Reward-Stabilität, Auswirkungen von Teleports oder Backlog-Aufbau) nachvollziehbar zu machen.

4.6.7 Systemumgebung und Durchführung

Hardware-Setup Alle Trainingsläufe wurden auf einem System mit Intel® Core™ i7-6700K Prozessor (4 Kerne, 8 Threads, 4,0 GHz) und 16 GB RAM durchgeführt.

Aufgrund der single-threaded Ausführung der SUMO-Simulation wurde der RL-Algorithmus ebenfalls auf der CPU betrieben (`device='cpu'`), was eine konsistente Laufzeitmessung und Reproduzierbarkeit sicherstellte. Der durchschnittliche Speicherverbrauch pro Trainingslauf lag bei rund 9 GB. Die Multi-Core-Parallelisierung über `concat_vec_envs_v1` war vorbereitet, wurde jedoch nicht aktiviert, um Speicherverbrauch und Interprozesskommunikations-Overhead zu vermeiden. Während des Trainings wurde eine durchschnittliche CPU-Auslastung von 60 %–70 % verzeichnet.

Versionen und Systemumgebung Die Experimente wurden unter folgender Systemumgebung durchgeführt:

- **Betriebssystem:** Windows 10 Pro (64-Bit)
- **Python:** 3.10.12
- **SUMO:** 1.18.0 (mit HBEFA 4.1-Emissionsmodell)
- **Stable-Baselines3:** 2.1.0
- **Gymnasium:** 0.29.1
- **PettingZoo:** 1.23.0
- **SuperSuit:** 3.9.0
- **PyTorch:** 2.1.2+cpu
- **sumo-rl:** 1.4.5

Die Pfade zu SUMO wurden über die Umgebungsvariable `SUMO_HOME` gesetzt. Alle Experimente wurden mit deaktiviertem GUI-Modus gestartet, um den Simulationsfaktor zu maximieren.

Reproduzierbarkeit und Laufverwaltung Zur Sicherstellung der Reproduzierbarkeit wurden Seeds für NumPy, PyTorch und SUMO gesetzt; zusätzlich wird (falls verfügbar) die Umgebung selbst mit `env.seed(SEED)` initialisiert. Es wurden vier verschiedene Seed-Werte $\{546456, 678678, 234256, 678\}$ verwendet. Eine Hilfsroutine (`find_latest_complete_run`) erlaubt die Identifikation des letzten vollständigen Laufs (finales Modell oder jüngster Checkpoint inkl. `VecNormalize`) zur Wiederaufnahme. Evaluationsläufe können nach dem Training mit deaktivierter Exploration (`deterministic=True`) gestartet werden.

Fehler- und Abbruchbehandlung Bei manuellem Abbruch (`KeyboardInterrupt`) werden der aktuelle Modellstand (`model_interrupt.zip`) und Normalisierungsdaten (`vecnormalize_interrupt.pkl`) gesichert. Im `finally`-Block wird die Normalisierung zusätzlich persistiert und die Umgebung sauber geschlossen.

Simulationsgeschwindigkeit und Parallelität Die Trainingsdauer für ein vollständiges Haupttraining mit 2×10^6 Schritten beträgt im Mittel etwa zwei Stunden pro Seed, wobei die genaue Laufzeit stark von der Komplexität der verwendeten Reward-Funktion abhängt. Eine Parallelisierung über mehrere CPU-Kerne wurde nicht umgesetzt, da `sumo-rl` in der genutzten Version nicht vollständig thread-sicher ist und `TraCI` keine automatisierte Portzuweisung für parallele SUMO-Instanzen erlaubt. Der durchschnittliche Simulationsdurchsatz lag bei etwa 700–800 Simulationsschritten pro realer Sekunde im `meso`-Modus und 250–300 Schritten/s im `default`-Modus (ohne GUI).

4.6.8 Zusammenfassung

Die konfigurierte RL-Umgebung erlaubt eine modulare und flexible Steuerung realer Verkehrsnetze auf Basis von SUMO. Durch die Kombination aus systematischer TLS-Auswahl, stabiler und geglätteter Reward-Funktion, adaptiven Hyperparameter-Schedules, vorbereiteter Multi-Core-Parallellisierung, konsequenter Reproduzierbarkeit und umfassendem Monitoring (inkl. Checkpoints, Metriklogging und Bestmodell-Erkennung) wurde eine solide Grundlage für die experimentelle Evaluation lernbasierter Verkehrssteuerung geschaffen.

4.7 Evaluationsstrategie

Zur Bewertung der trainierten Modelle wird ein systematischer, skriptbasierter Evaluationsplan verfolgt (Details und Ergebnisse siehe Kapitel 5). Dabei wird stets dasselbe, bereinigte Verkehrsnetz (`map.net.xml`) verwendet, um die Vergleichbarkeit zu gewährleisten. Variiert werden Signalsteuerung, Routenbelegung und Seeds. Statische Baselines werden direkt im Evaluationslauf mitgeführt (fester Phasenplan und aktuiert).

4.7.1 Vergleichsszenarien

Die Evaluation der RL-Agenten und der Baselines erfolgt vollautomatisch über `evaluate.py`. Vier Szenarien decken typische Lastprofile ab:

- **morning_peak** (`flows_morning.rou.xml`): morgendliche Lastspitze,
- **evening_peak** (`flows_evening.rou.xml`): abendliche Lastspitze,
- **uniform** (`flows_uniform.rou.xml`): gleichmäßige Zuflüsse,
- **random_heavy** (`flows_random_heavy.rou.xml`): stochastisch dichter Verkehr.

Die Lastprofile *morning_peak* und *evening_peak* wurden so definiert, dass morgens der Verkehr überwiegend in das Stadtzentrum hineinführt, während er abends nach außen abfließt. Das Szenario *random_heavy* dient dazu, das Netz an seine Kapazitätsgrenzen zu bringen und die Robustheit der Steuerungsansätze unter hoher Last zu überprüfen.

4.7.2 Bewertete Modelle

Die zu evaluierenden RL-Modelle werden automatisch aus `runs/` ermittelt:

- Alle Unterordner mit Präfix `ppo_sumo_*`.
- Aus jedem Lauf wird das **beste Modell** `best_model.zip` sowie die zugehörige **Normalisierung** `vecnormalize.pkl` geladen.

Zusätzlich zu den RL-Läufen werden im selben Skript die Baselines (Fixed-Time, Actuated) ausgeführt.

4.7.3 Reward-Funktionen

Im Evaluationslauf werden Rewards nicht genutzt. Stattdessen basiert die Bewertung ausschließlich auf den von SUMO gelieferten Systemmetriken (`system_mean_*`, `system_total_*`). Dies stellt sicher, dass unterschiedliche Modelle und Baselines unabhängig von der jeweiligen Trainings-Rewardfunktion objektiv vergleichbar sind.

4.7.4 Simulations- und Umgebungsparameter

Für alle Simulationen (RL und Baselines) gelten konsistente Parameter:

- Episodenlänge: `EP_LENGTH_S = 4096 s`,
- Minimaldauer Grünphase: `min_green = 5 s`,
- Maximaler Abfahrtsverzug: `max_depart_delay = 100 s`,
- Zusätzliche Systemmetriken: `add_system_info = True`,
- Pro-Agent-Infos: `add_per_agent_info = False`,
- SUMO läuft ohne GUI (`use_gui=False`).

RL-Umgebung. Die PettingZoo-Umgebung wird mit SuperSuit vorbereitet (`pad_observations_v0`, `pad_action_space_v0`, `pettingzoo_env_to_vec_env_v1`, `concat_vec_envs_v1`) und mit `VecMonitor` versehen. Die Normalisierung wird aus `vecnormalize.pkl` geladen und eingefroren (`training=False`, `norm_reward=False`); die Modelle werden deterministisch ausgeführt.

Baselines. Für die Baselines wird die SUMO-interne Steuerung genutzt (`fixed_ts=True` für Fixed-Time, `fixed_ts=False` für Actuated). Ein *Dummy-Reward* (0.0) stellt sicher, dass kein RL-Einfluss auf die Steuerung erfolgt; die Aktionen bestehen aus einer konstant wiederholten, gültigen Sample-Aktion.

4.7.5 Ablauf je Episode

Der Ablauf ist für RL und Baselines einheitlich:

1. Reset der Umgebung,
2. Simulation bis zum Terminalzustand,
3. Pro Schritt: (i) Aktion (bei RL deterministisch via `model.predict`, bei Baselines konstante Dummy-Aktion), (ii) `env.step`, (iii) Metrik-Sampling aus `infos`,
4. Episodenende: Mittelwerte aller `system_mean_*`-Keys sowie Übernahme der finalen `system_total_*`-Werte.

4.7.6 Seeds und Replikationsdesign

Das Seed-Design ist eindimensional: Es wird eine feste Menge von zehn *Episoden-Seeds* (`EP_SEEDS`) verwendet, die vom Training entkoppelt sind. Pro Kombination aus *Szenario*, *Methode* und *Modelllauf* werden `N_EPISODES = 10` Episoden ausgewertet. Da zwei Baselines (Fixed-Time, Actuated) sowie alle RL-Läufe unter `runs/ppo_sumo_*` berücksichtigt werden, ergibt sich insgesamt (mit 4 Seeds):

$$\begin{aligned}\#\text{Episoden} &= (2 + |\text{RUNS}|) \cdot |\text{SCENARIOS}| \cdot \text{N_EPISODES} \\ &= (2 + |\text{RUNS}|) \cdot 4 \cdot 10 \\ &= 240\end{aligned}$$

4.7.7 Metriken und Logging

Es werden alle numerischen `system_mean_*`- und `system_total_*`-Keys gespeichert. Für die Visualisierung werden Präfixe teilweise entfernt (`system_*` → ohne Präfix). Pro Kombination werden Metriken für `Baseline_FixedTime`, `Baseline_Actuated` und RL geloggt (`tensorboard` und `stdout`); die Logs liegen unter `evaluation/logs/eval_{scenario}`. Die aggregierten Ergebnisse werden zusätzlich in `evaluation/eval_results.json` persistiert.

4.7.8 Reproduzierbarkeit

Die Evaluation ist deterministisch durch:

1. feste `EP_SEEDS`,
2. deterministische Aktionswahl bei RL (`deterministic=True`),
3. eingefrorene `VecNormalize`-Parameter,
4. feste Simulationsdauer, identische Netz- und Routen-Dateien.

Diese Strategie stellt sicher, dass die in Kapitel 5 präsentierten Ergebnisse reproduzierbar, belastbar und zwischen RL-Varianten sowie Baselines vergleichbar sind.

Das gesamte Vorgehen ist in einem dedizierten Evaluationsskript (`evaluate.py`) implementiert, welches im Anhang dokumentiert ist (siehe Anhang C.1). Dieses Skript stellt sicher, dass sämtliche hier beschriebenen Schritte systematisch und reproduzierbar umgesetzt werden.

5 Evaluation und Ergebnisse

In diesem Kapitel werden die Ergebnisse der Evaluationsläufe präsentiert. Alle Modelle (vier Trainingsseeds pro Reward-Variante) sowie die beiden Baselines (Fixed-Time, Actuated) wurden in allen vier Szenarien (`morning_peak`, `evening_peak`, `uniform`, `random_heavy`) jeweils über zehn Episoden evaluiert.

Bemerkung: Die Resultate sind Mittelwerte pro Kombination aus *Methode* und *Szenario*. Die Baseline Actuated hat in allen Evaluierungen relativ schlecht abgeschlossen und wurde größten teils separat in Diagrammen dargestellt um sicherzustellen, dass die deutlich besser performende Baseline TimeFixed, visuell deutlich mit den Modellen vergleichbar bleibt. Bei allen Episoden sind gleich viele Fahrzeuge in das Netz gespeist worden.

5.1 Reward: Diff-Waiting-Time

Die erste Gruppe von RL-Modellen wurde mit einem Reward trainiert, der die Differenz der Wartezeiten minimiert.

5.1.1 Mittlere Wartezeiten

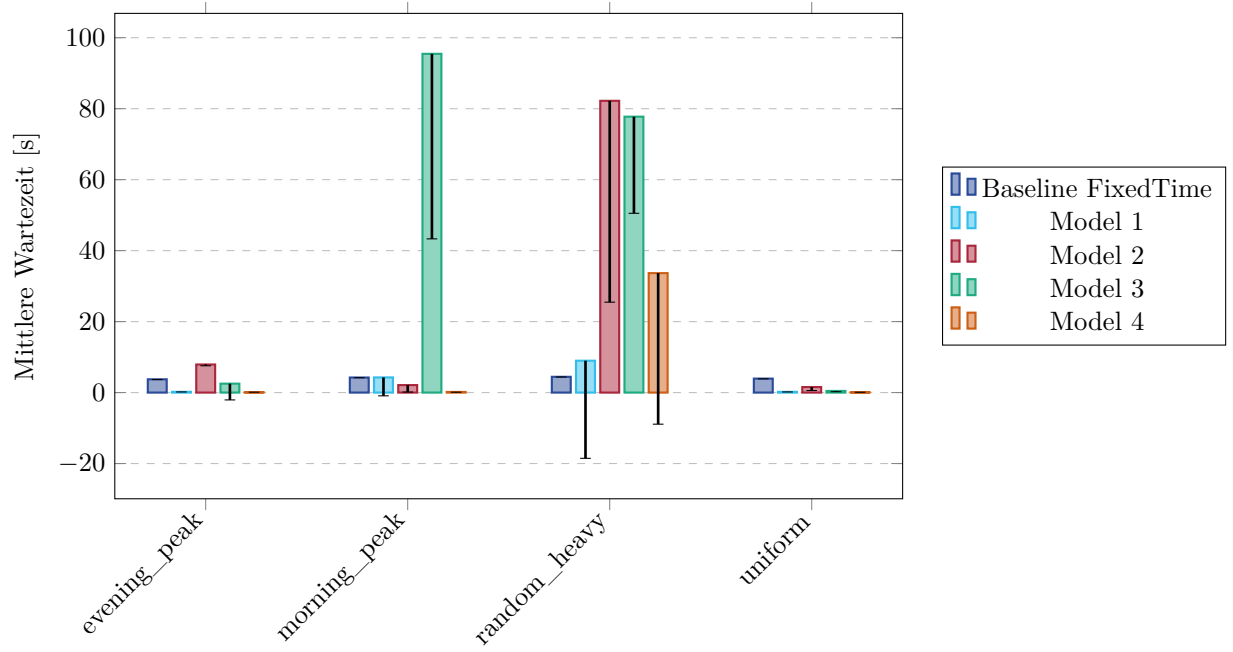


Abbildung 9: Mittlere Wartezeiten

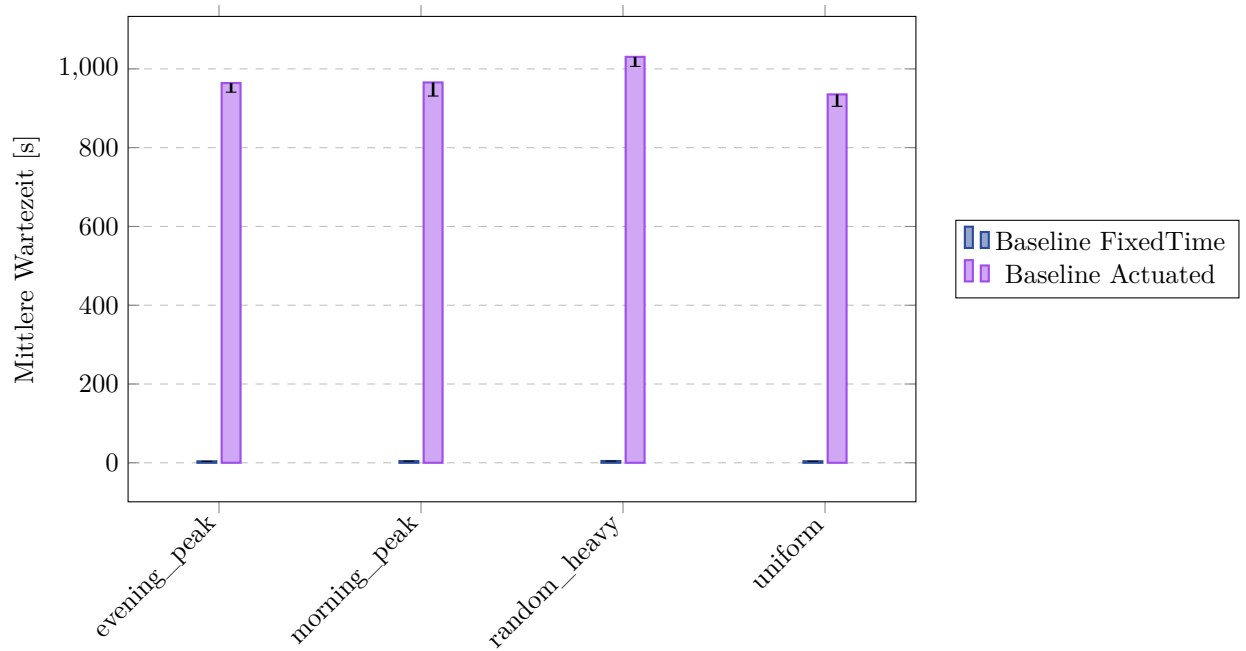


Abbildung 10: Mittlere Wartezeiten

Die Analyse der mittleren Wartezeit zeigt, dass die RL-Modelle in den meisten Szenarien bessere Ergebnisse als die Baselines erzielen. Insbesondere in den Szenarien `uniform`, `morning_peak` und `evening_peak` konnten mehrere Modelle die durchschnittliche Wartezeit signifikant reduzieren. Gleichzeitig ist zu beobachten, dass einzelne Modelle in bestimmten Szenarien auffällig schlechter abschnitten: Modell 3 in `morning_peak` sowie Modell 2 und Modell 3 in `random_heavy` weisen deutlich erhöhte Wartezeiten auf. Diese hohen Werte gehen zudem mit einer stark erhöhten Standardabweichung einher, was auf eine erhöhte Variabilität im Verhalten der Modelle in diesen Konfigurationen hinweist.

5.1.2 Anzahl stoppender Fahrzeuge

In sumo werden Fahrzeuge die sich mit einer Geschwindigkeit kleiner als 0.1 m/s bewegt.

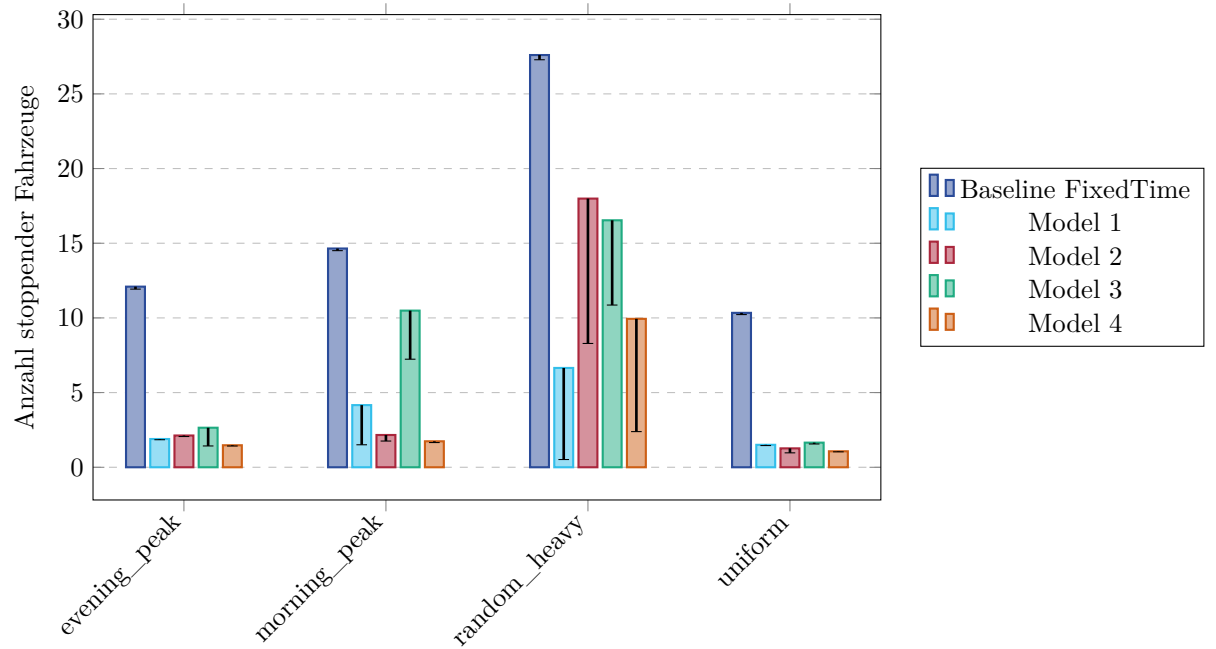


Abbildung 11: Anzahl stoppender Fahrzeuge

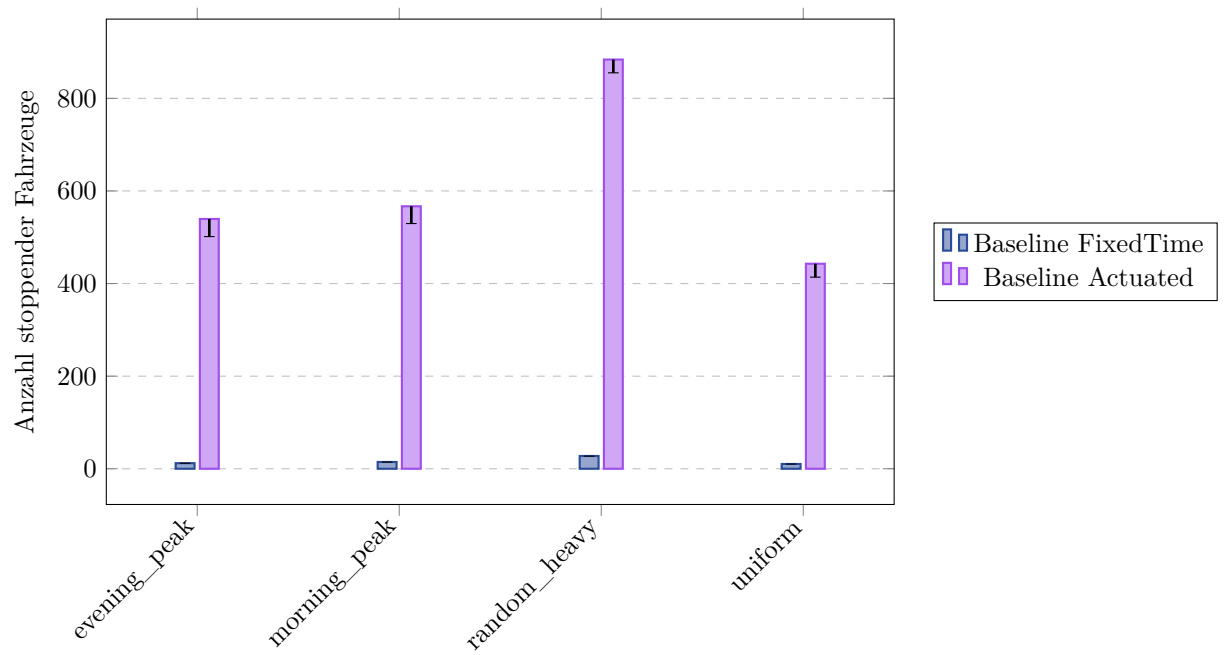


Abbildung 12: Anzahl stoppender Fahrzeuge

Ein besonders klarer Vorteil der RL-Modelle zeigt sich bei der Anzahl der stoppenden Fahrzeuge. Über alle Szenarien hinweg konnten die Modelle den Wert im Vergleich zur Fixed-Time-Baseline drastisch reduzieren. Teilweise stoppten Fahrzeuge nur in einem Fünftel der Fälle im Vergleich zur Baseline, was eine deutliche Verbesserung des Verkehrsflusses widerspiegelt. Dieser Effekt ist konsistent über alle Modelle und Szenarien hinweg nachweisbar.

5.1.3 Anzahl ankommender Fahrzeuge

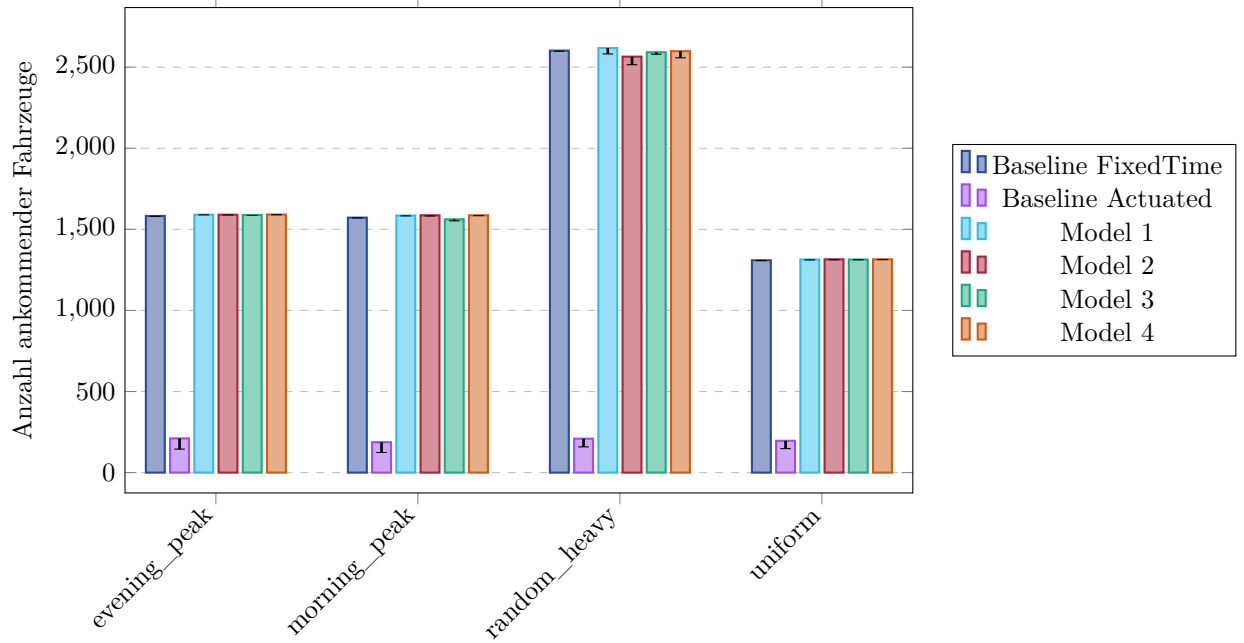


Abbildung 13: Anzahl ankommender Fahrzeuge

Hinsichtlich der Anzahl ankommender Fahrzeuge unterscheiden sich die RL-Modelle kaum von der Fixed-Time-Baseline. Alle Modelle erreichen nahezu identische Werte, was darauf hinweist, dass die Steuerungsstrategien trotz der reduzierten Warte- und Stoppzeiten keine signifikanten Auswirkungen auf die Durchsatzkapazität des Netzes haben. Einzig die Actuated-Baseline zeigt hier, wie auch in den anderen Metriken, ein deutlich schlechteres Abschneiden.

5.1.4 Durchschnitt fahrender Fahrzeuge

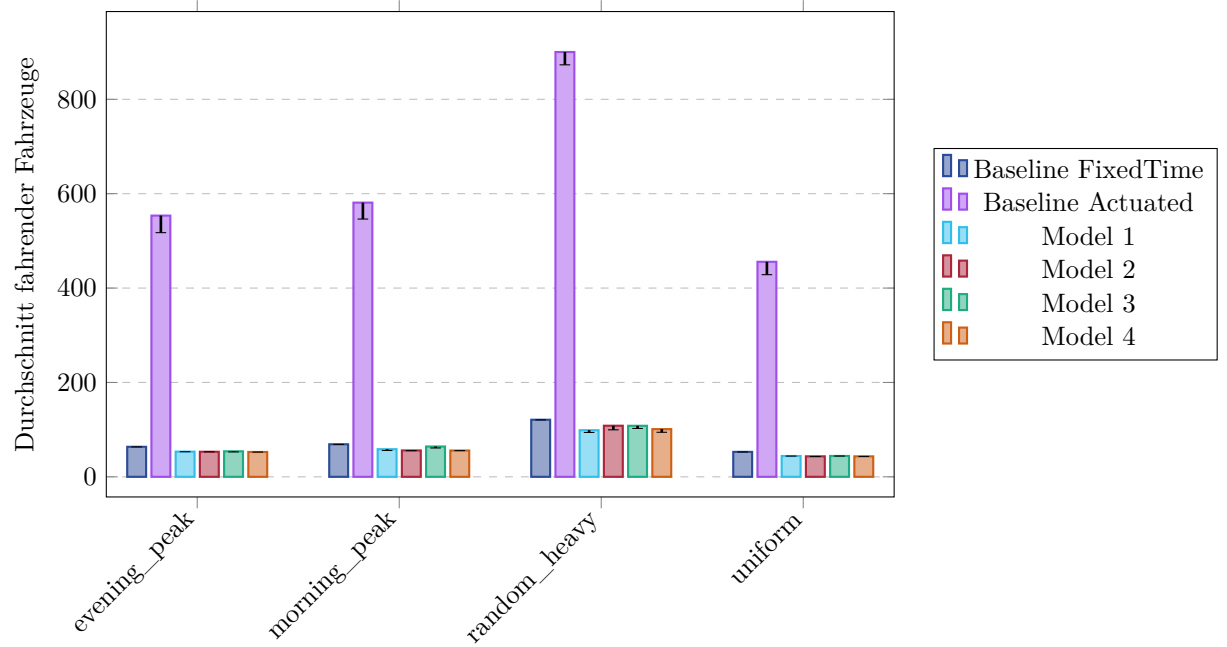


Abbildung 14: Durchschnitt fahrender Fahrzeuge

Bei der Betrachtung der durchschnittlich fahrenden Fahrzeuge wird ein konsistentes Muster sichtbar: In allen Szenarien liegen die Werte der RL-Modelle unterhalb der Fixed-Time-Baseline. Typischerweise beträgt der Unterschied etwa 10-15 Prozent, so dass bei einem Fixed-Time-Wert von 120 Fahrzeugen die RL-Modelle bei rund 105 Fahrzeugen liegen. Dies deutet darauf hin, dass die RL-Agenten den Verkehrsfluss stärker bündeln und damit die Anzahl gleichzeitig im Netz befindlicher Fahrzeuge reduzieren.

5.1.5 Durchschnittsgeschwindigkeiten

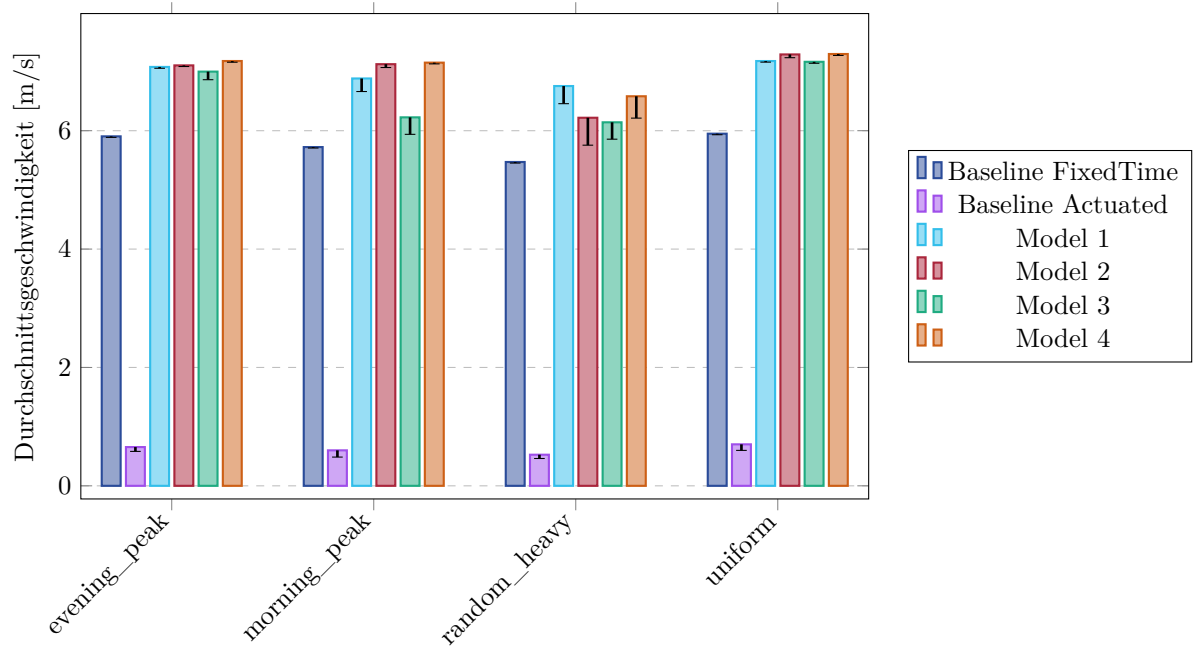


Abbildung 15: Durchschnittsgeschwindigkeiten

Die durchschnittliche Geschwindigkeit ist bei den RL-Modellen durchweg höher als bei der Fixed-Time-Baseline. Im Mittel beträgt der Vorsprung rund ein Sechstel, was eine signifikante Verbesserung der Reisegeschwindigkeit darstellt. Dieser Befund unterstreicht, dass die Modelle in der Lage sind, die Stoppzeiten und Stausituationen effektiv zu reduzieren und damit einen flüssigeren Verkehrsfluss zu ermöglichen.

5.1.6 Anzahl teleportierender Fahrzeuge

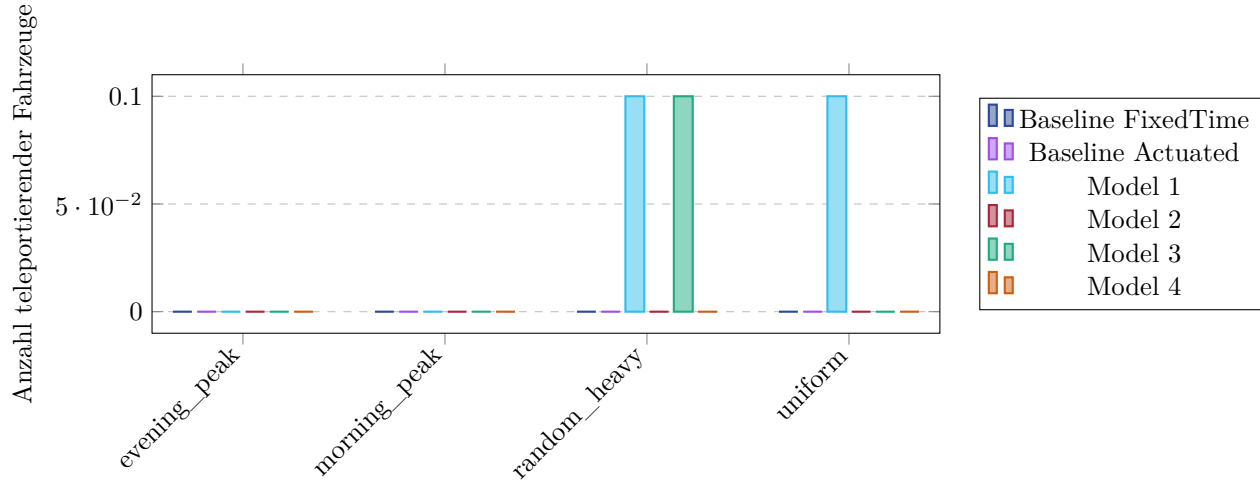


Abbildung 16: Anzahl teleportierender Fahrzeuge

Ein Sonderfall ergibt sich bei der Metrik der Teleportationen: Während in fast allen Szenarien keine Teleportationen auftraten, kam es in einzelnen Episoden zu vereinzelten Fällen. Konkret traten bei Modell 3 und Modell 1 im Szenario `random_heavy` sowie bei Modell 1 im Szenario `uniform` jeweils einzelne Teleportationen auf. Angesichts der insgesamt geringen Häufigkeit lassen sich diese Ereignisse als Ausnahmen werten, die die Gesamtbewertung der Modelle kaum beeinflussen.

5.1.7 Anzahl zurückgehaltener Fahrzeuge

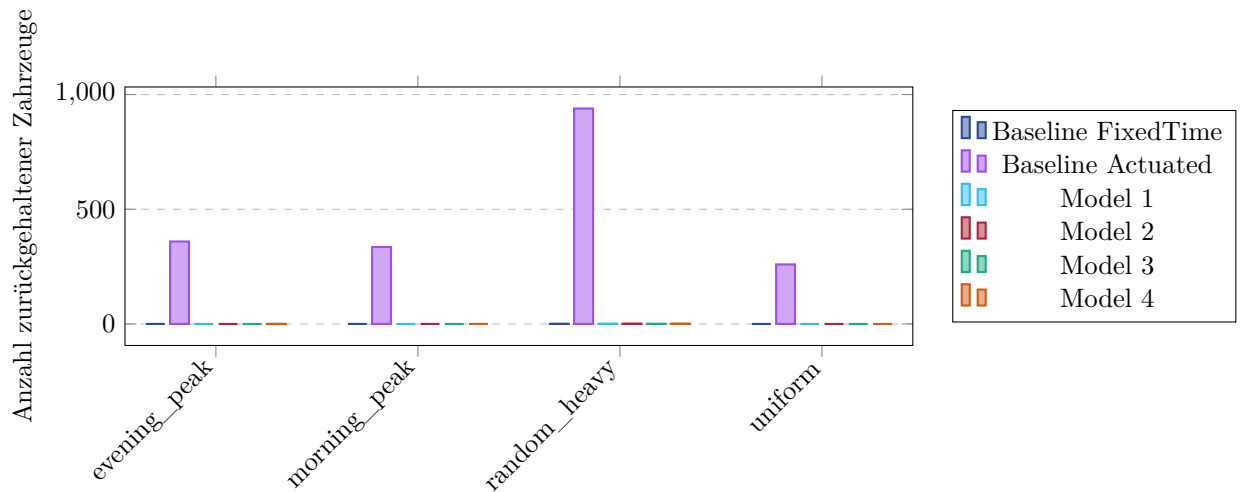


Abbildung 17: Anzahl zurückgehaltener Fahrzeuge

Es ist klar erkennbar, dass das Netz nicht an ihre maximal Auslastung gerät. Dies bedeutet jedoch nicht, dass keine Staus oder überlastet Teilgebiete gibt.

5.1.8 Einstufung

Zusammenfassend lässt sich feststellen, dass die RL-Modelle in nahezu allen betrachteten Metriken eine deutliche Überlegenheit gegenüber den Baselines, insbesondere gegenüber der Actuated-Steuerung, aufweisen. Während die Anzahl ankommender Fahrzeuge weitgehend unverändert blieb, konnten die RL-Ansätze die mittlere Wartezeit in mehreren Szenarien spürbar reduzieren und zugleich die Anzahl stoppender Fahrzeuge drastisch senken. Dies schlägt sich auch in einer höheren durchschnittlichen Reisegeschwindigkeit nieder, die bei den RL-Modellen im Mittel rund ein Sechstel über der Fixed-Time-Baseline liegt. Die Anzahl gleichzeitig im Netz befindlicher Fahrzeuge war bei den RL-Modellen konsistent geringer, was auf einen insgesamt flüssigeren Verkehrsfluss schließen lässt.

Auffällig ist jedoch, dass einzelne Modelle in spezifischen Szenarien (z. B. Modell 2 und 3 in `random_heavy` sowie Modell 3 in `morning_peak`) vergleichsweise schwach abschnitten und dabei hohe Varianzen aufwiesen. Diese Ausreißer mindern jedoch nicht die generelle Tendenz, dass die RL-Ansätze eine robuste und konsistente Verbesserung gegenüber den statischen Baselines darstellen.

Die Baseline Fixed-Time erweist sich im direkten Vergleich als deutlich konkurrenzfähiger als die Actuated-Variante, die über alle Metriken hinweg das schwächste Ergebnis liefert. In der Gesamtbetrachtung lässt sich daher schlussfolgern, dass RL-Agenten in der Lage sind, klassische Steuerungsverfahren zu übertreffen, indem sie Wartezeiten reduzieren, den Verkehrsfluss stabilisieren und die Effizienz des Gesamtsystems verbessern, auch wenn in Lastspitzen-Szenarien teilweise noch Optimierungspotenzial besteht.

5.2 Reward: Queue

Die zweite Gruppe zielt auf die Minimierung der Warteschlangenlänge.

5.2.1 Mittlere Wartezeiten

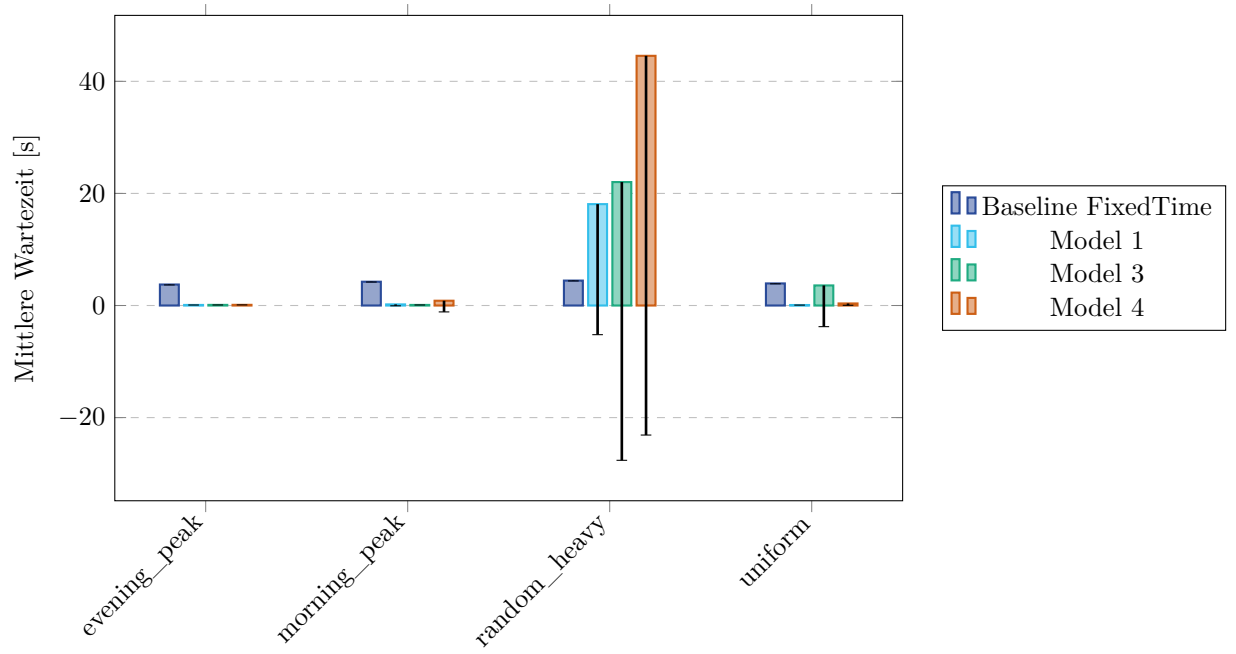


Abbildung 18: Mittlere Wartezeiten

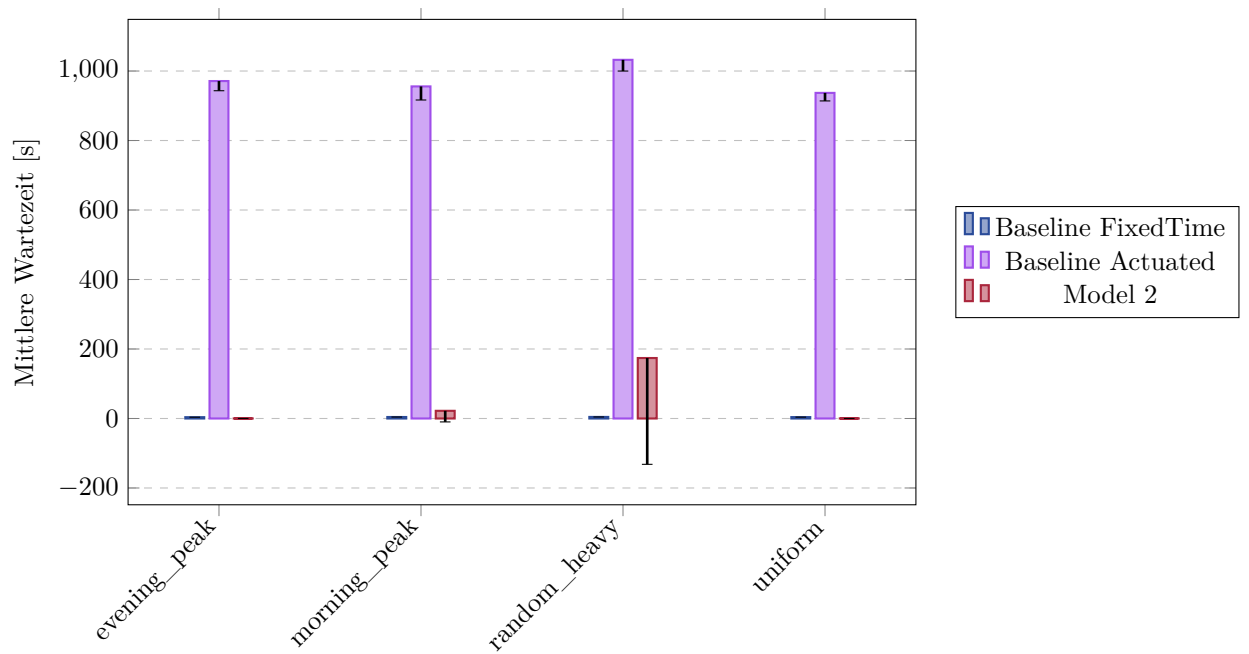


Abbildung 19: Mittlere Wartezeiten

Die Betrachtung der mittleren Wartezeit zeigt, dass die mit der Queue-Rewardfunktion trainierten Modelle in den meisten Szenarien sehr gute Ergebnisse erzielen. Insbesondere in `morning_peak`, `evening_peak` und `uniform` übertreffen die Modelle die Baselines deutlich. Eine Ausnahme bildet Modell 2, das im Szenario `morning_peak` eine auffällig schwache Performance aufweist. Zudem ist im Szenario `random_heavy` die Fixed-Time-Baseline den RL-Modellen überlegen, während alle Modelle in diesem Hochlastfall insgesamt schwache Ergebnisse erzielen.

5.2.2 Anzahl stoppender Fahrzeuge

In sumo werden Fahrzeuge die sich mit einer Geschwindigkeit kleiner als 0.1 m/s bewegt.

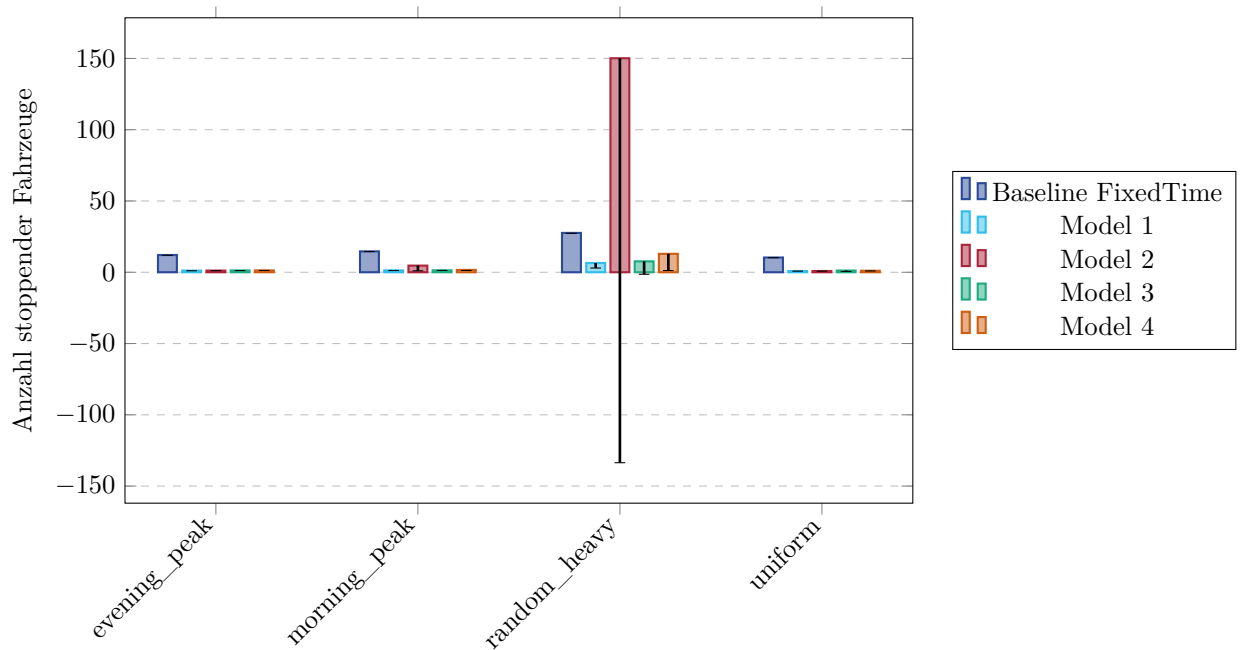


Abbildung 20: Anzahl stoppender Fahrzeuge

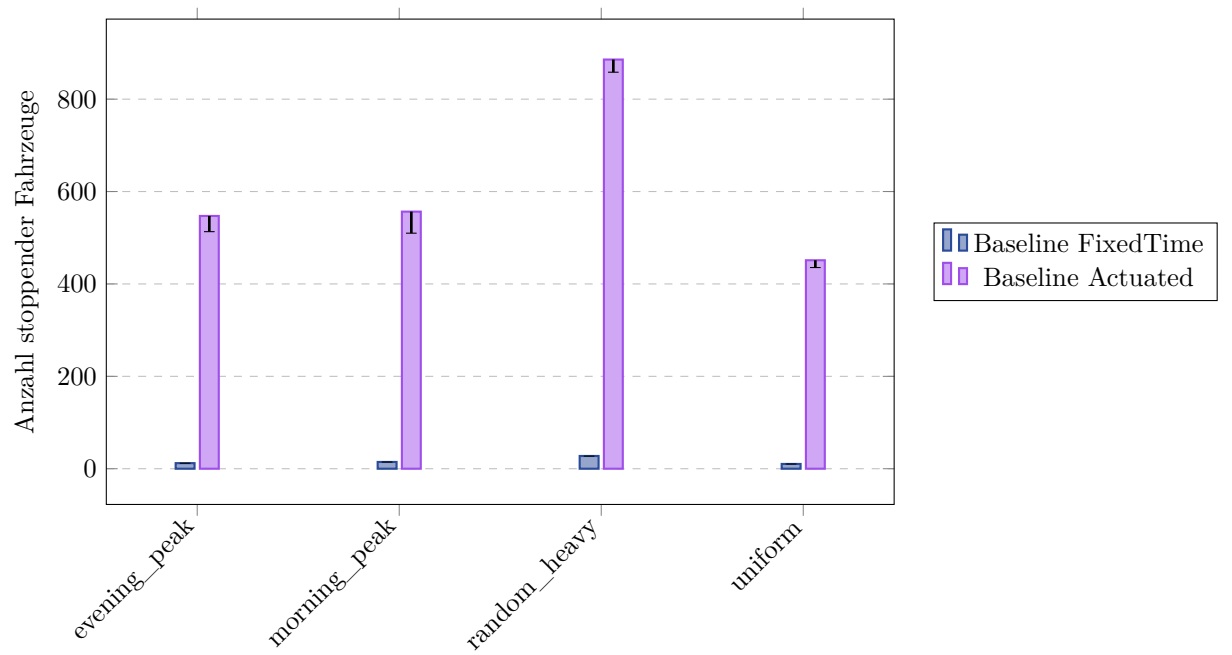


Abbildung 21: Anzahl stoppender Fahrzeuge

Auch bei der Anzahl stoppender Fahrzeuge schneiden die RL-Modelle durchweg deutlich besser ab als die Baselines. Die Werte liegen auf einem sehr niedrigen Niveau, teilweise fast bei null, was eine hohe Effizienz im Verkehrsfluss widerspiegelt. Erneut bildet Modell 2 im Szenario `random_heavy` eine Ausnahme, wo es im Vergleich zu den übrigen Modellen deutlich schlechter abschneidet. Die Actuated-Baseline bestätigt auch hier ihre schwache Gesamtperformance.

5.2.3 Anzahl ankommender Fahrzeuge

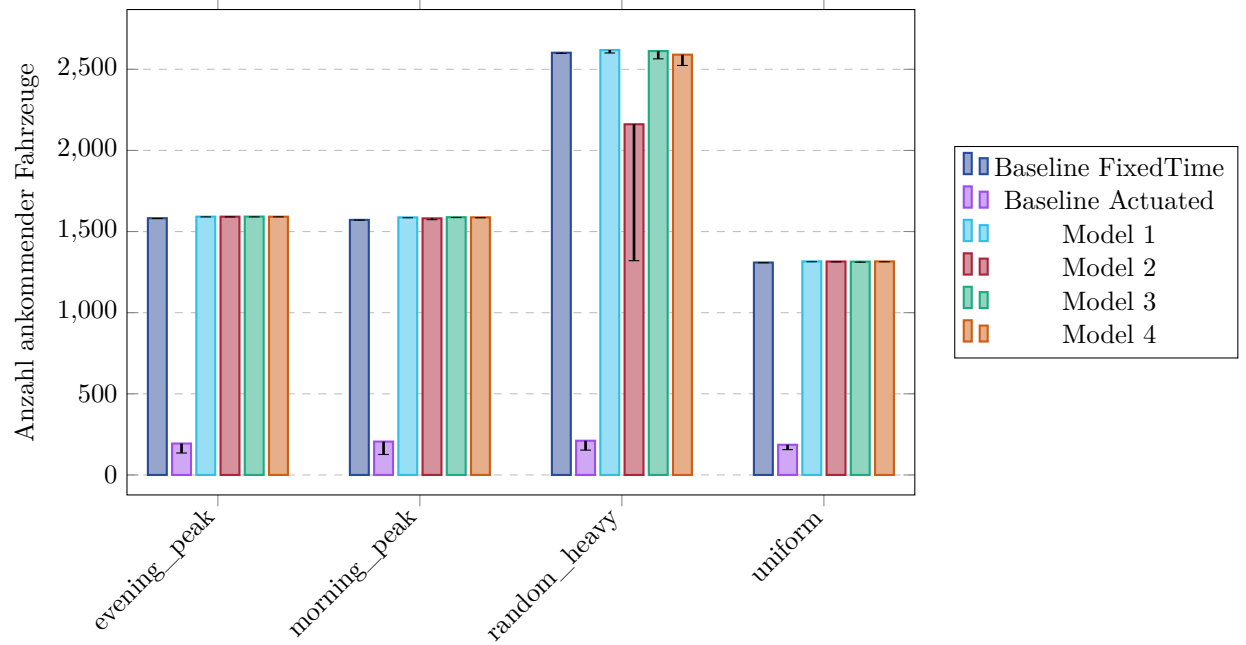


Abbildung 22: Anzahl ankommender Fahrzeuge

Hinsichtlich der Anzahl ankommender Fahrzeuge erreichen alle Modelle in nahezu allen Szenarien die maximale Durchsatzleistung. Lediglich Modell 2 weist im Szenario `random_heavy` leichte Defizite auf, was die zuvor beobachtete schwache Performance bestätigt. Abgesehen davon zeigen die RL-Modelle eine robuste und konsistente Leistungsfähigkeit.

5.2.4 Durchschnitt fahrender Fahrzeuge

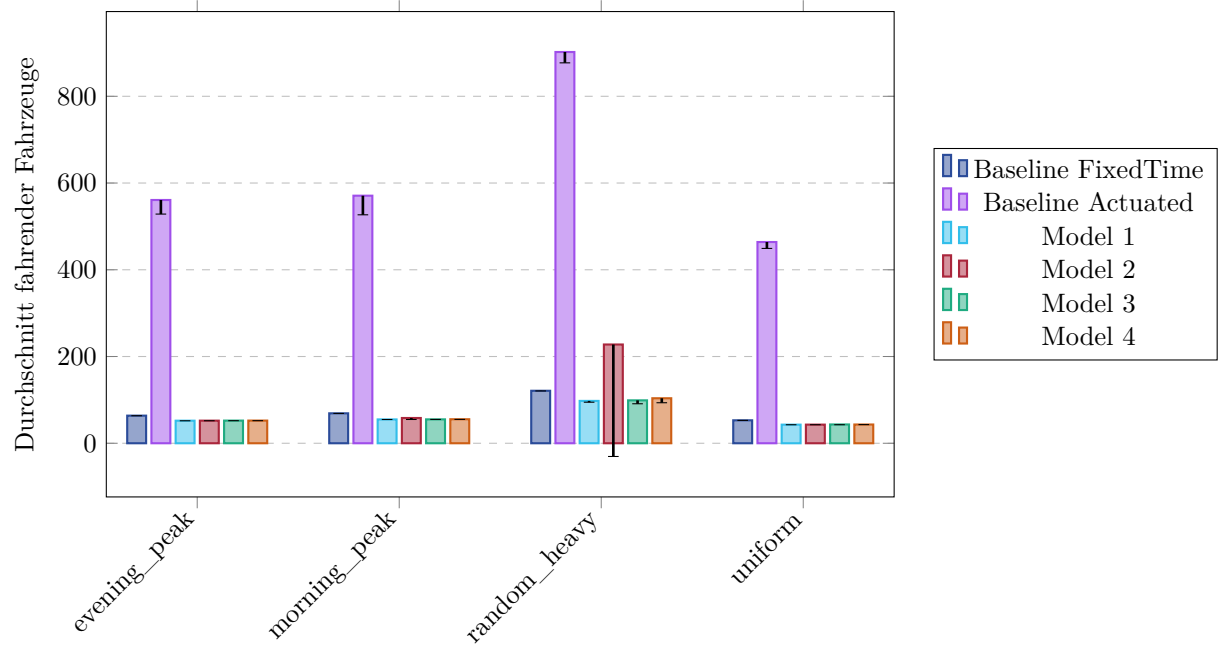


Abbildung 23: Durchschnitt fahrender Fahrzeuge

Bei der Betrachtung der durchschnittlich fahrenden Fahrzeuge zeigt sich ein differenziertes Bild: Die Actuated-Baseline weist überhöhte Werte auf, was ihre schwache Steuerungsleistung unterstreicht. Modell 2 fällt im Szenario `random_heavy` ebenfalls negativ auf und bestätigt so seine insgesamt schlechtere Performance. In allen anderen Szenarien schneiden die RL-Modelle besser ab als Fixed-Time, wobei die Unterschiede im Mittel bei etwa 10 Prozent liegen.

5.2.5 Durchschnittsgeschwindigkeiten

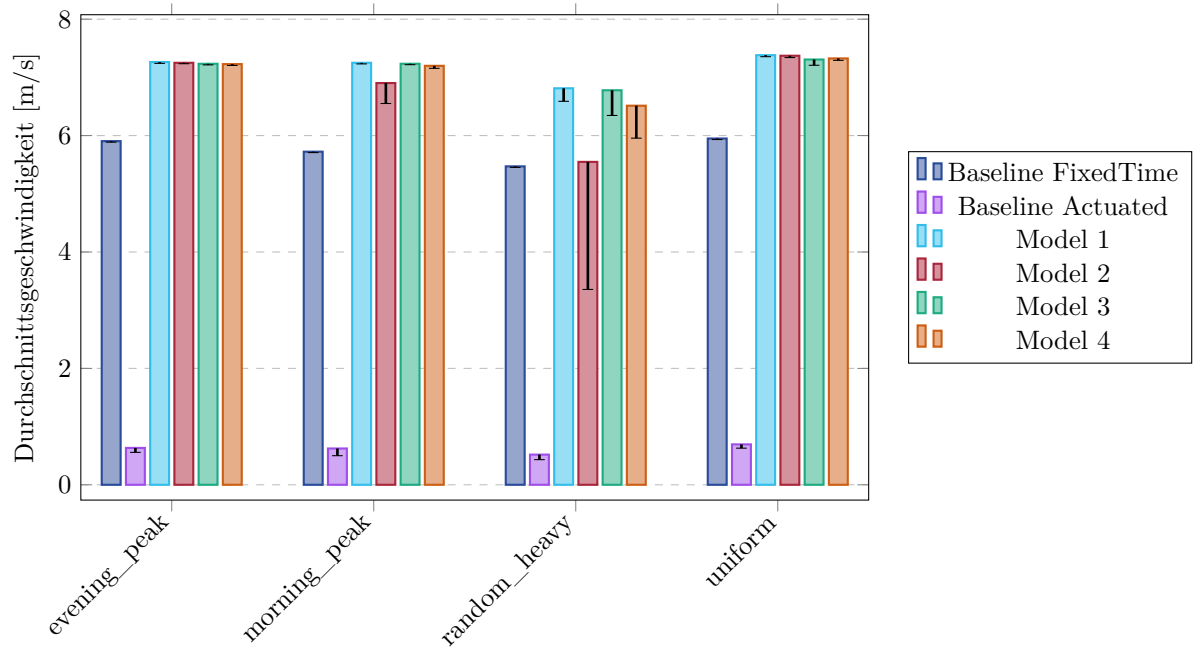


Abbildung 24: Durchschnittsgeschwindigkeiten

Ein besonders deutlicher Vorteil der RL-Modelle zeigt sich bei der durchschnittlichen Geschwindigkeit. Während Fixed-Time im Bereich von 5,5 bis 6 m/s liegt, erreichen die Modelle im Mittel Geschwindigkeiten von etwa 7 m/s über alle Szenarien hinweg. Dies deutet auf einen flüssigeren Verkehrsfluss und eine effiziente Reduzierung von Stopps und Wartezeiten hin. Die Actuated-Baseline schneidet auch hier am schlechtesten ab.

5.2.6 Anzahl teleportierender Fahrzeuge

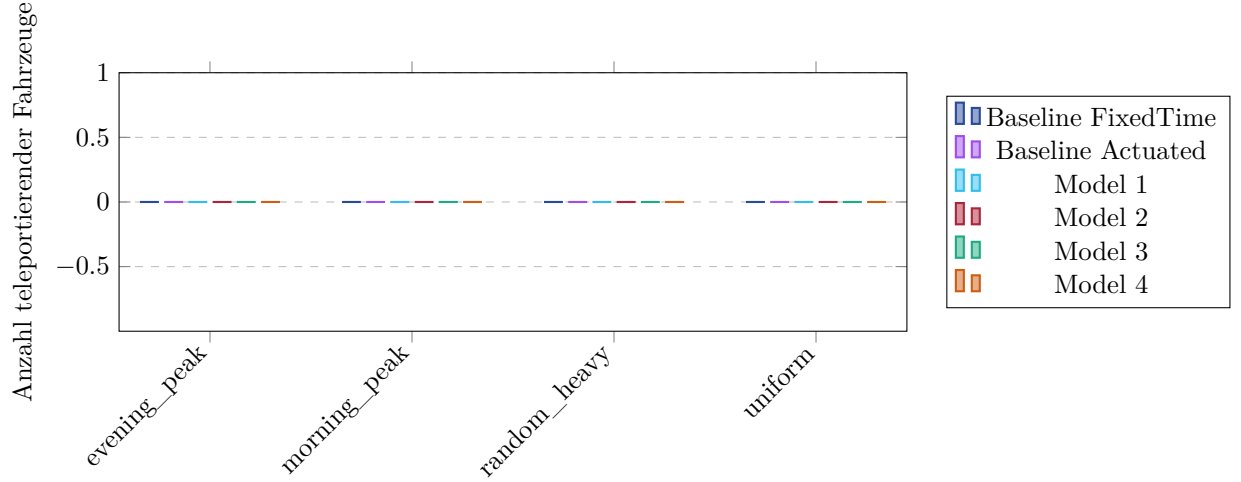


Abbildung 25: Anzahl teleportierender Fahrzeuge

Bei den Queue-basierten Modellen traten in keiner der Episoden Teleportationen auf. Damit konnte in allen Szenarien eine stabile und konsistente Steuerung ohne kritische Netzüberlastungen gewährleistet werden.

5.2.7 Anzahl zurückgehaltener Fahrzeuge

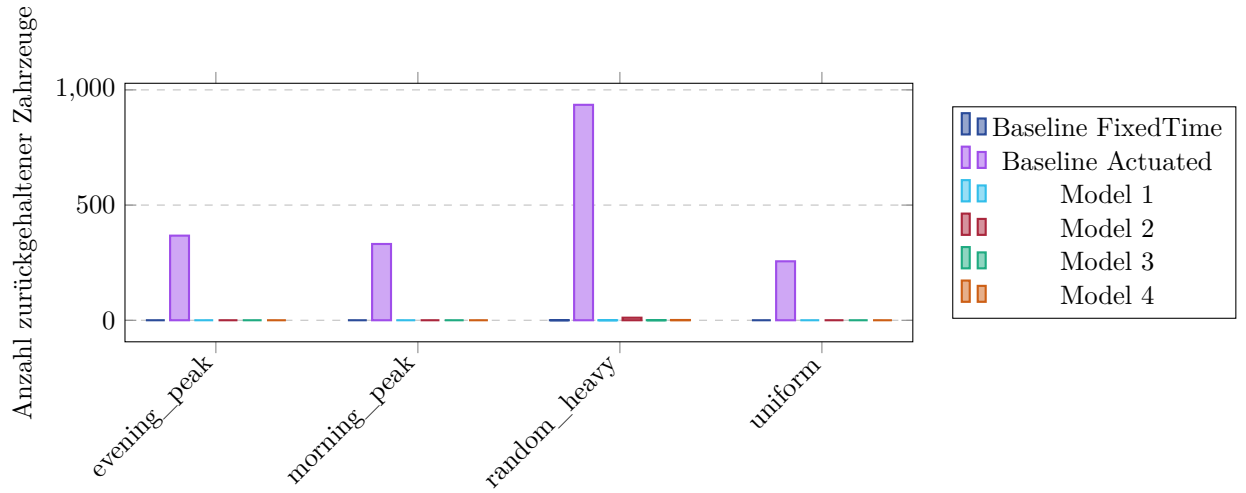


Abbildung 26: Anzahl zurückgehaltener Fahrzeuge

Die Metrik der zurückgehaltenen Fahrzeuge zeigt ebenfalls ein sehr positives Bild: Abgesehen von einem minimalen Rückstau bei Modell 2 im Szenario random_heavy treten bei keinem der Modelle relevante Rückstauereffekte auf. Lediglich die Actuated-Baseline verzeichnet regelmäßig hohe Werte, was ihre mangelnde Robustheit erneut bestätigt.

5.2.8 Einstufung

Die Evaluation der mit der Queue-Rewardfunktion trainierten Modelle zeigt insgesamt eine deutliche Überlegenheit gegenüber den Baselines, insbesondere gegenüber der Actuated-Steuerung. In den meisten Szenarien konnten die Modelle die mittlere Wartezeit erheblich senken, die Anzahl stoppender Fahrzeuge nahezu eliminieren und die durchschnittliche Reisegeschwindigkeit signifikant steigern. Zudem wurde in allen Szenarien eine maximale Anzahl ankommender Fahrzeuge erreicht, was auf eine hohe Durchsatzleistung hindeutet. Auch hinsichtlich zurückgehaltener Fahrzeuge und Teleportationen zeigen die Modelle eine robuste Performance, da in nahezu allen Fällen keine nennenswerten Rückstaueffekte oder Teleportationen auftraten.

Gleichzeitig offenbart die Analyse Schwächen in spezifischen Szenarien: Vor allem Modell 2 zeigt im Szenario `random_heavy` (und in geringerem Maße auch in `morning_peak`) eine deutlich schlechtere Performance, die sich in höheren Wartezeiten, mehr Stopps sowie leicht reduzierten Ankunftszahlen widerspiegelt. Dennoch bleibt das Gesamtbild klar positiv, da die übrigen Modelle konsistent gute Ergebnisse liefern und die Fixed-Time-Baseline in den meisten Szenarien übertreffen.

Insgesamt belegt die Auswertung, dass Queue-basierte Rewards einen robusten Ansatz für die Verkehrsflussoptimierung darstellen, der Wartezeiten und Rückstau effektiv reduziert und dabei eine hohe Systemstabilität gewährleistet.

5.3 Reward: Reale Welt

Die dritte Gruppe zielt auf ein kombiniertes Minimieren der Wartezeiten, Anzahl an Phasenwechsel und Staus.

5.3.1 Mittlere Wartezeiten

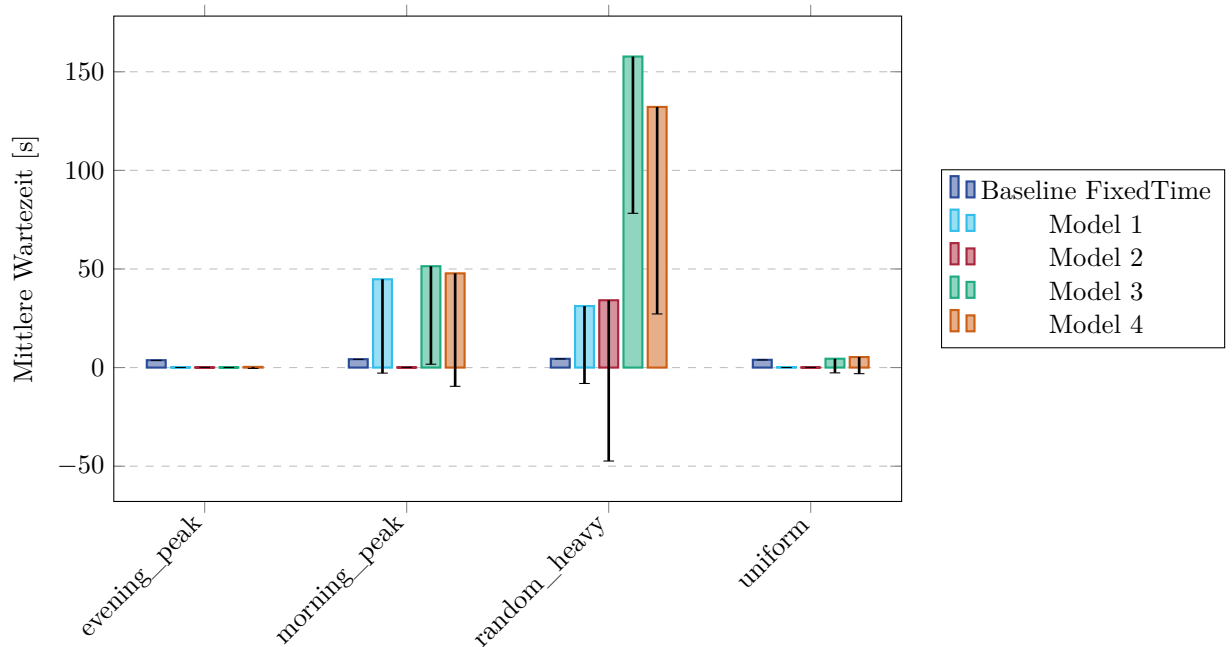


Abbildung 27: Mittlere Wartezeiten

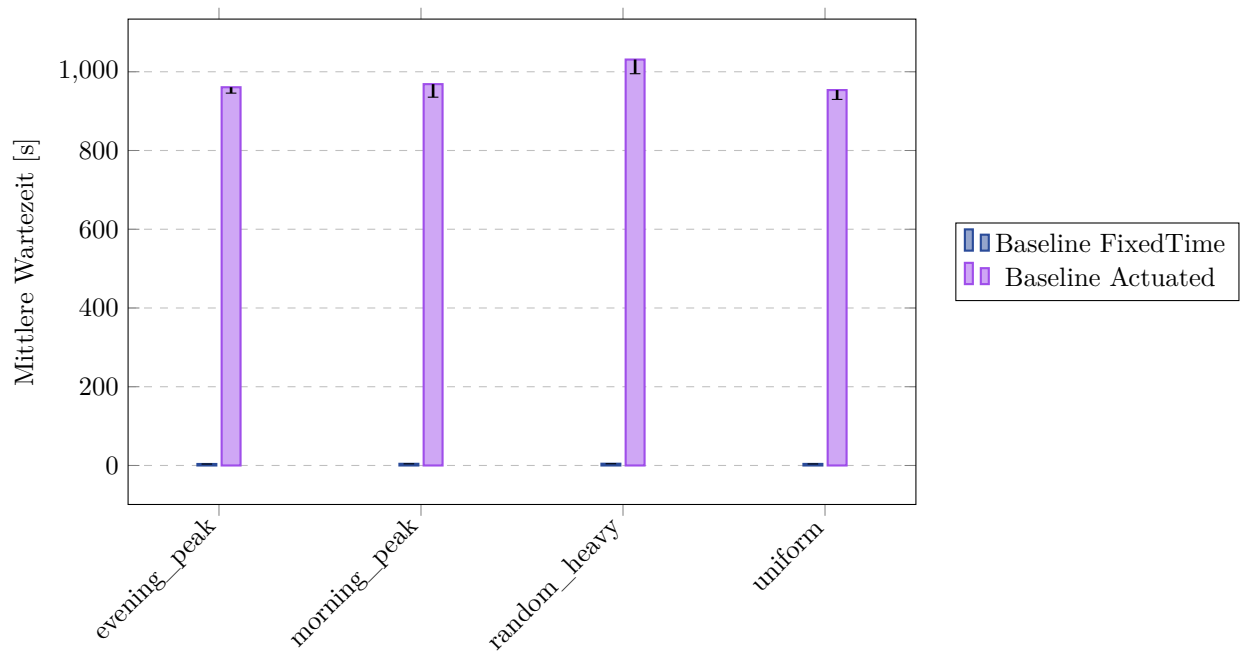


Abbildung 28: Mittlere Wartezeiten

5.3.2 Anzahl stoppender Fahrzeuge

In sumo werden Fahrzeuge die sich mit einer Geschwindigkeit kleiner als 0.1 m/s bewegt.

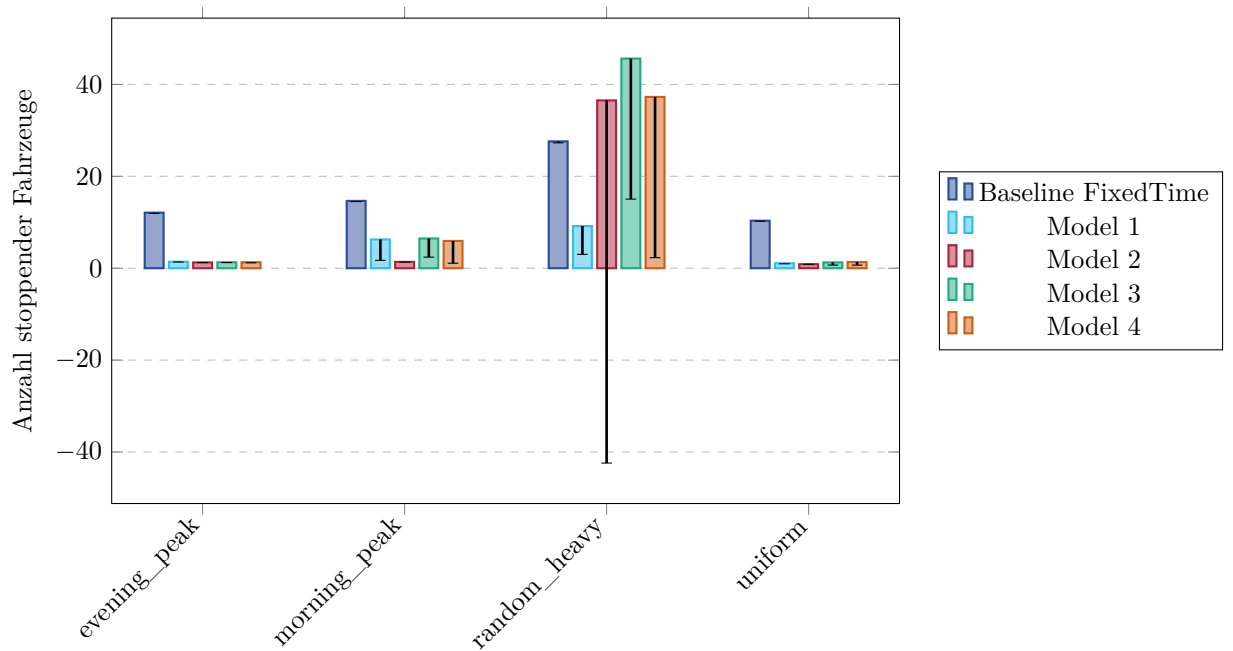


Abbildung 29: Anzahl stoppender Fahrzeuge

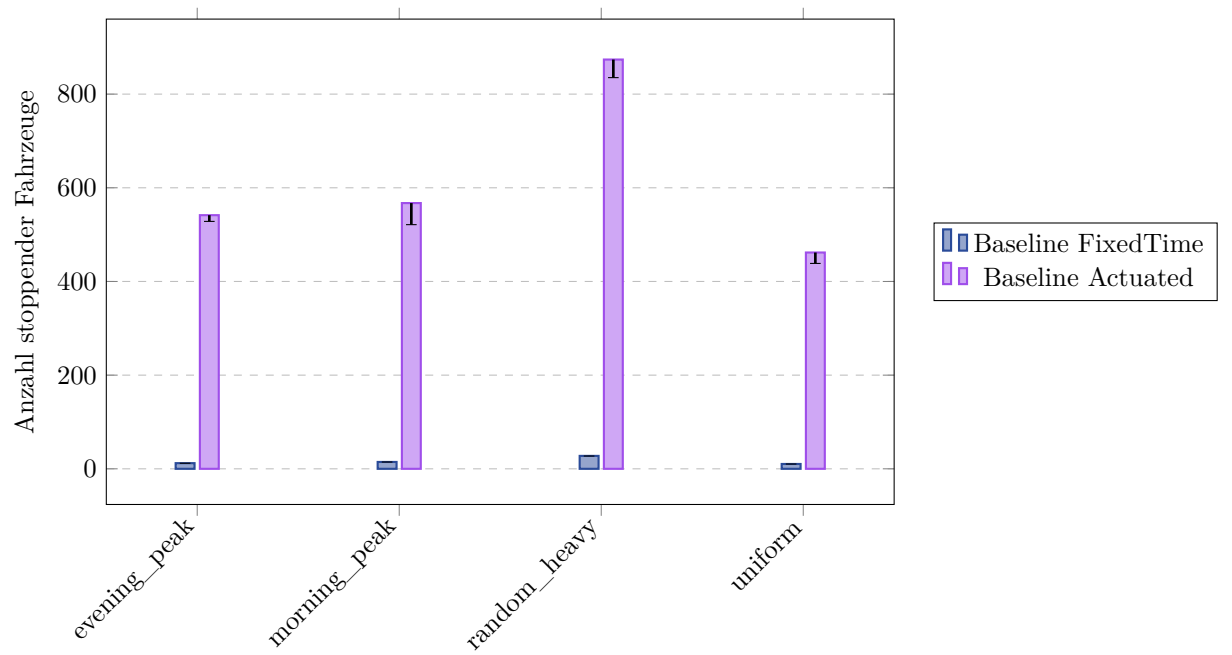


Abbildung 30: Anzahl stoppender Fahrzeuge

5.3.3 Anzahl ankommender Fahrzeuge

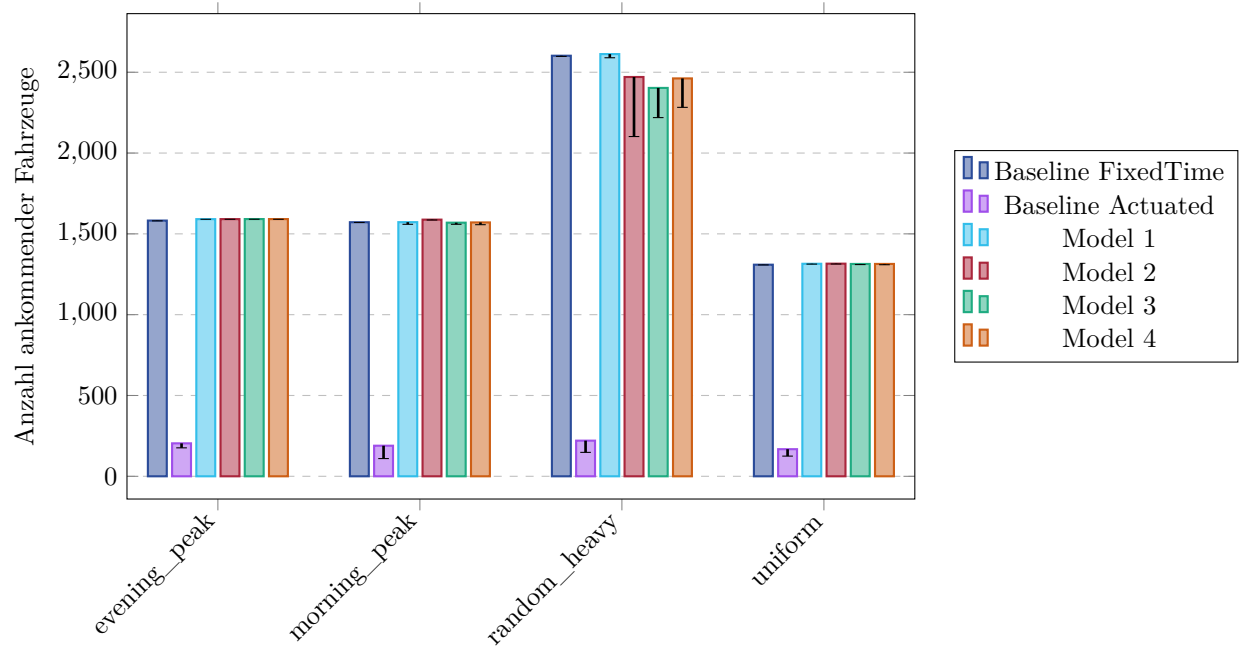


Abbildung 31: Anzahl ankommender Fahrzeuge

5.3.4 Durchschnitt fahrender Fahrzeuge

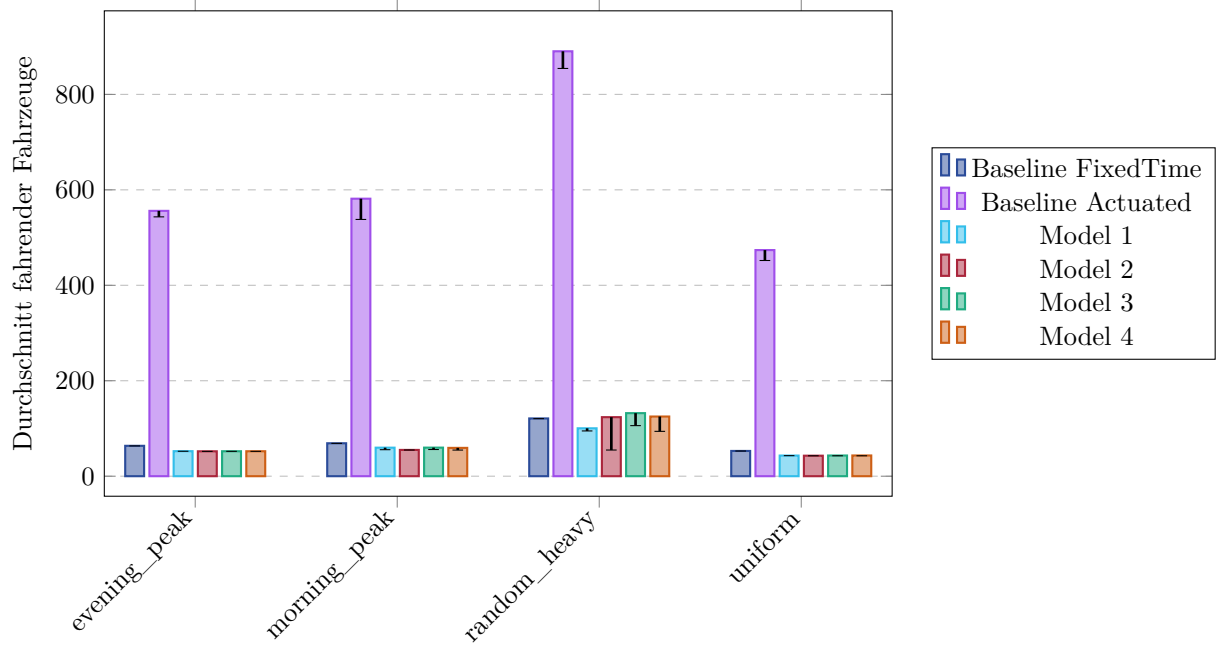


Abbildung 32: Durchschnitt fahrender Fahrzeuge

5.3.5 Durchschnittsgeschwindigkeiten

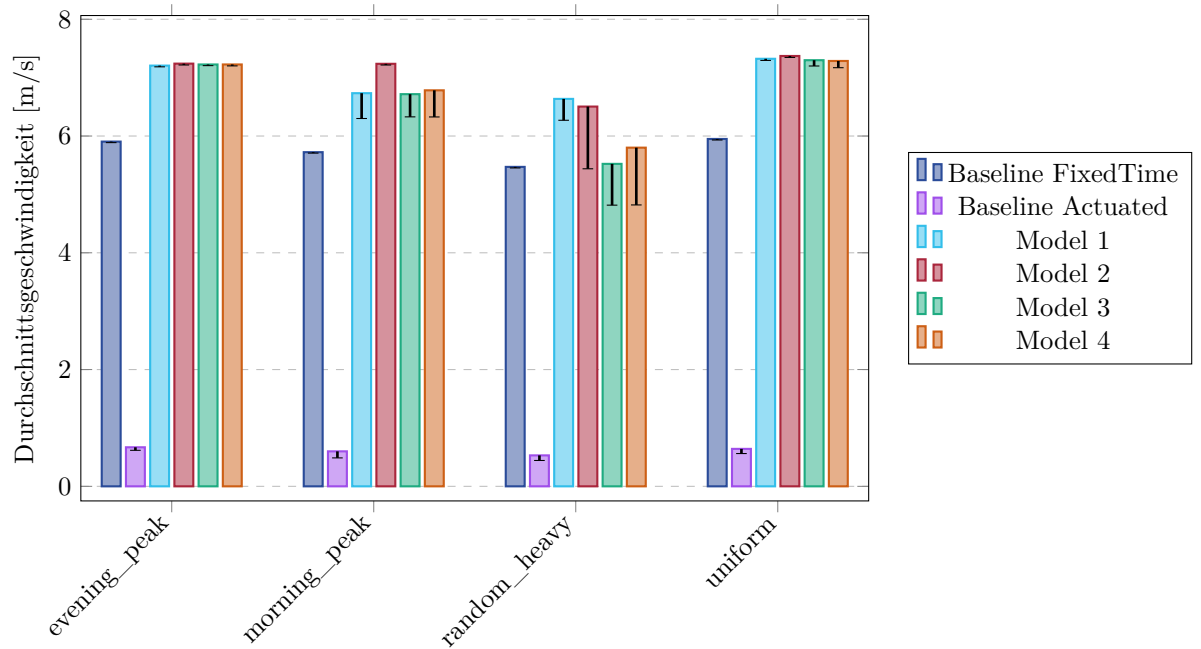


Abbildung 33: Durchschnittsgeschwindigkeiten

5.3.6 Anzahl teleportierender Fahrzeuge

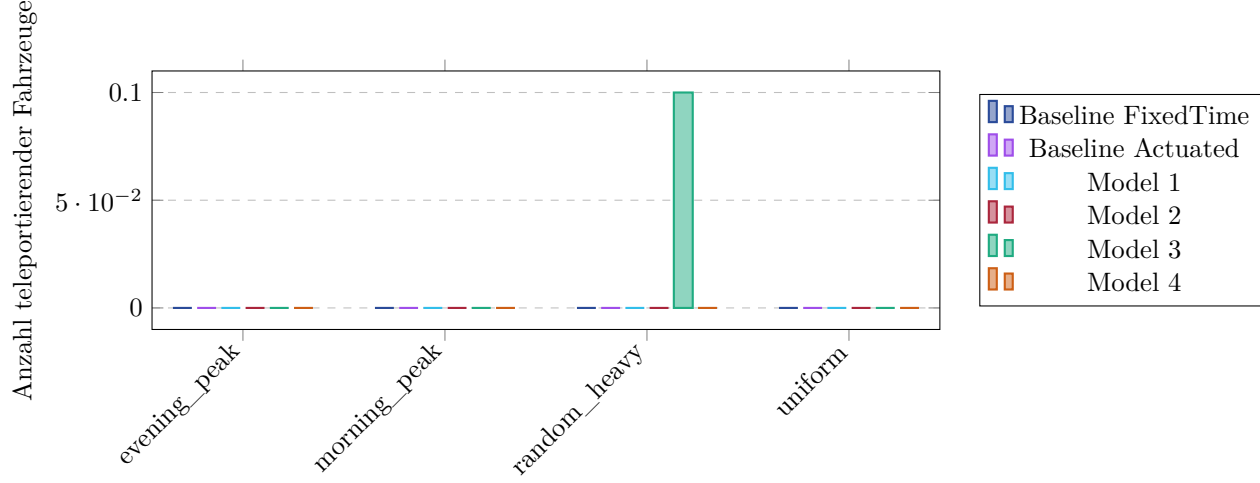


Abbildung 34: Anzahl teleportierender Fahrzeuge

5.3.7 Anzahl zurückgehaltener Fahrzeuge

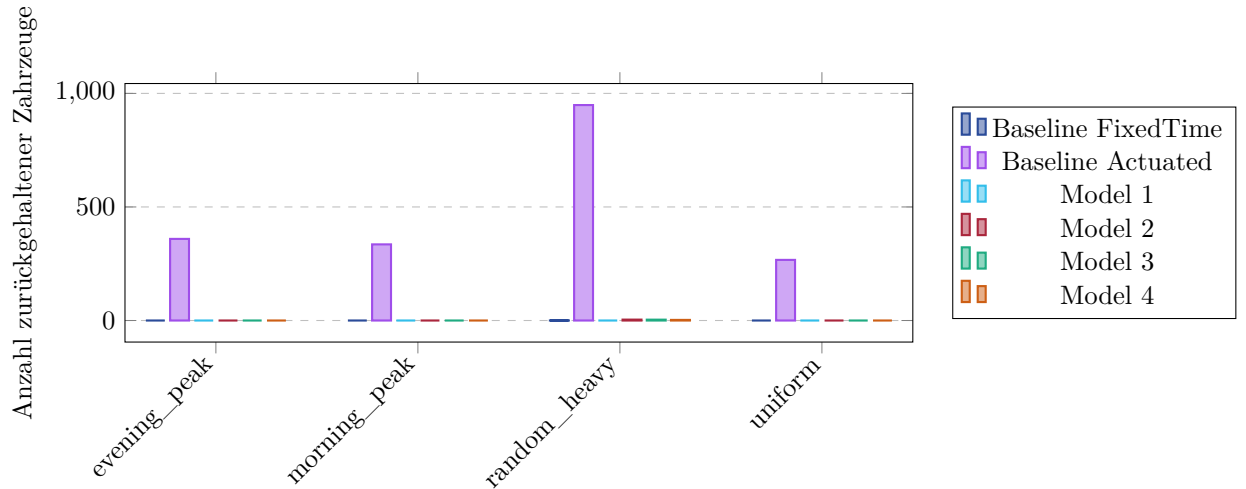


Abbildung 35: Anzahl zurückgehaltener Fahrzeuge

5.3.8 Einstufung

5.4 Reward: CO₂-Emissionen

Die vierte Gruppe zielt auf eine Minimierung der CO₂-Emissionen, während gleichzeitig versucht wird die Stabilität aufrecht zu erhalten. Für diese Emissions-basierten Modelle wird zusätzlich die CO₂-Emission als zentrale Metrik betrachtet.

5.4.1 CO₂-Emissionen

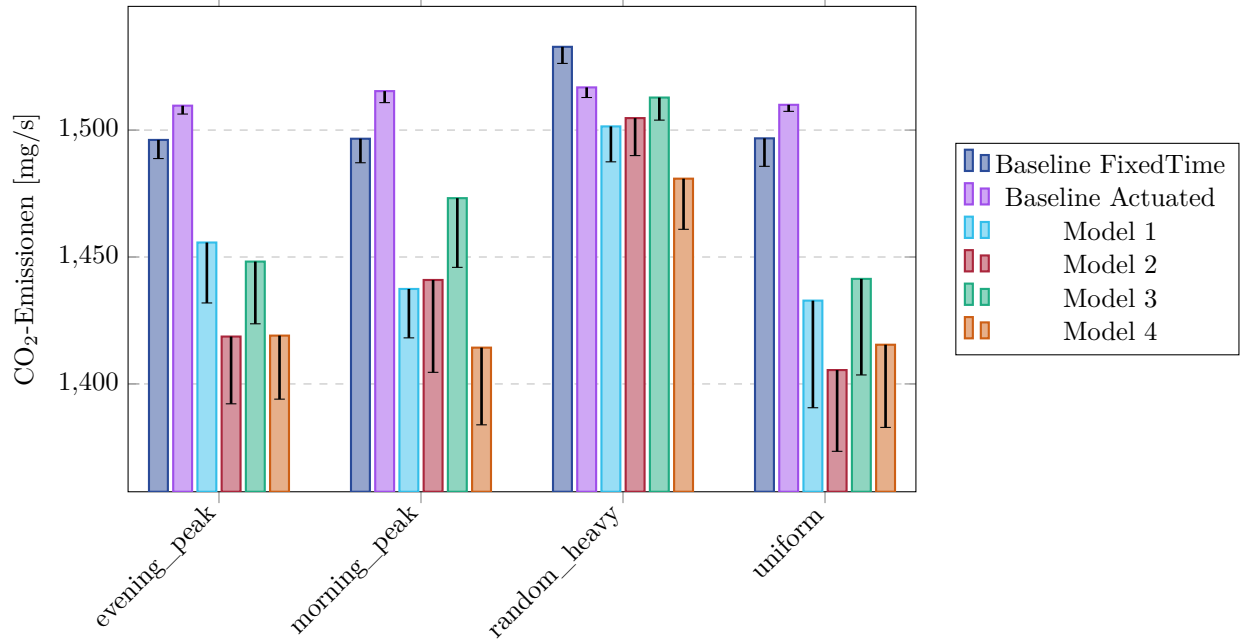


Abbildung 36: CO₂-Emissionen

Die Analyse der CO₂-Emissionen verdeutlicht die Vorteile der RL-Modelle gegenüber den Baselines. Während die Baselines in allen Szenarien relativ konstante Werte zwischen 1.490 und 1.560 mg/s aufweisen, liegen die Modelle fast durchweg darunter. Besonders in den Szenarien uniform, morning_peak und evening_peak erreichen die Modelle signifikant niedrigere Emissionen.

Konkret bewegen sich die Baselines im Bereich von 1.496 mg/s (Fixed-Time, uniform) bis 1.532 mg/s (Fixed-Time, random_heavy). Die Modelle erzielen dagegen Werte von 1.405 mg/s (Modell 2, uniform) bis 1.501 mg/s (Modell 1, random_heavy). Auffällig ist, dass in random_heavy alle Modelle zwar höhere Emissionen verzeichnen, diese jedoch weiterhin im Bereich oder leicht unterhalb der Baselines liegen.

Im Mittel schneiden Modell 2 und Modell 4 besonders positiv ab und zeigen in fast allen Szenarien die niedrigsten Emissionen. Damit belegt die Auswertung, dass die Emissions-basierten RL-Agenten in der Lage sind, die Gesamtemissionen signifikant zu reduzieren, ohne die Stabilität des Systems wesentlich zu beeinträchtigen.

5.4.2 Mittlere Wartezeiten

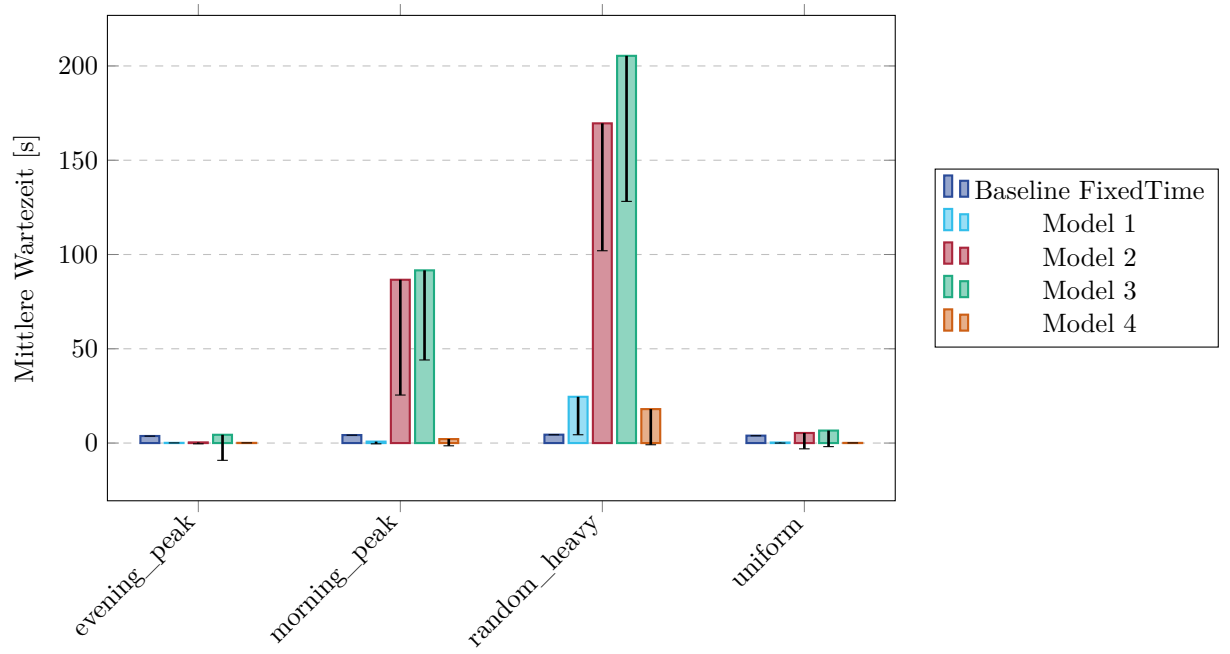


Abbildung 37: Mittlere Wartezeiten

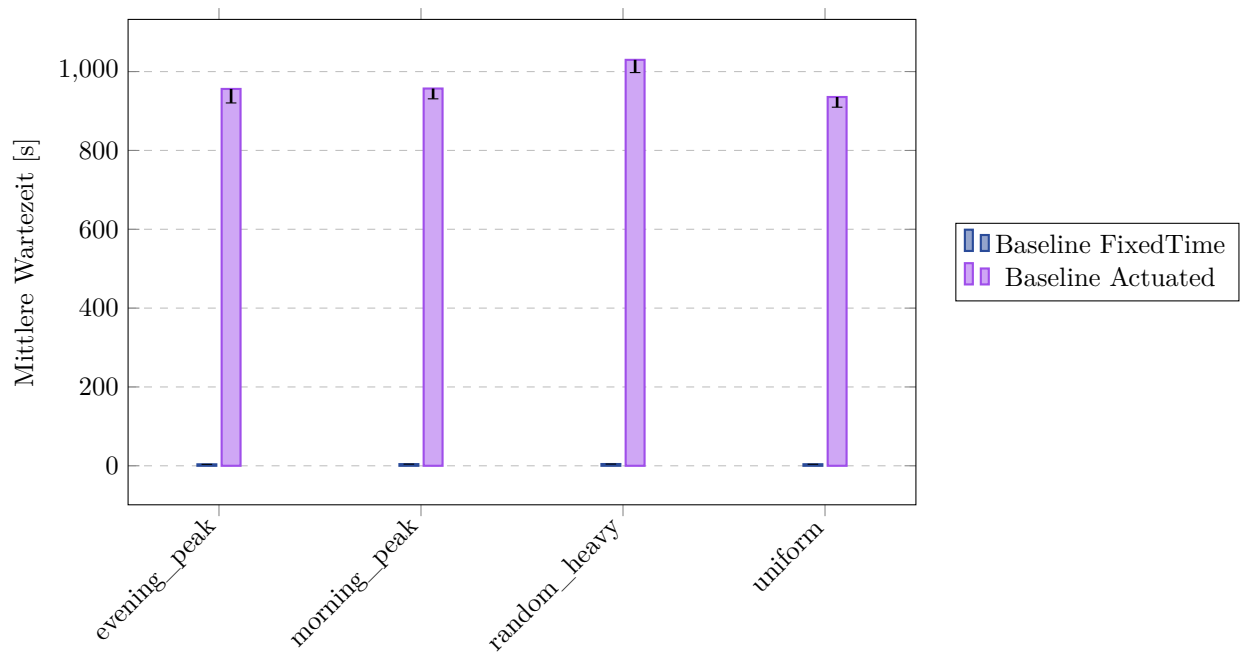


Abbildung 38: Mittlere Wartezeiten

Die Analyse der mittleren Wartezeit unterstreicht den deutlichen Vorteil der RL-Modelle gegenüber den Baselines - mit Ausnahme bestimmter Hochlastszenarien. Die Fixed-Time-Baseline liegt in allen Szenarien bei soliden, aber nicht optimalen Werten zwischen 3,7 und 4,4 Sekunden, während die Actuated-Baseline erneut massiv schlechter abschneidet (zwischen 935 und 1 029 Sekunden).

Die RL-Modelle zeigen in `evening_peak` und `uniform` hervorragende Ergebnisse: Modell 4 erreicht hier mit 0,4 Sekunden extrem niedrige Werte, während auch Modell 1 mit 1,8 Sekunden (`evening_peak`) bzw. 12,7 Sekunden (`uniform`) deutlich besser als die Baselines performt. Modell 2 und Modell 3 fallen dagegen negativ auf: Beide Modelle weisen in `morning_peak` und vor allem in `random_heavy` drastisch erhöhte Wartezeiten auf. Mit bis zu 26 745 Sekunden bei Modell 3 im Hochlastszenario überschreiten sie die Baselines deutlich und zeigen starke Instabilität.

Insgesamt lässt sich feststellen, dass die Emissions-basierten Modelle in Szenarien mit normaler Last die Wartezeiten signifikant reduzieren können, in Hochlastsituationen jedoch teilweise versagen und dadurch ihre Vorteile verlieren.

5.4.3 Anzahl stoppender Fahrzeuge

In sumo werden Fahrzeuge die sich mit einer Geschwindigkeit kleiner als 0.1 m/s bewegt.

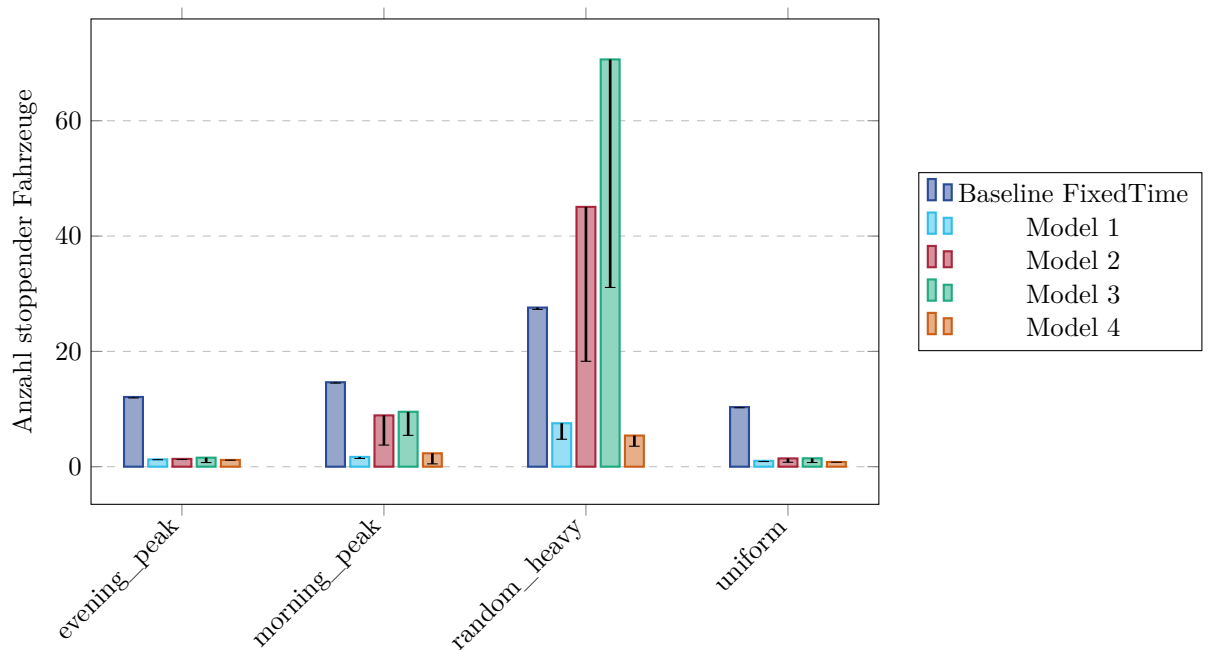


Abbildung 39: Anzahl stoppender Fahrzeuge

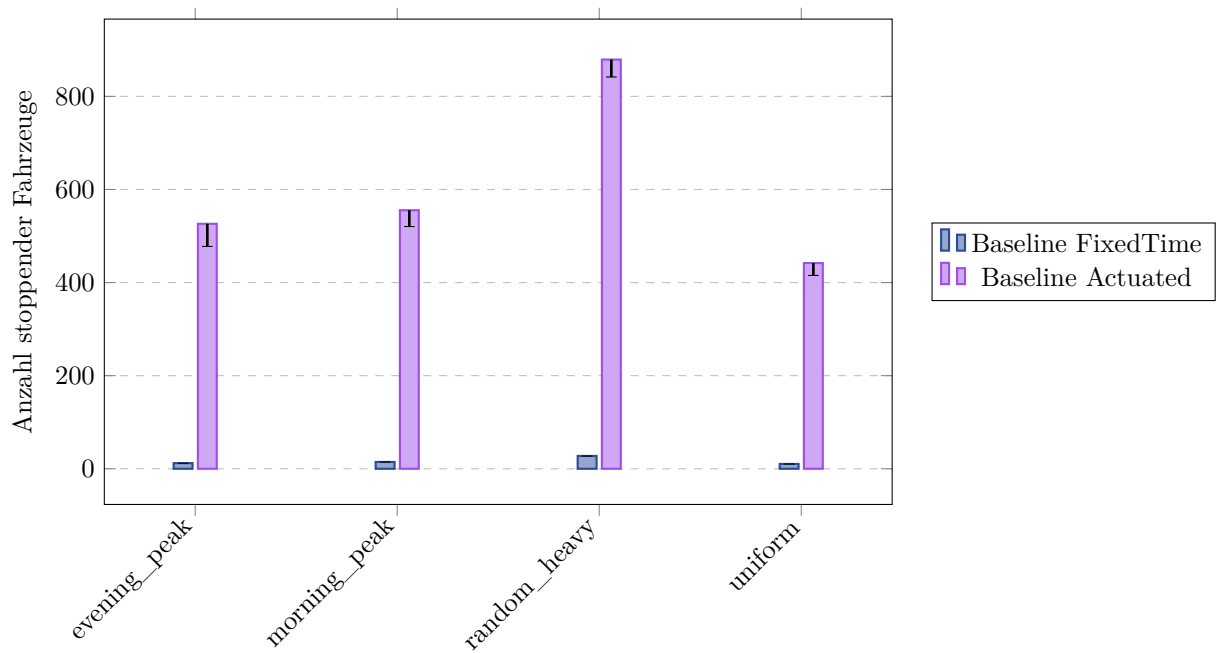


Abbildung 40: Anzahl stoppender Fahrzeuge

Bei der Metrik der stoppenden Fahrzeuge wird erneut ein klarer Unterschied zwischen Baselines und RL-Modellen sichtbar. Die Fixed-Time-Baseline erreicht mit Werten zwischen 10 und 27 stoppenden Fahrzeugen pro Szenario ein moderates Niveau, während die Actuated-Baseline mit extrem hohen Werten (z. B. 879 im `random_heavy`) durchgehend die schlechteste Performance zeigt.

Die RL-Modelle hingegen weisen in den meisten Szenarien nur sehr wenige Stopps auf. Besonders Modell 1 und Modell 4 schneiden konsistent am besten ab und liegen teilweise bei nahezu null Stopps (z. B. 1,0-1,2 Fahrzeuge in `uniform` bzw. `evening_peak`). Modell 2 und Modell 3 zeigen dagegen in `random_heavy` eine deutliche Schwäche: Mit 45 bzw. 70 stoppenden Fahrzeugen liegen ihre Werte hier deutlich über den übrigen RL-Modellen und sogar über Fixed-Time.

Insgesamt zeigt sich, dass die Emissions-basierten Modelle den Verkehrsfluss in normalen Lastszenarien äußerst effektiv stabilisieren, während in Hochlastsituationen (`random_heavy`) einzelne Modelle ihre Vorteile nicht aufrechterhalten können.

5.4.4 Anzahl ankommender Fahrzeuge

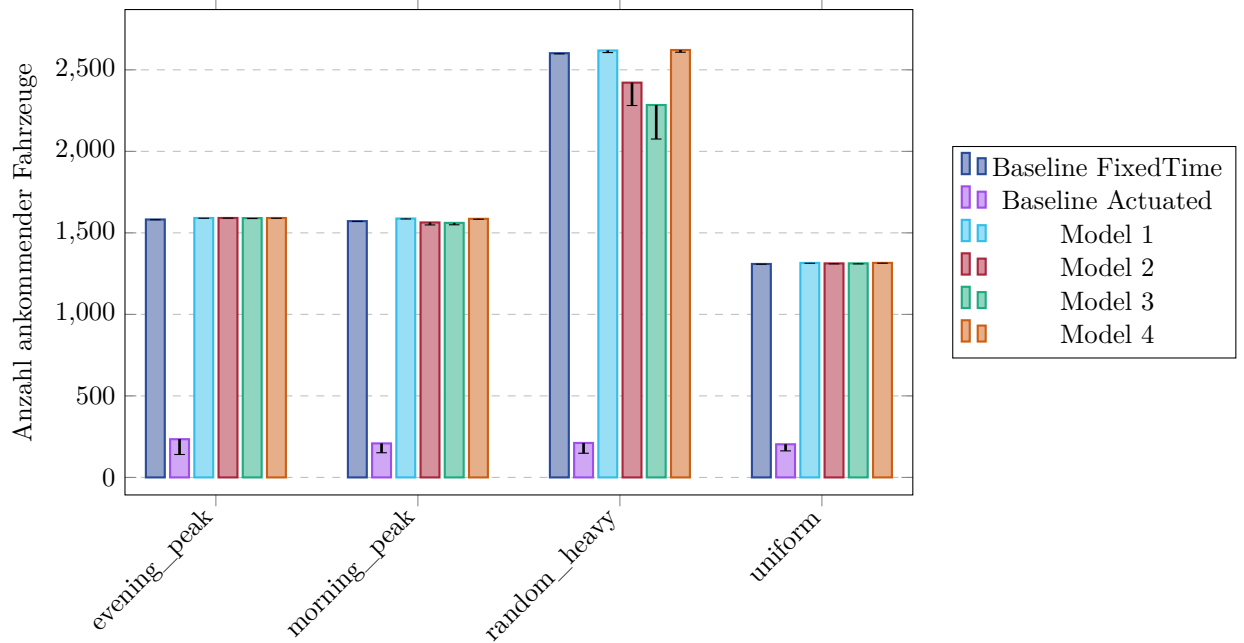


Abbildung 41: Anzahl ankommender Fahrzeuge

Bei der Anzahl ankommender Fahrzeuge erzielen die Emissions-basierten Modelle durchweg sehr gute Ergebnisse. In allen Szenarien erreichen nahezu alle Modelle die maximale Anzahl an Zielankünften, was auf einen stabilen Verkehrsfluss und eine hohe Netzkapazität hinweist. Die Fixed-Time-Baseline bestätigt dies ebenfalls mit konstanten Maximalwerten, während die Actuated-Baseline erneut deutlich schlechter abschneidet und in keinem Szenario die volle Ankunftsleistung erreicht.

Einzig im Szenario `random_heavy` zeigen Modell 2 und Modell 3 Schwächen: Mit 2 421 bzw. 2 284 ankommenden Fahrzeugen liegen sie unterhalb des Maximums von 2 602 Fahrzeugen (Fixed-Time und andere Modelle). Dieser Rückstand deckt sich mit den zuvor beobachteten Leistungseinbußen in Hochlastsituationen, insbesondere bei Wartezeiten und Stopps.

Insgesamt bestätigt die Metrik, dass die Emissions-basierten Modelle eine hohe Durchsatzleistung erzielen, mit Ausnahme vereinzelter Schwächen unter extremer Belastung.

5.4.5 Durchschnitt fahrender Fahrzeuge

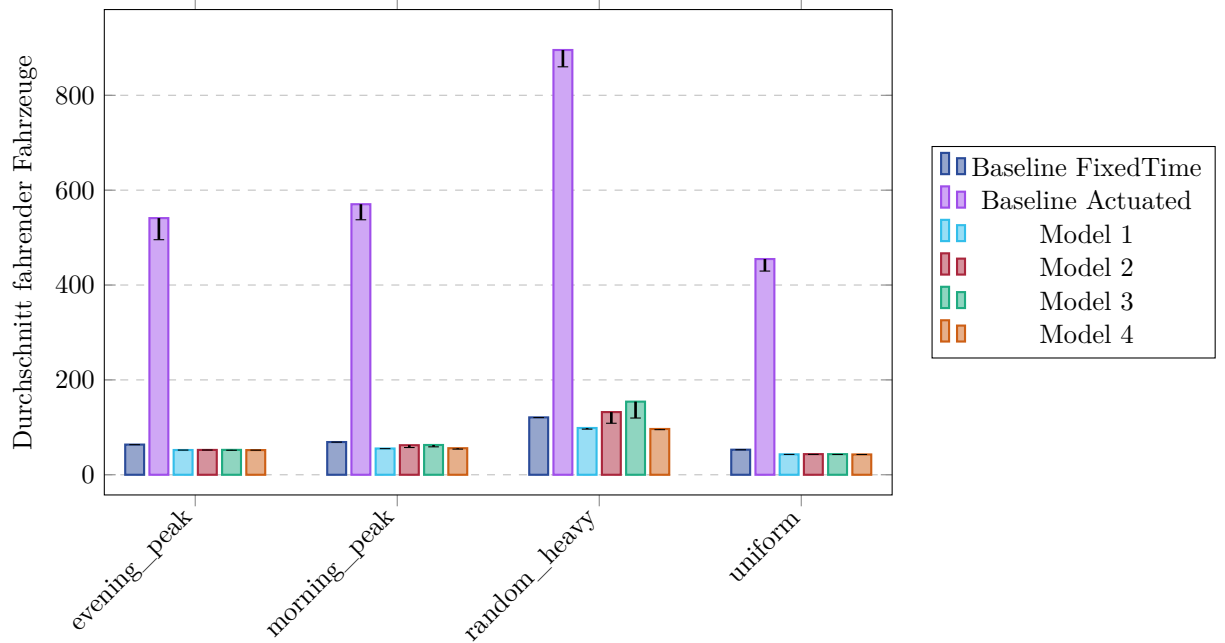


Abbildung 42: Durchschnitt fahrender Fahrzeuge

Die Analyse der durchschnittlich fahrenden Fahrzeuge bestätigt das bereits aus den anderen Metriken erkennbare Muster. Die Actuated-Baseline zeigt erneut die schlechteste Performance und liegt mit deutlich überhöhten Werten weit außerhalb des erwartbaren Bereichs. Die Fixed-Time-Baseline dient als stabiler Referenzpunkt, wird jedoch in den meisten Szenarien klar von den RL-Modellen unterboten.

Die Emissions-basierten Modelle reduzieren in `evening_peak`, `uniform` und teilweise auch `morning_peak` die Anzahl gleichzeitig fahrender Fahrzeuge um etwa 20-50 Prozent im Vergleich zu Fixed-Time. Dies deutet auf eine effizientere Steuerung hin, bei der sich weniger Fahrzeuge gleichzeitig im Netz befinden und der Verkehrsfluss stabiler wird. Auffällig ist jedoch, dass Modell 2 und Modell 3 in `random_heavy` sowie leicht auch in `morning_peak` schwächere Werte aufweisen, was die zuvor beobachteten Probleme unter hoher Last widerspiegelt.

Insgesamt verdeutlicht die Metrik, dass die Emissions-basierten Modelle in normalen Szenarien eine deutliche Entlastung des Netzes erreichen, während unter Hochlastbedingungen insbesondere Modell 2 und 3 ihre Vorteile nicht aufrechterhalten können.

5.4.6 Durchschnittsgeschwindigkeiten

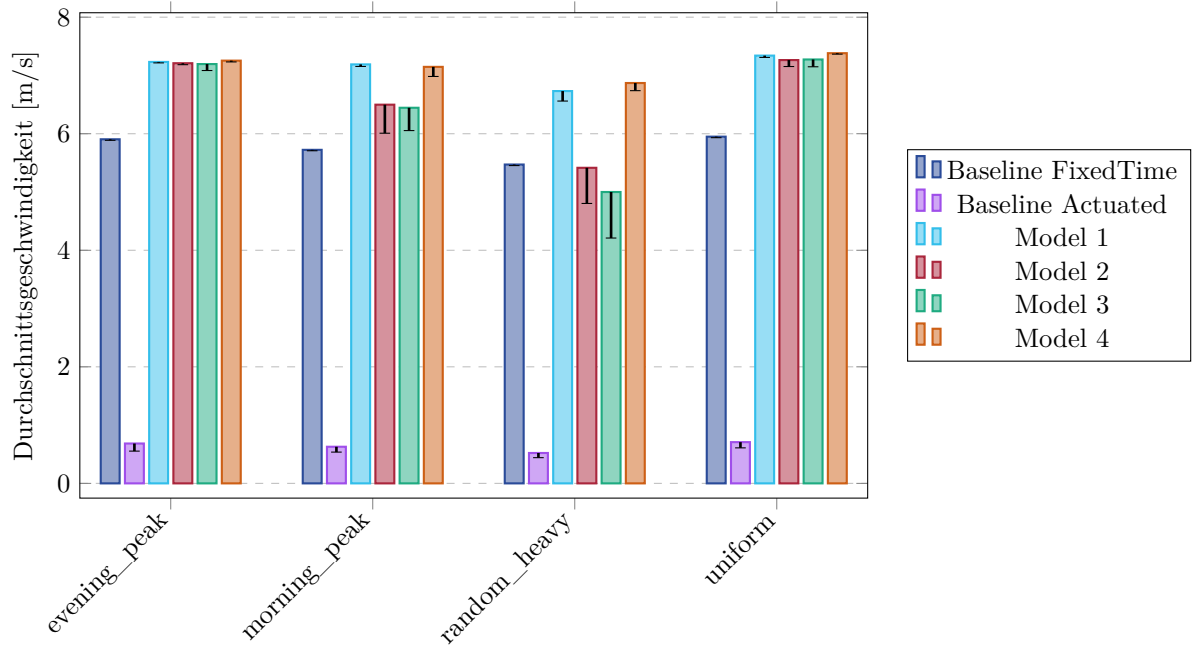


Abbildung 43: Durchschnittsgeschwindigkeiten

Die Analyse der durchschnittlichen Geschwindigkeit zeigt ein konsistentes Bild zur Metrik der durchschnittlich fahrenden Fahrzeuge. Die Actuated-Baseline erreicht durchweg die schlechtesten Ergebnisse, während die Fixed-Time-Baseline mit einer maximalen Durchschnittsgeschwindigkeit von rund 6 m/s einen stabilen Referenzwert darstellt.

Die Emissions-basierten RL-Modelle übertreffen diese Werte in allen Szenarien deutlich. Im Mittel erreichen sie etwa 7 m/s, wobei im Szenario uniform sogar 7,3 m/s erzielt werden. Dies unterstreicht die Fähigkeit der Modelle, den Verkehrsfluss effizienter zu gestalten und die Reisegeschwindigkeit im Netz spürbar zu erhöhen.

Wie bereits bei den vorherigen Metriken fallen jedoch Modell 2 und Modell 3 in den Szenarien random_heavy und teilweise morning_peak durch schwächere Ergebnisse auf. Trotz dieser Einbußen liegen ihre Werte jedoch größtenteils immer noch oberhalb der Fixed-Time-Baseline.

Insgesamt bestätigen die Ergebnisse, dass die Emissions-basierten Modelle den Verkehrsfluss in normalen Szenarien deutlich verbessern können, auch wenn in Hochlastsituationen erneut Leistungseinbußen sichtbar werden.

5.4.7 Anzahl teleportierender Fahrzeuge

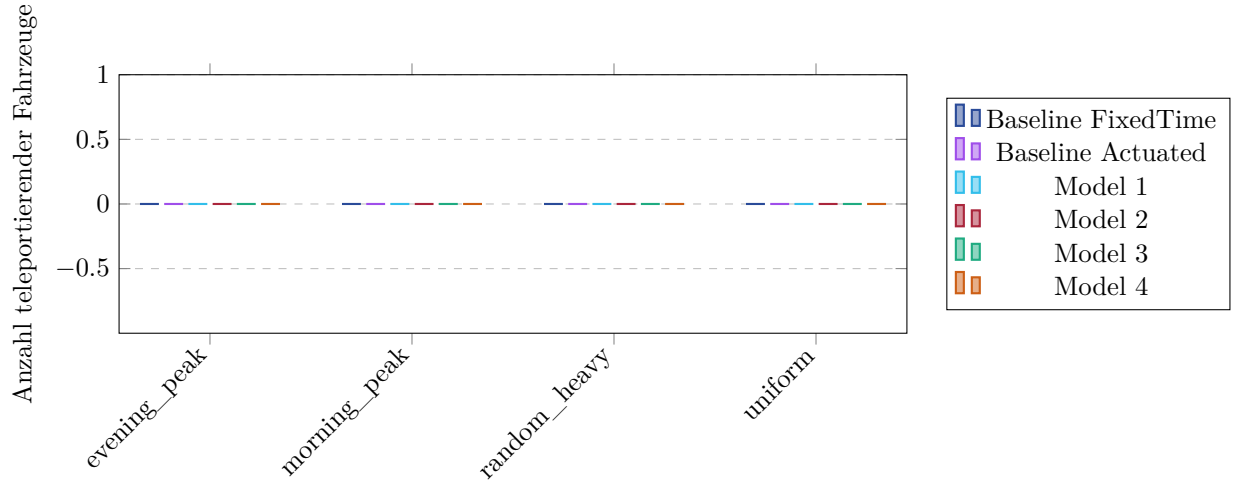


Abbildung 44: Anzahl teleportierender Fahrzeuge

In den Evaluationsläufen der Emissions-basierten Modelle traten keinerlei Teleportationen auf. Damit zeigt sich, dass die Steuerung auch unter Belastungsszenarien stabil arbeitet und keine kritischen Netzüberlastungen entstehen, die ein Eingreifen des Simulators erforderlich machen würden. Dieses Ergebnis gilt gleichermaßen für alle Modelle und hebt sie deutlich von der Actuated-Baseline ab, die in anderen Metriken wiederholt Schwächen aufweist.

5.4.8 Anzahl zurückgehaltener Fahrzeuge

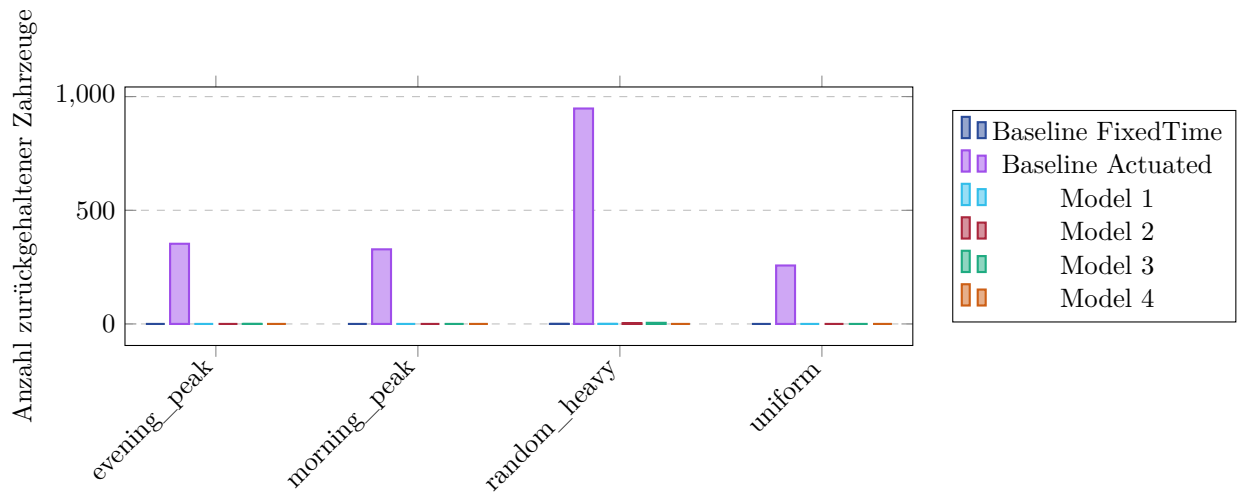


Abbildung 45: Anzahl zurückgehaltener Fahrzeuge

Bei der Metrik der zurückgehaltenen Fahrzeuge bestätigen die Emissions-basierten Modelle ihre Robustheit: In keinem der Szenarien kam es zu nennenswerten Rück-

stauwirkungen. Alle Modelle erreichen hier konsistent einen Wert von null. Einzige Ausnahme ist die Actuated-Baseline, die regelmäßig Fahrzeuge zurückhält und damit ihre unzureichende Leistungsfähigkeit im Umgang mit hoher Netzlast erneut unterstreicht.

5.4.9 Einstufung

Die mit der Emissions-Rewardfunktion trainierten Modelle zeigen insgesamt ein sehr positives Bild. In nahezu allen betrachteten Metriken übertreffen sie die Baselines deutlich. Besonders in Bezug auf die CO₂-Emissionen erreichen alle Modelle konsistent niedrigere Werte als Fixed-Time und Actuated. Gleichzeitig werden in normalen Lastszenarien auch die mittlere Wartezeit, die Anzahl stoppender Fahrzeuge sowie die Durchschnittsgeschwindigkeit signifikant verbessert. So erzielen die Modelle durchschnittliche Geschwindigkeiten von etwa 7 m/s gegenüber 5,5-6 m/s bei Fixed-Time und reduzieren die Zahl stoppender Fahrzeuge teilweise auf nahezu null.

Die Systemstabilität bleibt ebenfalls hoch: Es treten weder Teleportationen noch nennenswerte Rückstauwirkungen auf. Zudem wird die maximale Anzahl ankommender Fahrzeuge in allen Szenarien (mit Ausnahme einzelner Schwächen in `random_heavy`) erreicht.

Auffällig ist allerdings, dass Modell 2 und Modell 3 in den Szenarien `random_heavy` sowie teilweise in `morning_peak` deutlich schwächere Ergebnisse zeigen. Dort verzeichnen sie teils drastisch erhöhte Wartezeiten und eine erhöhte Anzahl stoppender Fahrzeuge, was ihre Effizienz unter extremer Belastung einschränkt. Dennoch liegen ihre Ergebnisse in diesen Fällen meist noch im Bereich der Fixed-Time-Baseline.

Insgesamt lässt sich festhalten, dass die Emissions-basierten Modelle die gesteckten Ziele weitgehend erfüllen: Sie reduzieren die Gesamtemissionen und verbessern zugleich den Verkehrsfluss in normalen Szenarien erheblich. Lediglich in Hochlastsituationen besteht noch Optimierungspotenzial, insbesondere bei einzelnen Modellvarianten.

5.5 Robustheit und Replikationsanalyse

5.6 Zusammenfassung

6 Herausforderungen und Limitationen

6.1 Technische und methodische Hürden

6.2 Repräsentativität und Qualität der Daten

6.3 Generalisierbarkeit der Ergebnisse

7 Fazit und Ausblick

7.1 Zusammenfassung der wichtigsten Erkenntnisse

7.2 Mögliche Weiterentwicklungen

7.3 Relevanz für reale Verkehrsplanung

Dieser Anhang enthält die vollständigen Python-Skripte, die zur Validierung, Reparatur und Steuerung der SUMO-basierten Reinforcement-Learning-Umgebung eingesetzt wurden. Jedes Unterkapitel dokumentiert ein spezifisches Tool oder Modul aus dem Projekt.

A Trainings-Skripte

A.1 train.py – Trainingsskript für PPO über mehrere Seeds

Das folgende Skript enthält die vollständige Trainingslogik für das Reinforcement Learning mit `sumo-rl` unter Verwendung von `Stable-Baselines3`.

```
1  # ===== Bibliotheken und Module =====
2  # Standard-Module für Dateiverwaltung, Zeit und Regex
3  import os
4  import re
5  import time
6  import datetime
7  import random
8
9  # SUMO-Interface (TraCI) für Simulation
10 import traci
11
12 # Mathematische und numerische Berechnungen
13 import numpy as np
14
15 # PyTorch für neuronale Netze und Reproduzierbarkeit
16 import torch
17
18 # Stable-Baselines3 (RL-Algorithmen, hier PPO)
19 from stable_baselines3 import PPO
20 from stable_baselines3.common.vec_env import VecNormalize, VecMonitor
21 from stable_baselines3.common.callbacks import BaseCallback, CallbackList
22
23 # SUMO-RL-Umgebung (PettingZoo-kompatibel)
24 from sumo_rl.environment.env import parallel_env
25
26 # SuperSuit - Hilfsfunktionen, um PettingZoo-Umgebungen mit SB3 zu verwenden
27 from supersuit import (
28     pad_observations_v0,          # Padding für Beobachtungen, um feste Größe
29     ↪ zu garantieren
30     pad_action_space_v0,         # Padding für Aktionsraum
31     pettingzoo_env_to_vec_env_v1, # Konvertierung zu SB3-kompatiblen
32     ↪ Vektor-Env
33     concat_vec_envs_v1           # Mehrere Envs parallel laufen lassen
34 )
35
36 # Gymnasium für RL-Umgebungs-Schnittstellen
37 import gymnasium as gym
38 from gymnasium import Wrapper
39
40 # ===== Trainings-Setup =====
41 SEEDS = [143534, 456, 635768, 13755] # Verschiedene Zufalls-Seed-Werte für
42     ↪ reproduzierbare Runs
```

```

41 ROUTE_FILES = [
42     "flows_low.rou.xml",
43     "flows_medium.rou.xml",
44     "flows_high.rou.xml",
45 ]
46
47 # ===== Schedules für Hyperparameter-Anpassung =====
48 # (Funktionen, die während des Trainings den Wert z. B. von Lernrate oder
49 ↪ Clip-Bereich dynamisch anpassen)
50 def adaptive_entropy_schedule(start=0.01):
51     return lambda progress: max(0.001, start * (1 - progress))
52
53 def dynamic_clip_range(start=0.2, end=0.1):
54     return lambda pr: end + (start - end) * pr
55
56 def cosine_clip(start=0.2, end=0.1):
57     return lambda pr: end + (start - end) * 0.5 * (1 + np.cos(np.pi * (1 -
58 ↪ pr)))
59
60 def linear_schedule(start):
61     return lambda progress: start * (1 - progress)
62
63 def cosine_warmup_floor(start=3e-4, warmup_frac=0.05, min_lr_frac=0.1):
64     """
65     Lernrate: Erst linear hochfahren (Warmup), dann mit Cosinus auf
66     ↪ Minimalwert absenken.
67     """
68     min_lr = start * min_lr_frac
69     warmup_frac = max(0.0, min(0.5, warmup_frac))
70     def schedule(progress_remaining: float) -> float:
71         t = 1.0 - progress_remaining
72         if t < warmup_frac:
73             base = start * 0.1 + (start - start * 0.1) * (t / warmup_frac)
74         else:
75             tt = (t - warmup_frac) / max(1e-8, (1.0 - warmup_frac))
76             cos_term = 0.5 * (1 + np.cos(np.pi * tt))
77             base = min_lr + (start - min_lr) * cos_term
78         return float(base)
79     return schedule
80
81 # ===== Hilfsfunktionen und Callbacks =====
82 # (Modelle finden, Checkpoints speichern, Metriken loggen, bestes Modell
83 ↪ sichern)
84 # ===== Letzten vollständigen Run finden =====
85 def find_latest_complete_run(base_dir="runs", prefix="ppo_sumo_"):
86     """
87     Sucht im 'runs'-Ordner nach dem neuesten Trainingslauf, der
88     - eine gespeicherte VecNormalize-Instanz hat
89     - und entweder ein finales Modell oder mindestens einen Checkpoint.
90     Gibt die Pfade zu Run-Ordner, Modell und Normalisierungsdatei zurück.
91     """
92     subdirs = sorted(
93         [d for d in os.listdir(base_dir) if d.startswith(prefix)],
94         reverse=True
95     )
96     for d in subdirs:
97         dir_path = os.path.join(base_dir, d)

```

```

94     norm_path = os.path.join(dir_path, "vecnormalize.pkl")
95     if not os.path.exists(norm_path):
96         continue
97
98     # Prüfe auf finales Modell
99     final_model = os.path.join(dir_path, "model.zip")
100     if os.path.exists(final_model):
101         return dir_path, final_model, norm_path
102
103     # Falls kein finales Modell: Prüfe auf Checkpoints
104     checkpoint_models = [
105         f for f in os.listdir(dir_path)
106         if re.match(r"ppo_sumo_model_(\d+)_steps\.zip", f)
107     ]
108     if checkpoint_models:
109         checkpoint_models.sort(key=lambda x: int(re.findall(r"\d+",
110             ↪ x)[0]), reverse=True)
111         best_checkpoint = checkpoint_models[0]
112         return dir_path, os.path.join(dir_path, best_checkpoint),
113             ↪ norm_path
114
115     return None
116
117 def make_env(seed, route_files):
118     def _init():
119         env = parallel_env(
120             net_file="map.net.xml",
121             route_file=route_files[0], # Platzhalter
122             use_gui=False,
123             num_seconds=4096,
124             reward_fn="diff-waiting-time",
125             min_green=5,
126             max_depart_delay=100,
127             sumo_seed=seed,
128             add_system_info=True,
129             add_per_agent_info=False,
130         )
131         if hasattr(env, "seed"):
132             env.seed(seed)
133
134     orig_reset = env.reset
135     idx = {"i": -1} # mutierbares Zähl-Objekt im Closure
136
137     def reset_with_round_robin(**kwargs):
138         idx["i"] = (idx["i"] + 1) % len(route_files)
139         new_route = route_files[idx["i"]]
140         env.route_file = new_route
141         if hasattr(env, "sumo_seed"):
142             env.sumo_seed = seed
143         print(f"\n[DEBUG] Reset → Route: {new_route} | Seed: {seed}\n",
144             ↪ flush=True)
145         return orig_reset(**kwargs)
146
147     env.reset = reset_with_round_robin
148     return env
149
150 return _init

```

```

148
149 def shorten_key(orig_key: str) -> str:
150     return orig_key.replace("system_", "")
151
152 # ===== Callback: Zeitbasiertes Speichern =====
153 class TimeBasedCheckpointCallback(BaseCallback):
154     """
155     Speichert Modell und Normalisierungsdaten in festen Zeitintervallen
156     ↳ (Sekunden).
157     """
158     def __init__(self, save_interval_sec, save_path,
159         ↳ name_prefix="ppo_sumo_model", verbose=0):
160         super().__init__(verbose)
161         self.save_interval_sec = save_interval_sec
162         self.save_path = save_path
163         self.name_prefix = name_prefix
164         self.last_save_time = time.time()
165
166     def _on_step(self) -> bool:
167         return True # Keine Aktion bei jedem einzelnen Step
168
169     def _on_rollout_end(self) -> bool:
170         # Am Ende eines Rollouts prüfen, ob das Zeitintervall abgelaufen ist
171         current_time = time.time()
172         if current_time - self.last_save_time >= self.save_interval_sec:
173             timestep = self.num_timesteps
174             filename = f"{self.name_prefix}_{timestep}_steps"
175             self.model.save(os.path.join(self.save_path, filename + ".zip"))
176             if hasattr(self.training_env, "save"):
177                 self.training_env.save(os.path.join(self.save_path,
178                     ↳ f"{filename}_vecnormalize.pkl"))
179             print(f"[Checkpoint] Modell gespeichert bei {timestep} Schritten
180                 ↳ ({filename})")
181             self.last_save_time = current_time
182         return True
183
184 # ===== Callback: Metriken aus der Env loggen =====
185 class EpisodeMetricsLoggerCallback(BaseCallback):
186     def __init__(self, prefix="episode", verbose=0):
187         super().__init__(verbose)
188         self.prefix = prefix
189         self.verbose = verbose
190         self.sums = {}
191         self.counts = {}
192         self.last_totals = {}
193
194     def _on_step(self) -> bool:
195         dones = self.locals.get("dones")
196         infos = self.locals.get("infos")
197         if infos is None:
198             return True
199
200         for i, info in enumerate(infos):
201             if not isinstance(info, dict):
202                 continue

```

```

201         if done is not None and done[i]:
202             # --- Episode zu Ende ---
203             fin = info.get("final_info") or info.get("terminal_info")
204             if isinstance(fin, dict):
205                 for k, v in fin.items():
206                     if not isinstance(v, (int, float)) or not
207                         ↪ np.isfinite(v):
208                         continue
209                     if k.startswith("system_mean_"):
210                         self.sums[k] = self.sums.get(k, 0.0) + float(v)
211                         self.counts[k] = self.counts.get(k, 0) + 1
212                     elif k.startswith("system_total_"):
213                         self.last_totals[k] = float(v)
214             else:
215                 # --- Nur Zwischenschritt, solange Episode noch läuft ---
216                 for k, v in info.items():
217                     if not isinstance(v, (int, float)) or not np.isfinite(v):
218                         continue
219                     if k.startswith("system_mean_") or k in [
220                         "system_total_waiting_time",
221                         "system_total_stopped",
222                         "system_total_running",
223                     ]:
224                         self.sums[k] = self.sums.get(k, 0.0) + float(v)
225                         self.counts[k] = self.counts.get(k, 0) + 1
226                     elif k.startswith("system_total_"):
227                         self.last_totals[k] = float(v)
228
229             # Episode fertig → loggen
230             if done is not None and any(done):
231                 for k, total in self.sums.items():
232                     mean_val = total / max(1, self.counts.get(k, 1))
233                     short_key = shorten_key(k)
234                     self.logger.record(f"{self.prefix}/{short_key}", mean_val)
235                     if self.verbose:
236                         print(f"[EpisodeMetrics] {short_key} (mean) =
237                             ↪ {mean_val:.3f}")
238
239                 for k, v in self.last_totals.items():
240                     short_key = shorten_key(k)
241                     self.logger.record(f"{self.prefix}/{short_key}", v)
242                     if self.verbose:
243                         print(f"[EpisodeMetrics] {short_key} (total) = {v:.0f}")
244
245             # Reset für nächste Episode
246             self.sums.clear()
247             self.counts.clear()
248             self.last_totals.clear()
249
250             return True
251
252         # ===== Callback: Bestes Modell speichern =====
253         class BestModelSaverCallback(BaseCallback):
254             """
255             Speichert das Modell mit dem bisher höchsten mittleren Episodenreward.
256             """
257             def __init__(self, save_path, verbose=0):

```

```

256         super().__init__(verbose)
257         self.best_mean_reward = -float('inf')
258         self.save_path = save_path
259
260     def _on_step(self) -> bool:
261         return True
262
263     def _on_rollout_end(self):
264         ep_info_buffer = self.model.ep_info_buffer
265         if len(ep_info_buffer) > 0:
266             mean_rew = np.mean([ep_info['r'] for ep_info in ep_info_buffer])
267             if mean_rew > self.best_mean_reward:
268                 self.best_mean_reward = mean_rew
269                 model_path = os.path.join(self.save_path, "best_model.zip")
270                 self.model.save(model_path)
271                 if hasattr(self.model.env, "save"):
272                     norm_path = os.path.join(self.save_path,
273                                             ↪ "best_model_vecnormalize.pkl")
274                     self.model.env.save(norm_path)
275                 print(f"[AUTOLOG] Neuer Bestwert {mean_rew:.2f} → best_model
276                     ↪ gespeichert!", flush=True)
277
278 # ===== Haupt-Trainingsschleife =====
279 for SEED in SEEDS:
280     # Reproduzierbarkeit sicherstellen
281     np.random.seed(SEED)
282     torch.manual_seed(SEED)
283
284     # Log-Verzeichnis erstellen
285     now = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
286     log_dir = os.path.join("runs", f"ppo_sumo_{SEED}_{now}")
287     os.makedirs(log_dir, exist_ok=True)
288
289     print(f"\n[INFO] Starte Training mit Seed: {SEED}")
290
291     # SUMO-Umgebung initialisieren
292     env = make_env(SEED, ROUTE_FILES)()
293
294     # Falls die Env einen seed()-Aufruf unterstützt
295     if hasattr(env, "seed"):
296         env.seed(SEED)
297
298     # Anpassung der Beobachtungen und Aktionen an SB3
299     env = pad_observations_v0(env)
300     env = pad_action_space_v0(env)
301     env = pettingzoo_env_to_vec_env_v1(env)
302
303     # WICHTIG: trotzdem concat_vec_envs_v1 mit num_vec_envs=1
304     env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=1,
305                             ↪ base_class="stable_baselines3")
306
307     # Logging und Normalisierung
308     env = VecMonitor(env, filename=os.path.join(log_dir, "monitor.csv"))
309     env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.0)

```

```

310 # PPO-Agent erstellen
311 model = PPO(
312     policy="MlpPolicy",          # Mehrschicht-Perzeptron-Policy
313     env=env,
314     verbose=1,                  # Ausführliches Logging
315     tensorboard_log=log_dir,    # TensorBoard-Pfad
316     batch_size=256,            # Minibatch-Größe für PPO
317     n_steps=2048,              # Rollout-Länge
318     learning_rate=cosine_warmup_floor(start=3e-4, warmup_frac=0.05,
319     ↪ min_lr_frac=0.1),
320     clip_range=cosine_clip(),   # Clipping-Range dynamisch
321     ent_coef=0.01,             # Entropie-Koeffizient (Exploration)
322     gamma=0.99,                # Diskontfaktor
323     gae_lambda=0.95,           # Lambda für GAE
324     device="cpu",              # Training auf CPU
325     policy_kwargs=dict(net_arch=dict(pi=[128, 128], vf=[128, 128])), #
326     ↪ Netzarchitekturgit
327 )
328
329 # Callback-Liste: Checkpoints, Logging, Best-Model-Speicherung
330 callbacks = CallbackList([
331     TimeBasedCheckpointCallback(
332         save_interval_sec=3600, # Jede Stunde speichern
333         save_path=log_dir,
334         name_prefix="ppo_sumo_model",
335         verbose=1,
336     ),
337     EpisodeMetricsLoggerCallback(),
338     BestModelSaverCallback(save_path=log_dir),
339 ])
340
341 # Training starten
342 try:
343     time.sleep(3) # Kurze Pause für saubere Konsolenlogs
344     model.learn(
345         total_timesteps=2_000_000,
346         callback=callbacks,
347     )
348     # Nach Abschluss final speichern
349     model.save(os.path.join(log_dir, "model.zip"))
350     env.save(os.path.join(log_dir, "vecnormalize.pkl"))
351     print(f"\n[INFO] Training abgeschlossen für Seed {SEED}. Modell
352     ↪ gespeichert unter: {log_dir}")
353
354 # Falls Training manuell abgebrochen wird (Strg+C)
355 except KeyboardInterrupt:
356     print("[ABBRUCH] Manuelles Beenden erkannt. Speichere aktuellen
357     ↪ Stand...")
358     model.save(os.path.join(log_dir, "model_interrupt.zip"))
359     env.save(os.path.join(log_dir, "vecnormalize_interrupt.pkl"))
360
361 # Generelle Fehlerbehandlung
362 except Exception as e:
363     print(f"\n[FEHLER] Während des Trainings bei Seed {SEED} aufgetreten:
364     ↪ {e}")
365
366 # Cleanup: Env schließen und Normalisierungsdaten sichern

```

```

362     finally:
363         try:
364             env.save(os.path.join(log_dir, "vecnormalize.pkl"))
365         except Exception as e:
366             print(f"[WARNUNG] VecNormalize konnte nicht gespeichert werden:
367                   ↪ {e}")
368     env.close()

```

A.2 continuetrain.py – Trainingsskript zum Weitertrainieren

Startet für jede einzelne Ampelkreuzung eine Minimalumgebung und überprüft, ob diese in `sumo-rl` trainierbar ist.

```

1  import os
2  import re
3  import time
4  import datetime
5  import traci
6  import numpy as np
7  import torch
8  import json
9  from stable_baselines3 import PPO
10 from stable_baselines3.common.vec_env import VecNormalize, VecMonitor
11 from stable_baselines3.common.callbacks import BaseCallback, CallbackList
12 from sumo_rl.environment.env import parallel_env
13 from supersuit import (
14     pad_observations_v0,
15     pad_action_space_v0,
16     pettingzoo_env_to_vec_env_v1,
17     concat_vec_envs_v1
18 )
19 from gym import Wrapper
20
21 # ==== Seed setzen ====
22 SEED = 42
23 np.random.seed(SEED)
24 torch.manual_seed(SEED)
25
26 # ==== Adaptive Parameter-Schedules ====
27 def dynamic_clip_range(start=0.2):
28     return lambda progress: max(0.1, start * (1 - 0.5 * progress))
29
30 def linear_schedule(start):
31     return lambda progress: start * (1 - progress)
32
33 # ==== Finde letzten vollständigen Run ====
34 def find_latest_complete_run(base_dir="runs", prefix="ppo_sumo_"):
35     subdirs = sorted(
36         [d for d in os.listdir(base_dir) if d.startswith(prefix)],
37         reverse=True
38     )
39     for d in subdirs:
40         dir_path = os.path.join(base_dir, d)
41         norm_path = os.path.join(dir_path, "vecnormalize.pkl")
42         if not os.path.exists(norm_path):

```



```

43         continue
44
45     final_model = os.path.join(dir_path, "model.zip")
46     if os.path.exists(final_model):
47         return dir_path, final_model, norm_path
48
49     checkpoint_models = [
50         f for f in os.listdir(dir_path)
51         if re.match(r"ppo_sumo_model_(\d+)_steps\.zip", f)
52     ]
53     if checkpoint_models:
54         checkpoint_models.sort(key=lambda x: int(re.findall(r"\d+",
55             ↪ x)[0]), reverse=True)
56         best_checkpoint = checkpoint_models[0]
57         return dir_path, os.path.join(dir_path, best_checkpoint),
58             ↪ norm_path
59
60     return None
61
62 # ==== Zeitbasierter Checkpoint Callback ====
63 class TimeBasedCheckpointCallback(BaseCallback):
64     def __init__(self, save_interval_sec, save_path,
65         ↪ name_prefix="ppo_sumo_model", verbose=0):
66         super().__init__(verbose)
67         self.save_interval_sec = save_interval_sec
68         self.save_path = save_path
69         self.name_prefix = name_prefix
70         self.last_save_time = time.time()
71
72     def _on_step(self) -> bool:
73         return True
74
75     def _on_rollout_end(self) -> bool:
76         current_time = time.time()
77         if current_time - self.last_save_time >= self.save_interval_sec:
78             timestep = self.num_timesteps
79             filename = f"{self.name_prefix}_{timestep}_steps"
80             self.model.save(os.path.join(self.save_path, filename + ".zip"))
81             if hasattr(self.training_env, "save"):
82                 self.training_env.save(os.path.join(self.save_path,
83                     ↪ f"{filename}_vecnormalize.pkl"))
84             print(f"[Checkpoint] Modell gespeichert bei {timestep} Schritten
85                 ↪ ({filename})")
86             self.last_save_time = current_time
87         return True
88
89 # ==== Learning Rate Logger Callback ====
90 class LearningRateLoggerCallback(BaseCallback):
91     def __init__(self, verbose=0):
92         super().__init__(verbose)
93
94     def _on_step(self) -> bool:
95         lr = self.model.lr_schedule(self.num_timesteps /
96             ↪ self.model._total_timesteps)
97         self.logger.record("train/learning_rate", lr)
98         return True

```

```

94 # ==== Logging ====
95 now = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
96 log_dir = os.path.join("runs", f"ppo_sumo_{now}")
97 os.makedirs(log_dir, exist_ok=True)
98
99 # ==== SUMO-RL Umgebung ====
100 env = parallel_env(
101     net_file="network.net.xml",
102     route_file="flow.rou.xml",
103     use_gui=False,
104     num_seconds=4096,
105     reward_fn="diff-waiting-time",
106     min_green=5,
107     max_depart_delay=100,
108     sumo_seed=SEED,
109     add_system_info=True,
110     add_per_agent_info=False,
111 )
112
113 if hasattr(env, "seed"):
114     env.seed(SEED)
115
116 # ==== Wrapping ====
117 env = pad_observations_v0(env)
118 env = pad_action_space_v0(env)
119 env = pettingzoo_env_to_vec_env_v1(env)
120 env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=8,
121     ↪ base_class="stable_baselines3")
122 env = VecMonitor(env)
123
124 # ==== Modell laden oder neu starten ====
125 result = find_latest_complete_run()
126 if result:
127     latest_run_dir, model_path, normalize_path = result
128     print("Fortsetzung wird gestartet mit:")
129     print(f"Verzeichnis : {latest_run_dir}")
130     print(f"Modell       : {model_path}")
131     print(f"Normalize     : {normalize_path}\n")
132
133     env = VecNormalize.load(normalize_path, env)
134     env.training = True
135     env.norm_reward = True
136
137     model = PPO.load(model_path, env=env, tensorboard_log=log_dir, verbose=1,
138     ↪ device="cpu")
139     print(f"[INFO] Modell startet bei {model.num_timesteps} Timesteps.")
140 else:
141     print("[INFO] Kein vorheriges Modell gefunden. Starte frisches
142     ↪ Training.\n")
143     env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.0)
144     model = PPO(
145         policy="MlpPolicy",          # Mehrschicht-Perzeptron-Policy
146         env=env,
147         verbose=1,                   # Ausführliches Logging
148         tensorboard_log=log_dir,     # TensorBoard-Pfad
149         batch_size=256,              # Minibatch-Größe für PPO
150         n_steps=2048,               # Rollout-Länge

```

```

148         learning_rate=cosine_warmup_floor(start=3e-4, warmup_frac=0.05,
149         ↪ min_lr_frac=0.1),
150         clip_range=cosine_clip(), # Clipping-Range dynamisch
151         ent_coef=0.01,           # Entropie-Koeffizient (Exploration)
152         gamma=0.99,             # Diskontfaktor
153         gae_lambda=0.95,        # Lambda für GAE
154         device="cpu",           # Training auf CPU
155         policy_kwargs=dict(net_arch=dict(pi=[128, 128], vf=[128, 128])), #
156         ↪ Netzarchitekturgit
157     )
158
159     # ==== Automatisches Speichern bei verbessertem ep_rew_mean ====
160     class BestModelSaverCallback(BaseCallback):
161         def __init__(self, save_path, verbose=0):
162             super().__init__(verbose)
163             self.best_mean_reward = -float('inf')
164             self.save_path = save_path
165
166         def _on_step(self) -> bool:
167             # Muss vorhanden sein, selbst wenn sie nichts tut
168             return True
169
170         def _on_rollout_end(self):
171             ep_info_buffer = self.model.ep_info_buffer
172             if len(ep_info_buffer) > 0:
173                 mean_rew = np.mean([ep_info['r'] for ep_info in ep_info_buffer])
174
175                 if mean_rew > self.best_mean_reward:
176                     self.best_mean_reward = mean_rew
177
178                     model_path = os.path.join(self.save_path, "best_model.zip")
179                     self.model.save(model_path)
180
181                     if hasattr(self.model.env, "save"):
182                         norm_path = os.path.join(self.save_path,
183                         ↪ "best_model_vecnormalize.pkl")
184                         self.model.env.save(norm_path)
185
186                     print(f"[AUTOLOG] Neuer Bestwert {mean_rew:.2f} → best_model
187                     ↪ gespeichert!", flush=True)
188
189     # ==== Callbacks kombinieren ====
190     callbacks = CallbackList([
191         TimeBasedCheckpointCallback(
192             save_interval_sec=3600,
193             save_path=log_dir,
194             name_prefix="ppo_sumo_model",
195             verbose=1,
196         ),
197         LearningRateLoggerCallback(),
198         BestModelSaverCallback(save_path=log_dir),
199     ])
200
201     # ==== Training starten ====
202     try:
203         model.learn(
204             total_timesteps=1_000_000,

```

```

201         callback=callbacks,
202     )
203     model.save(os.path.join(log_dir, "model.zip"))
204     env.save(os.path.join(log_dir, "vecnormalize.pkl"))
205     print(f"\n[INFO] Training abgeschlossen. Modell gespeichert unter:
    ↳ {log_dir}")
206
207 except KeyboardInterrupt:
208     print("[ABBRUCH] Manuelles Beenden erkannt. Speichere aktuellen Stand...")
209     model.save(os.path.join(log_dir, "model_interrupt.zip"))
210     env.save(os.path.join(log_dir, "vecnormalize_interrupt.pkl"))
211
212 except Exception as e:
213     print(f"\n[FEHLER] Während des Trainings aufgetreten: {e}")
214
215 finally:
216     try:
217         env.save(os.path.join(log_dir, "vecnormalize.pkl"))
218     except Exception as e:
219         print(f"[WARNUNG] VecNormalize konnte nicht gespeichert werden: {e}")
220     env.close()

```

B Belohnungsfunktionen

B.1 diff-waiting-time

Diese Belohnungsfunktion misst die Differenz der kumulierten Wartezeit zwischen zwei Zeitschritten. Sie belohnt eine Abnahme der Gesamtwartezeit.

```

1 def diff_waiting_time_reward(traffic_signal):
2     ts_wait = sum(traffic_signal.get_accumulated_waiting_time_per_lane()) /
    ↳ 100.0
3     reward = traffic_signal.last_ts_waiting_time - ts_wait
4     traffic_signal.last_ts_waiting_time = ts_wait
5     return reward

```

B.2 queue

Hier wird die Anzahl an gestoppten Fahrzeugen direkt als negativer Reward verwendet. Weniger Stau → höherer Reward.

```

1 def queue_reward(traffic_signal):
2     return -traffic_signal.get_total_queued()
3
4 def get_total_queued(traffic_signal) -> int:
5     """Returns the total number of vehicles halting in the intersection."""
6     return sum(traffic_signal.sumo.lane.getLastStepHaltingNumber(lane) for
    ↳ lane in traffic_signal.lanes)

```

B.3 realworld

Diese Funktion kombiniert Geschwindigkeit, Warteschlangenlänge und mittlere Wartezeit in einem additiven Reward.

```

1 def realworld_reward(traffic_signal):
2     # Speed (0-7.5 m/s -> 0-1)
3     avg_speed = traffic_signal.get_average_speed()
4     speed_term = min(max(avg_speed, 0.0), 7.5) / 7.5
5
6     # Queue (0-20 Fzg -> 0-1)
7     total_queue = traffic_signal.get_total_queued()
8     queue_term = min(max(total_queue, 0), 20) / 20.0
9
10    # Mean waiting time (0-10 s -> 0-1)
11    waits_per_lane = traffic_signal.get_accumulated_waiting_time_per_lane()
12    mean_wait = sum(waits_per_lane) / len(waits_per_lane) if waits_per_lane
13    ↪ else 0.0
14    wait_term = min(max(mean_wait, 0.0), 20.0) / 10.0
15
16    reward = speed_term - queue_term - wait_term
17    return reward

```

B.4 emissions

Diese Variante erweitert den Reward zusätzlich um einen Term für die CO₂-Emissionen, sodass sowohl Verkehrsfluss als auch Nachhaltigkeit berücksichtigt werden.

```

1 def emissions_reward(traffic_signal):
2     env = getattr(traffic_signal, "env", None)
3     if env is None or getattr(env, "sumo", None) is None:
4         return 0.0
5     if float(env.sim_step) >= float(env.sim_max_time):
6         return 0.0
7
8     sumo = env.sumo
9
10    # Speed (0-7.5 m/s -> 0..1)
11    avg_speed = traffic_signal.get_average_speed()
12    speed_term = min(max(avg_speed, 0.0), 7.5) / 7.5
13
14    # Queue (0-20 -> 0..1)
15    total_queue = traffic_signal.get_total_queued()
16    queue_term = min(max(total_queue, 0), 20) / 20.0
17
18    # Wait (0-10 s -> 0..1)
19    waits = traffic_signal.get_accumulated_waiting_time_per_lane()
20    mean_wait = (sum(waits) / len(waits)) if waits else 0.0
21    wait_term = min(max(mean_wait, 0.0), 10.0) / 10.0
22
23    # Emissionen
24    try:
25        lanes = getattr(traffic_signal, "lanes", [])
26        total_co2 = sum(sumo.lane.getCO2Emission(lane) for lane in lanes)
27        n_veh = sum(sumo.lane.getLastStepVehicleNumber(lane) for lane in
28        ↪ lanes)
29        BASELINE = 300.0 * max(1, len(lanes))
30        CAP = 2000.0 * max(1, len(lanes))
31        co2_term = max(0.0, min(total_co2 - BASELINE, CAP - BASELINE)) / (CAP
32        ↪ - BASELINE)

```

```

31     except Exception:
32         co2_term = 0.0
33
34     reward = speed_term - queue_term - wait_term - co2_term
35     return reward

```

C Evaluierungs-Skripte

C.1 evaluate.py – Evaluationsskript für PPO-Modelle und Baselines

Dieses Skript führt die Evaluation aller trainierten PPO-Modelle in SUMO durch und vergleicht sie mit den Baselines *Fixed-Time* und *Actuated*. Es rollt mehrere Episoden über verschiedene Seeds aus, extrahiert Metriken (z. B. Wartezeit, Geschwindigkeit, Emissionen) und speichert die Ergebnisse als `eval_results.json`.

```

1  import os, json, numpy as np
2  import glob
3  from stable_baselines3 import PPO
4  from stable_baselines3.common.vec_env import VecNormalize, VecMonitor
5  from stable_baselines3.common.logger import configure
6  from sumo_rl.environment.env import parallel_env
7  from supersuit import pad_observations_v0, pad_action_space_v0
8  from supersuit import pettingzoo_env_to_vec_env_v1, concat_vec_envs_v1
9
10 # ----- Config -----
11 RUNS = sorted(glob.glob(os.path.join("runs", "ppo_sumo_*")))
12 MODEL_NAME = "best_model.zip"
13 N_EPISODES = 10
14 EP_LENGTH_S = 4096
15 EP_SEEDS = [12345, 67890, 13579, 24680, 11223, 44556, 77889, 99100, 31415,
16             ↪ 27182]
17 SCENARIOS = [
18     {"name": "morning_peak", "route_file": "flows_morning.rou.xml"},
19     {"name": "evening_peak", "route_file": "flows_evening.rou.xml"},
20     {"name": "uniform", "route_file": "flows_uniform.rou.xml"},
21     {"name": "random_heavy", "route_file": "flows_random_heavy.rou.xml"},
22 ]
23
24 # ----- Env Factory -----
25 def make_env(route_file, sumo_seed):
26     print(f"[DEBUG] Creating SUMO env with route={route_file},
27           ↪ seed={sumo_seed}")
28     env = parallel_env(
29         net_file="map.net.xml",
30         route_file=route_file,
31         use_gui=False,
32         num_seconds=EP_LENGTH_S,
33         reward_fn=dummy_reward,
34         min_green=5,
35         max_depart_delay=100,
36         sumo_seed=sumo_seed,
37         add_system_info=True,
38         add_per_agent_info=False,

```

```

37     )
38     env = pad_observations_v0(env)
39     env = pad_action_space_v0(env)
40     env = pettingzoo_env_to_vec_env_v1(env)
41     env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=1,
42     ↪ base_class="stable_baselines3")
43     env = VecMonitor(env)
44     return env
45
46 # ----- Model Loader -----
47 def load_model_and_norm(env, run_dir):
48     vecnorm_path = os.path.join(run_dir, "vecnormalize.pkl")
49     model_path = os.path.join(run_dir, MODEL_NAME)
50
51     #print(f"[DEBUG] Loading VecNormalize from {vecnorm_path}")
52     env = VecNormalize.load(vecnorm_path, env)
53     env.training = False
54     env.norm_reward = False
55
56     #print(f"[DEBUG] Loading PPO model from {model_path}")
57     model = PPO.load(model_path, env=env, device="cpu")
58     return model, env
59
60 # ----- Rollout -----
61 def rollout(model, env):
62     obs = env.reset()
63     dones = [False]
64
65     sums = {}
66     counts = {}
67     last_totals = {}
68
69     while True:
70         action, _ = model.predict(obs, deterministic=True)
71         obs, rewards, dones, infos = env.step(action)
72
73         info = infos[0] if isinstance(infos, list) else infos
74         if not isinstance(info, dict):
75             info = {}
76
77         # Wenn Episode zu Ende ist:
78         if dones[0]:
79             # Falls vorhanden, final_info/terminal_info verwenden
80             fin = info.get("final_info") or info.get("terminal_info")
81             if isinstance(fin, dict):
82                 # Mittelwerte vom finalen Step noch einrechnen
83                 for k, v in fin.items():
84                     if k.startswith("system_mean_") and isinstance(v, (int,
85                     ↪ float)) and np.isfinite(v):
86                         sums[k] = sums.get(k, 0.0) + float(v)
87                         counts[k] = counts.get(k, 0) + 1
88
89                 # Totals aus final_info (echte Endstände)
90                 for k, v in fin.items():
91                     if k.startswith("system_total_") and isinstance(v, (int,
92                     ↪ float)) and np.isfinite(v):
93                         last_totals[k] = float(v)
94
95     break

```

```

91
92     # Normaler Zwischenschritt: Mittelwerte sammeln + Totals „letzten
93     ↳ gültigen" merken
94     for k, v in info.items():
95         if not isinstance(v, (int, float)) or not np.isfinite(v):
96             continue
97         if k.startswith("system_mean_") or k in
98             ↳ ["system_total_waiting_time", "system_total_stopped",
99             ↳ "system_total_running"]:
100             # momentane Werte mitteln
101             sums[k] = sums.get(k, 0.0) + float(v)
102             counts[k] = counts.get(k, 0) + 1
103         elif k.startswith("system_total_"):
104             # Totals: nur letzten Wert merken
105             last_totals[k] = float(v)
106
107     mean_metrics = {k: (sums[k] / max(1, counts.get(k, 0))) for k in sums}
108     mean_metrics.update(last_totals)
109     return mean_metrics
110
111
112 def shorten_key(orig_key: str) -> str:
113     return orig_key.replace("system_", "")
114
115 # ----- Env Factory für Baselines -----
116 def make_env_baseline(route_file, sumo_seed, fixed_time=True):
117     """
118     Erstellt eine SUMO-Umgebung, die den internen Controller verwendet.
119     fixed_time=True -> Fester Phasenplan aus net.xml
120     fixed_time=False -> SUMO Actuated Control (falls in net.xml konfiguriert)
121     """
122     env = parallel_env(
123         net_file="map.net.xml",
124         route_file=route_file,
125         use_gui=False,
126         num_seconds=EP_LENGTH_S,
127         reward_fn=dummy_reward,
128         fixed_ts=fixed_time,
129         sumo_seed=sumo_seed,
130         add_system_info=True,
131         add_per_agent_info=False,
132     )
133     env = pad_observations_v0(env)
134     env = pad_action_space_v0(env)
135     env = pettingzoo_env_to_vec_env_v1(env)
136     env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=1,
137         ↳ base_class="stable_baselines3")
138     env = VecMonitor(env)
139     return env
140
141 def dummy_reward(_ts):
142     return 0.0
143
144 def rollout_baseline(env):
145     obs = env.reset()
146     dones = [False]

```



```

144     # Mittelwerte über die Episode
145     sums = {}
146     counts = {}
147     # Letzte gültige Totals (vor Reset)
148     last_totals = {}
149
150     # gültige Dummy-Aktion aus dem Action Space
151     dummy_action = np.array([env.action_space.sample() for _ in
152                               ↪ range(env.num_envs)])
153
154     while True:
155         obs, rewards, dones, infos = env.step(dummy_action)
156
157         info = infos[0] if isinstance(infos, list) else infos
158         if not isinstance(info, dict):
159             info = {}
160
161         # Wenn Episode zu Ende ist:
162         if dones[0]:
163             # Falls vorhanden, final_info/terminal_info verwenden
164             fin = info.get("final_info") or info.get("terminal_info")
165             if isinstance(fin, dict):
166                 # Mittelwerte vom finalen Step noch einrechnen
167                 for k, v in fin.items():
168                     if k.startswith("system_mean_") and isinstance(v, (int,
169                               ↪ float)) and np.isfinite(v):
170                         sums[k] = sums.get(k, 0.0) + float(v)
171                         counts[k] = counts.get(k, 0) + 1
172                 # Totals aus final_info (echte Endstände)
173                 for k, v in fin.items():
174                     if k.startswith("system_total_") and isinstance(v, (int,
175                               ↪ float)) and np.isfinite(v):
176                         last_totals[k] = float(v)
177
178             break
179
180         # Normaler Zwischenschritt: Mittelwerte sammeln + Totals „letzten
181         ↪ gültigen“ merken
182         for k, v in info.items():
183             if not isinstance(v, (int, float)) or not np.isfinite(v):
184                 continue
185             if k.startswith("system_mean_") or k in
186             ↪ ["system_total_waiting_time", "system_total_stopped",
187             ↪ "system_total_running"]:
188                 # momentane Werte mitteln
189                 sums[k] = sums.get(k, 0.0) + float(v)
190                 counts[k] = counts.get(k, 0) + 1
191             elif k.startswith("system_total_"):
192                 # Totals: nur letzten Wert merken
193                 last_totals[k] = float(v)
194
195     # Mittelwerte berechnen
196     mean_metrics = {k: (sums[k] / max(1, counts.get(k, 0))) for k in sums}
197     # Letzte gültige Totals übernehmen
198     mean_metrics.update(last_totals)
199
200     return mean_metrics

```

```

195
196 def to_serializable(obj):
197     if isinstance(obj, (np.integer,)):
198         return int(obj)
199     elif isinstance(obj, (np.floating,)):
200         return float(obj)
201     elif isinstance(obj, (np.ndarray,)):
202         return obj.tolist()
203     return str(obj)
204
205 # ----- Evaluation Loop -----
206 # ----- Evaluation Loop -----
207 def evaluate():
208     results = []
209     log_dir_root = os.path.join("evaluation", "logs")
210
211     # Zählung: 2 Baselines + len(RUNS) RL pro (scenario × episode)
212     total_episodes = (2 + len(RUNS)) * len(SCENARIOS) * N_EPISODES
213     ep_counter = 0
214
215     for sc in SCENARIOS:
216         scen_log_dir = os.path.join(log_dir_root, f"eval_{sc['name']}")
217         os.makedirs(scen_log_dir, exist_ok=True)
218         logger = configure(scen_log_dir, ["tensorboard", "stdout"])
219
220         print(f"[INFO] Evaluating scenario={sc['name']}")
221
222         for ep in range(N_EPISODES):
223             ep_seed = EP_SEEDS[ep]
224
225             # --- 1) Fixed-Time ---
226             env = make_env_baseline(sc["route_file"], sumo_seed=ep_seed,
227                                     ↪ fixed_time=True)
228             ep_counter += 1
229             print(f"[PROGRESS] FixedTime | {sc['name']} | Ep
230                 ↪ {ep+1}/{N_EPISODES} "
231                   f"({ep_counter}/{total_episodes})")
232             m = rollout_baseline(env)
233             m.update({
234                 "scenario": sc["name"],
235                 "episode": ep,
236                 "method": "Baseline_FixedTime"
237             })
238             results.append(m)
239
240             # --- 2) Actuated ---
241             env = make_env_baseline(sc["route_file"], sumo_seed=ep_seed,
242                                     ↪ fixed_time=False)
243             ep_counter += 1
244             print(f"[PROGRESS] Actuated | {sc['name']} | Ep
245                 ↪ {ep+1}/{N_EPISODES} "
246                   f"({ep_counter}/{total_episodes})")
247             m = rollout_baseline(env)
248             m.update({
249                 "scenario": sc["name"],
250                 "episode": ep,
251                 "method": "Baseline_Actuated"

```

```

248         })
249         results.append(m)
250
251     # --- 3) RL-Modelle ---
252     for run_dir in RUNS:
253         env_raw = make_env(sc["route_file"], sumo_seed=ep_seed)
254         model, env = load_model_and_norm(env_raw, run_dir)
255         ep_counter += 1
256         model_name = os.path.basename(run_dir)
257         print(f"[PROGRESS] RL | {sc['name']} | {model_name} "
258               f"| Ep {ep+1}/{N_EPISODES}"
259               ↪ ({ep_counter}/{total_episodes})")
260         m = rollout(model, env)
261
262         # Seed extrahieren (3. Teil vom Namen)
263         parts = model_name.split("_")
264         model_seed = parts[2] if len(parts) > 2 else "unknown"
265
266         m.update({
267             "scenario": sc["name"],
268             "episode": ep,
269             "method": f"{model_name}_{model_seed}"
270         })
271         results.append(m)
272
273     # --- Logging dieser Episode (Baselines + alle RL) ---
274     for entry in results[-(2 + len(RUNS))]:
275         for k, v in entry.items():
276             if isinstance(v, (int, float)) and k not in ["episode",
277                 ↪ "ep_seed"]:
278                 short_key = shorten_key(k)
279                 logger.record(f"{entry['method']}/{short_key}", v)
280     logger.dump(step=ep)
281
282     results_path = os.path.join("evaluation", "eval_results.json")
283     os.makedirs(os.path.dirname(results_path), exist_ok=True)
284     with open(results_path, "w") as f:
285         json.dump(results, f, indent=2, default=to_serializable)
286
287     print(f"[INFO] Evaluation abgeschlossen. Ergebnisse: {results_path}")
288
289 if __name__ == "__main__":
290     evaluate()

```

D Postprocessing der Evaluationsergebnisse

D.1 json2csv.py – Konvertierung und Aggregation von Evaluationsergebnissen

Dieses Skript verarbeitet die von `evaluate.py` erzeugte JSON-Datei `eval_results.json`. Es wandelt die Rohdaten zunächst in ein CSV-Format um, berechnet anschließend Mittelwerte und Standardabweichungen pro *Scenario* \times *Methode* und erzeugt sowohl eine aggregierte Gesamttabelle als auch separate CSV-Dateien pro Methode.

```

1  import json
2  import pandas as pd
3  import os
4
5  # Pfade
6  json_path = "evaluation/eval_results.json"
7  raw_csv_path = "evaluation/eval_results_raw.csv"
8  agg_csv_path = "evaluation/eval_results_agg.csv"
9
10 # -----
11 # Schritt 1: JSON -> Raw CSV
12 # -----
13 print(f"Lesen JSON-Datei: {json_path}")
14 with open(json_path, "r") as f:
15     data = json.load(f)
16
17 df = pd.DataFrame(data)
18 df.to_csv(raw_csv_path, index=False)
19 print(f"Raw CSV geschrieben: {raw_csv_path}")
20
21 # -----
22 # Schritt 2: Aggregation
23 # -----
24 # numerische Spalten automatisch finden (alles außer scenario, method,
25   ↳ episode)
26 numeric_cols = df.select_dtypes(include="number").columns.tolist()
27 numeric_cols = [c for c in numeric_cols if c not in ["episode"]] # episode
28   ↳ nicht mitteln
29
30 # Aggregationsdict
31 agg_dict = {}
32 for col in numeric_cols:
33     agg_dict[f"{col}_mean"] = (col, "mean")
34     agg_dict[f"{col}_std"] = (col, "std")
35
36 # Gruppieren nach Szenario + Methode
37 agg = df.groupby(["scenario", "method"]).agg(**agg_dict).reset_index()
38
39 # Gesamte Aggregation speichern
40 agg.to_csv(agg_csv_path, index=False)
41 print(f"Aggregierte Datei geschrieben: {agg_csv_path}")
42 print("Zeilen:", len(agg))
43
44 # -----
45 # Schritt 3: Pro-Methode CSVs
46 # -----
47 for method, df_method in agg.groupby("method"):
48     safe_name = method.replace(" ", "_").replace("/", "_")
49     out_path = f"evaluation/{safe_name}.csv"
50     df_method.to_csv(out_path, index=False)
51     print(f"Datei für Methode '{method}' geschrieben: {out_path}")

```

E Netzwerk-Skripte

E.1 check_tls_consistency.py – Prüfung inkonsistenter Phasenlängen

Dieses Tool analysiert alle TLS im SUMO-Netz und prüft, ob die Länge des `state`-Strings mit der Anzahl der kontrollierten Verbindungen übereinstimmt.

```
1 import xml.etree.ElementTree as ET
2
3 # === Konfiguration ===
4 net_file = "karlsruhe.net.xml"
5
6 # === Einlesen ===
7 tree = ET.parse(net_file)
8 root = tree.getroot()
9
10 # === Alle controlledLinks zählen ===
11 tls_controlled_links = {}
12 for connection in root.findall("connection"):
13     if "tl" in connection.attrib and "linkIndex" in connection.attrib:
14         tls_id = connection.attrib["tl"]
15         tls_controlled_links.setdefault(tls_id,
16             ↪ set()).add(int(connection.attrib["linkIndex"]))
17
18 # === Alle Phasen prüfen ===
19 def check_tls_lengths():
20     print("Überprüfe alle TLS auf inkonsistente Phasenlängen...\n")
21     any_issues = False
22     for logic in root.findall("tlLogic"):
23         tls_id = logic.attrib["id"]
24         expected_len = len(tls_controlled_links.get(tls_id, []))
25
26         if expected_len == 0:
27             print(f" TLS '{tls_id}' hat keine controlledLinks (wird evtl.
28                 ↪ nicht gesteuert)")
29             continue
30
31         for i, phase in enumerate(logic.findall("phase")):
32             actual_len = len(phase.attrib["state"])
33             if actual_len != expected_len:
34                 print(f" Phase {i} von TLS '{tls_id}' hat Länge {actual_len},
35                     ↪ erwartet: {expected_len}")
36                 print(f"      → state=\"{phase.attrib['state']}\"")
37                 any_issues = True
38
39     if not any_issues:
40         print(" Alle TLS-Phasen stimmen mit ihren controlledLinks überein!")
41
42 check_tls_lengths()
```

E.2 check_tls_requests.py – Prüfung ungültiger <request>-Indizes

Prüft, ob alle request-Indizes innerhalb der zulässigen Grenzen liegen, um Laufzeitfehler in sumo-rl zu vermeiden.

```
1 import xml.etree.ElementTree as ET
2
3 net_file = "karlsruhe.net.xml"
4 tree = ET.parse(net_file)
5 root = tree.getroot()
6
7 # Zähle für jedes TLS wie viele signal indices es gibt (controlled links)
8 tls_signal_indices = {}
9 for conn in root.findall("connection"):
10     if "tl" in conn.attrib and "linkIndex" in conn.attrib:
11         tls_id = conn.attrib["tl"]
12         tls_signal_indices.setdefault(tls_id,
13             ↪ set()).add(int(conn.attrib["linkIndex"]))
14
15 # Vergleiche mit den request-Elementen
16 print("Überprüfe request-Indizes gegen Signalindizes...\n")
17 any_issues = False
18 for junction in root.findall("junction"):
19     tls_id = junction.attrib.get("id")
20     requests = junction.findall("request")
21     if tls_id in tls_signal_indices:
22         expected_max = len(tls_signal_indices[tls_id])
23         for req in requests:
24             index = int(req.attrib["index"])
25             if index >= expected_max:
26                 print(f"Junction '{tls_id}': request index {index} > max
27                     ↪ signal index {expected_max - 1}")
28                 any_issues = True
29
30 if not any_issues:
31     print("Alle request-Indizes passen zu den TLS-Signalindizes!")
```

E.3 fix_requests.py – Automatische Korrektur von Requests und Phasen

Dieses Skript bereinigt überzählige <request>-Einträge und passt state-Strings in den Phasenlängen an.

```
1 import xml.etree.ElementTree as ET
2
3 net_file = "karlsruhe.net.xml"
4 output_file = "karlsruhe_fixed_tls.net.xml"
5
6 tree = ET.parse(net_file)
7 root = tree.getroot()
8
9 # Finde maximal verwendete Signal-Indices pro TLS
10 tls_max_index = {}
```

```

11 for conn in root.findall("connection"):
12     tl = conn.get("tl")
13     idx = conn.get("linkIndex")
14     if tl and idx:
15         idx = int(idx)
16         tls_max_index[tl] = max(tls_max_index.get(tl, -1), idx)
17
18 # Bereinigung
19 total_removed_requests = 0
20 total_adjusted_phases = 0
21 changed_tls = []
22
23 for junction in root.findall("junction"):
24     tls_id = junction.get("id")
25     if tls_id not in tls_max_index:
26         continue
27
28     max_idx = tls_max_index[tls_id]
29     requests = list(junction.findall("request"))
30     removed = 0
31
32     for req in requests:
33         req_idx = int(req.get("index"))
34         if req_idx > max_idx:
35             junction.remove(req)
36             removed += 1
37
38     if removed > 0:
39         print(f"TLS '{tls_id}': {removed} ungültige <request>-Einträge
40             ↳ entfernt.")
41         total_removed_requests += removed
42         changed_tls.append(tls_id)
43
44 # Kürze zugehörige Phasen
45 for tl in root.findall("tlLogic"):
46     if tl.get("id") == tls_id:
47         adjusted = 0
48         for phase in tl.findall("phase"):
49             state = phase.get("state")
50             if len(state) > max_idx + 1:
51                 old_len = len(state)
52                 phase.set("state", state[:max_idx + 1])
53                 adjusted += 1
54         if adjusted > 0:
55             print(f" TLS '{tls_id}': {adjusted} <phase>-Strings auf Länge
56                 ↳ {max_idx + 1} gekürzt.")
57             total_adjusted_phases += adjusted
58             if tls_id not in changed_tls:
59                 changed_tls.append(tls_id)
60
61 # Speichern
62 tree.write(output_file, encoding="utf-8")
63 print("\n Reparatur abgeschlossen.")
64 print(f" Gesamt entfernte <request>-Einträge: {total_removed_requests}")
65 print(f" Gesamt angepasste <phase>-Einträge: {total_adjusted_phases}")
66 print(f" Betroffene TLS-IDs: {len(changed_tls)} Stück")
67 for tls in changed_tls:

```

```

66     print(f" - {tls}")
67     print(f"\n Bereinigte Datei gespeichert unter: {output_file}")

```

E.4 repair_net.py – manuelle TLS-Reparatur auf Basis eines Referenz-Dictionaries

Repariert TLS-Definitionen durch Abgleich mit einer vordefinierten Mapping-Tabelle von korrekten Phasenlängen.

```

1  from xml.etree import ElementTree as ET
2
3  # Manuell gepflegte Dictionary mit {TLS-ID: Anzahl controlledLinks}
4  controlled_links = {
5      "1720933516": 6,
6      "3538953167": 2,
7      "3664415977": 10,
8      "cluster_14795187_1720919996_2670370290_2670370291": 11,
9      "cluster_14795804_55474925_6655074904_765746891_#1more": 49,
10     "cluster_15431428_1719671850_1720917935": 20,
11     "cluster_1590912233_3664415976_5083348337_5083348350": 11,
12     "cluster_1692973685_1692973722_1718084055_1718084058_#11more": 36,
13     "cluster_1729190097_3687504105": 8,
14     "cluster_1744031943_5131521735": 10,
15
16     ↪ "joinedS_1623835169_cluster_1137679587_1626739216_1728272870_1728272909_#17more":
17     ↪ 33,
18
19     ↪ "joinedS_309108716_cluster_11001804363_1125509937_12515596172_1784859792_#5more":
20     ↪ 14,
21     "joinedS_5092985445_cluster_1590912226_2911376263": 10,
22     # ggf. mehr hinzufügen
23 }
24
25 tree = ET.parse("karlsruhe.net.xml")
26 root = tree.getroot()
27 changed = False
28
29 for logic in root.findall("tlLogic"):
30     tl_id = logic.attrib["id"]
31     if tl_id not in controlled_links:
32         continue
33
34     correct_len = controlled_links[tl_id]
35     for phase in logic.findall("phase"):
36         state = phase.attrib["state"]
37         if len(state) != correct_len:
38             new_state = state[:correct_len].ljust(correct_len, 'r')
39             print(f" Fixing {tl_id}: {len(state)} → {correct_len}")
40             phase.attrib["state"] = new_state
41             changed = True
42
43 if changed:
44     tree.write("karlsruhe_fixed.net.xml")
45     print(" Bereinigte Datei gespeichert: karlsruhe_fixed.net.xml")
46 else:

```



```
43     print(" Alle Phasen bereits korrekt.")
44
```

E.5 statecheck.py – Prüfung auf Ziel-Phasenlänge

Hilft bei der Kontrolle einheitlicher Phasenlängen über das gesamte Netz hinweg (z. B. Zielwert = 57).

```
1  from xml.etree import ElementTree as ET
2
3  tree = ET.parse("karlsruhe.net.xml")
4  root = tree.getroot()
5
6  for logic in root.findall("tlLogic"):
7      tl_id = logic.attrib["id"]
8      for i, phase in enumerate(logic.findall("phase")):
9          state = phase.attrib["state"]
10         if len(state) != 57:
11             print(f" Phase {i} of TLS '{tl_id}' has length {len(state)}")
```

E.6 find_valid_tls.py – Validierung lauffähiger TLS für SUMO-RL

Startet für jede einzelne Ampelkreuzung eine Minimalumgebung und überprüft, ob diese in sumo-rl trainierbar ist.

```
1  from sumo_rl import SumoEnvironment
2  import traci
3  import os
4
5  def test_tls(tls_id):
6      try:
7          env = SumoEnvironment(
8              net_file="karlsruhe.net.xml",
9              route_file="karlsruhe.rou.xml",
10             use_gui=False,
11             single_agent=True
12         )
13         env.ts_ids = [tls_id]
14         env.reset()
15         env.close()
16         return True
17     except Exception as e:
18         print(f" TLS {tls_id} nicht gültig: {e}")
19         return False
20
21  # Alle TLS holen
22  try:
23      env = SumoEnvironment(
24          net_file="karlsruhe.net.xml",
25          route_file="karlsruhe.rou.xml",
26          use_gui=False,
27          single_agent=True
28      )
```

```

29     all_tls = env.ts_ids
30     env.close()
31 except Exception as e:
32     print(" Konnte TLS nicht auslesen:", e)
33     all_tls = []
34
35 print(f" Teste {len(all_tls)} TLS auf Gültigkeit...\n")
36 valid_tls = []
37
38 for tls_id in all_tls:
39     if test_tls(tls_id):
40         valid_tls.append(tls_id)
41
42 print("\n Gültige TLS:")
43 print(valid_tls)

```

E.7 find_relevant_edges.py – Suche alle relevanten edges

```

1 import xml.etree.ElementTree as ET
2
3 # Konfiguration: Pfad zur .net.xml-Datei und Suchbegriffe
4 NET_FILE = "network.net.xml"
5 SUCHBEGRIFFE = ["B10", "B36", "L605", "Durlacher Allee",
6     ↪ "Reinhold-Frank-Straße"]
7
8 # Ausgabe-Datei für gefundene Kanten
9 OUTPUT_FILE = "edges.txt"
10
11 def finde_relevante_kanten(net_file, suchbegriffe):
12     tree = ET.parse(net_file)
13     root = tree.getroot()
14
15     relevante_kanten = []
16
17     for edge in root.findall("edge"):
18         name = edge.get("name")
19         if name:
20             for begriff in suchbegriffe:
21                 if begriff.lower() in name.lower():
22                     relevante_kanten.append((edge.get("id"), name))
23                     break # nicht doppelt eintragen, falls mehrere Begriffe
24                         ↪ passen
25
26     return relevante_kanten
27
28 if __name__ == "__main__":
29     kanten = finde_relevante_kanten(NET_FILE, SUCHBEGRIFFE)
30
31     # Ergebnisse speichern
32     with open(OUTPUT_FILE, "w", encoding="utf-8") as f:
33         for edge_id, name in kanten:
34             f.write(f"{edge_id}\t{name}\n")
35
36     print(f"{len(kanten)} relevante Kanten gefunden.")
37     print(f"Ergebnisse in '{OUTPUT_FILE}' gespeichert.")

```

F Sumo-Konfiguration

F.1 `sumoconfig_.sumocfg`

Die folgende Konfigurationsdatei definiert die zentralen Eingaben und Parameter für die Simulation in SUMO. Sie verweist auf die zu ladende Netzdatei und die zugehörigen Routendateien sowie auf den zu simulierenden Zeitraum.

```
1 <configuration>
2   <input>
3     <net-file value="network.net.xml"/>
4     <route-files value="routes.xml"/>
5   </input>
6   <time>
7     <begin value="0"/>
8     <end value="5000"/>
9   </time>
10 </configuration>
```

Literatur

- [1] R. Agand und et al. „EcoLight: Reward Shaping in Deep Reinforcement Learning for Ergonomic Traffic Signal Control“. In: *arXiv preprint* (2021). URL: <https://s3.us-east-1.amazonaws.com/climate-change-ai/papers/neurips2021/43/paper.pdf>.
- [2] L. N. Alegre und et al. „Quantifying the impact of non-stationarity in reinforcement learning-based traffic signal control“. In: *PeerJ Computer Science* (2021). URL: <https://doi.org/10.7717/peerj-cs.589>.
- [3] A. Almeida und et al. „Multiagent Reinforcement Learning for Traffic Signal Control: a k-Nearest Neighbors Based Approach“. In: *CEUR Workshop Proceedings* (2022). URL: <https://ceur-ws.org/Vol-3173/3.pdf>.
- [4] A. Ault und G. Sharon. „Reinforcement Learning Benchmarks for Traffic Signal Control“. In: *OpenReview* (2021). URL: <https://openreview.net/forum?id=LqRSh6V0vR>.
- [5] *Baden-wuerttemberg*. URL: <https://www.baden-wuerttemberg.de/de/service/presse/pressemitteilung/pid/land-startet-testfeld-mit-ki-gesteuerten-ampeln>.
- [6] *Bundesanstalt für Straßenwesen*. URL: <https://edocs.tib.eu/files/e01fn19/166939879X.pdf>.
- [7] *Bundesanstalt für Straßenwesen (BASt)*. URL: <https://www.bast.de>.
- [8] *Datenschutz-Grundverordnung*. URL: <https://dsgvo-gesetz.de/>.
- [9] *Dauerzählstellen Ergebnisse*. URL: https://mobidata-bw.de/vm/Ergebnisse_Ganglinien_Dauerzaehlstellen_BW/Ergebnisse_2023_PDF/VZ_2023_01-06.pdf.
- [10] *duarouter*. URL: <https://sumo.dlr.de/docs/duarouter.html>.
- [11] *Europäische Umweltagentur*. URL: https://www.kba.de/DE/Presse/Pressemitteilungen/Nr1Segmente/2025/pm33_2025_nr1_seg_07_25_komplett.html.
- [12] H. Ghanadbashi und et al. „Handling uncertainty in self-adaptive systems: an ontology-based reinforcement learning model“. In: *Springer* (2023). URL: <https://link.springer.com/article/10.1007/s40860-022-00198-x>.
- [13] *Google Maps Traffic API*. URL: <https://developers.google.com/maps/documentation/traffic>.
- [14] *Gymnasium*. URL: <https://gymnasium.farama.org/index.html>.
- [15] S. Hwang und et al. „Information-Theoretic State Space Model for Multi-View Reinforcement Learning“. In: *OpenReview* (2023). URL: <https://openreview.net/forum?id=jwy77xkyPt>.
- [16] *Inrix Traffic Scorecard*. URL: <https://inrix.com/scorecard/>.
- [17] *JOSM – Java OpenStreetMap Editor*. URL: <https://josm.openstreetmap.de>.
- [18] *KI-Ampeln in BW*. URL: <https://www.baden-wuerttemberg.de/de/service/presse/pressemitteilung/pid/land-startet-testfeld-mit-ki-gesteuerten-ampeln>.
- [19] *Kooperative Lichtsignalsteuerung: Integration von Fahrzeugen in die Steuerung vernetzter Verkehrssysteme*. URL: <https://mediatum.ub.tum.de/doc/1382853/123509.pdf>.

- [20] *Kraftfahrt-Bundesamt*. URL: https://www.kba.de/DE/Presse/Pressemitteilungen/Nr1Segmente/2025/pm33_2025_nr1_seg_07_25_komplett.html.
- [21] *Künstliche Intelligenz für LichtSignalAnlagen*. URL: <https://www.iosb-ina.fraunhofer.de/de/geschaeftsbereiche/maschinelles-lernen/forschungsthemen-und-projekte/projekt-KI4LSA.html>.
- [22] *Landesanstalt für Umwelt Baden-Württemberg*. URL: <https://www.lubw.baden-wuerttemberg.de/luft/verkehrsemissionen>.
- [23] *Low-traffic Amsterdam*. URL: <https://openresearch.amsterdam/en/page/49784/low-traffic-amsterdam>.
- [24] *MobiData BW – Mobilitätsdatenplattform Baden-Württemberg*. URL: <https://www.mobidata-bw.de>.
- [25] *MobiData BW: Karte der Dauerzählstellen im Straßenverkehr*. URL: https://mobidata-bw.de/dataset/karte_strassenverkehrszaehlung.
- [26] *MobiData BW: Stundenwerte an Dauerzählstellen (Straßenverkehr)*. URL: https://mobidata-bw.de/dataset/stundenwerte_dauerzaehlstellen.
- [27] *netconvert*. URL: <https://sumo.dlr.de/docs/netconvert.html>.
- [28] *netedit*. URL: <https://sumo.dlr.de/docs/Netedit/index.html>.
- [29] *netgenerate*. URL: <https://sumo.dlr.de/docs/netgenerate.html>.
- [30] *OpenStreetMap*. URL: <https://www.openstreetmap.org>.
- [31] *OpenStreetMap Wiki: Export*. URL: <https://wiki.openstreetmap.org/w/index.php?title=Export&oldid=2822860>.
- [32] *OSM Traffic signals*. URL: https://wiki.openstreetmap.org/wiki/DE:Tag:highway%3Dtraffic_signals.
- [33] *OSM-Git*. URL: <https://github.com/openstreetmap>.
- [34] *OSM-Guide*. URL: https://wiki.openstreetmap.org/wiki/Beginners%27_guide.
- [35] *PettingZoo*. URL: <https://pettingzoo.farama.org/>.
- [36] *Proximal Policy Optimization*. URL: <https://spinningup.openai.com/en/latest/algorithms/ppo.html>.
- [37] *Proximal Policy Optimization Algorithms*. URL: <https://arxiv.org/pdf/1707.06347>.
- [38] *Proximal Policy Optimization with Adaptive Exploration*. URL: <https://arxiv.org/html/2405.04664v1>.
- [39] *randomTrips*. URL: <https://sumo.dlr.de/docs/Tools/Trip.html>.
- [40] *Reinforcement Learning + traffic microsimulation (via SUMO)*. URL: <https://github.com/mschrader15/reinforcement-learning-sumo?tab=readme-ov-file>.
- [41] M. Reza und et al. „A citywide TD-learning based intelligent traffic signal control for autonomous vehicles: Performance evaluation using SUMO“. In: *Wiley Online Library* (2023). URL: <https://onlinelibrary.wiley.com/doi/full/10.1111/exsy.13301>.
- [42] *Stable-Baselines3*. URL: <https://stable-baselines3.readthedocs.io/en/master/>.

- [43] *Stau in Karlsruhe, bnn*. URL: <https://bnn.de/karlsruhe/karlsruhe-stadt/karlsruhe-prueft-neue-verkehrsfuehrung-am-kriegsstrassentunnel-zur-stau-entlastung>.
- [44] *Straßenverkehrszentrale Baden-Württemberg*. URL: <https://www.svz-bw.de>.
- [45] *Sumo - OSM import*. URL: <https://sumo.dlr.de/docs/Networks/Import/OpenStreetMap.html>.
- [46] *Sumo - Traffic Light System*. URL: https://sumo.dlr.de/docs/Simulation/Traffic_Lights.html.
- [47] *SUMO Dokumentation*. URL: <https://sumo.dlr.de>.
- [48] *SUMO Toolchain: Netzwerk- und Routengeneratoren*. URL: <https://sumo.dlr.de/docs/Tools.html>.
- [49] *Sumo-Git*. URL: <https://github.com/eclipse-sumo/sumo>.
- [50] *SUMO-Gui*. URL: <https://sumo.dlr.de/daily/userdoc/sumo-gui.html>.
- [51] *Sumo-HBEFA*. URL: <https://sumo.dlr.de/docs/Models/Emissions/HBEFA-based.html>.
- [52] *Sumo-Publikation*. URL: https://sumo.dlr.de/pdf/sysmea_v5_n34_2012_4.pdf.
- [53] *Sumo-rl docs*. URL: <https://lucasalegre.github.io/sumo-rl/>.
- [54] *Sumo-rl TLS-interface*. URL: https://github.com/LucasAlegre/sumo-rl/blob/main/sumo_rl/environment/traffic_signal.py.
- [55] *sumo-rl: GitHub Repository*. URL: <https://github.com/LucasAlegre/sumo-rl>.
- [56] *SuperSuit*. URL: <https://github.com/Farama-Foundation/SuperSuit>.
- [57] *SURTRAC*. URL: https://www.ri.cmu.edu/pub_files/2013/1/13-0315.pdf.
- [58] *Sutton, Reinforcement Learning: An Introduction*. URL: <http://incompleteideas.net/book/the-book-2nd.html>.
- [59] *TensorBoard*. URL: <https://www.tensorflow.org/tensorboard>.
- [60] *TomTom Traffic API*. URL: <https://developer.tomtom.com/traffic-api>.
- [61] *TraCI-Interface*. URL: https://sumo.dlr.de/docs/TraCI/Interfacing_TraCI_from_Python.html.
- [62] *Traffic Flow Theory – A State-of-the-Art Report*. URL: https://rosap.nhtl.bts.gov/view/dot/35775/dot_35775_DS1.pdf.
- [63] *Umweltbundesamt*. URL: <https://www.umweltbundesamt.de/daten/private-haushalte-konsum/mobilitaet-privater-haushalte#-hoher-motorisierungsgrad>.
- [64] *Umweltbundesamt*. URL: <https://www.umweltbundesamt.de/daten/verkehr/emissionen-des-verkehrs#pkw-fahren-heute-klima-und-umweltvertraglicher>.
- [65] *Umweltbundesamt*. URL: <https://www.umweltbundesamt.de/themen/luft/luftschadstoffe/feinstaub/umweltzonen-in-deutschland>.
- [66] *VecMonitor*. URL: https://stable-baselines3.readthedocs.io/en/master/_modules/stable_baselines3/common/vec_env/vec_monitor.html.
- [67] *VecNormalize*. URL: https://stable-baselines3.readthedocs.io/en/master/guide/vec_envs.html.

- [68] *Verfahren zur dynamischen Verkehrsumlegung*. URL: https://www.isv.uni-stuttgart.de/vuv/publikationen/downloads/200503_Fr_DynUmlg-SVT.pdf.
- [69] *Verkehrszählungen in Baden-Württemberg*. URL: <https://www.lubw.baden-wuerttemberg.de/luft/verkehrszaehlungen-in-baden-wuerttemberg#karte>.
- [70] Y. Zheng und et al. „From Local to Global: A Curriculum Learning Approach for Reinforcement Learning-based Traffic Signal Control“. In: *IEEE Xplore* (2022). URL: <https://ieeexplore.ieee.org/abstract/document/9832372>.