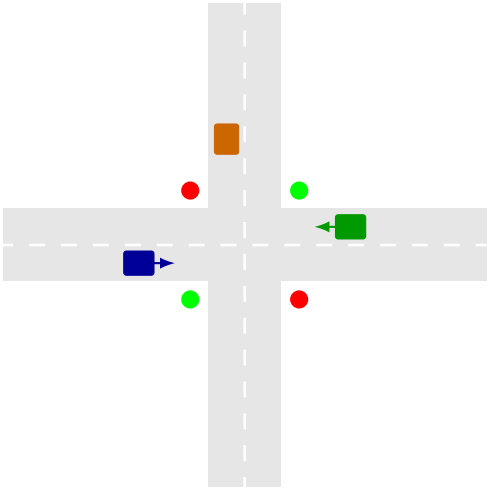
Hochschule Karlsruhe – Technik und Wirtschaft

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 22. August 2025

Inhaltsverzeichnis

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-rl`[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]

Insbesondere 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 realiätsnahem 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 CO_2-, 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]

$$\text{obs} = [\text{phase\_one\_hot}, \text{min\_green}, \text{lane}_1\text{\_density}, \ldots, \text{lane}_n\text{\_density}, \text{lane}_1\text{\_queue}, \ldots, \text{lane}_n\text{\_queue}]$$

Diese Komponenten bedeuten im Detail:

- **phase_one_hot**: One-Hot-Kodierung der aktuell aktiven Grünphase.
- **min_green**: Binärwert, der anzeigt, ob die Mindestdauer (`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 $< 0{,}1\,\text{m/s}$ 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]).

Abbildung 2: Screenshot des ursprünglichen OpenStreetMap-Ausschnitts (OpenStreetMap [30]).

Die Wahl von OpenStreetMap als Datenquelle gewährleistet eine offene, reproduzierbare und erweiterbare Modellierungsbasis. Jedoch bringt die Nutzung von OSM-Daten auch einige Einschränkungen mit sich, die bei der Modellierung berücksichtigt werden müssen:[31, 30, 34]

- **Uneinheitlicher Detaillierungsgrad:** Die Erfassungstiefe variiert regional stark, was dazu führt, dass z. B. Tempolimits, Fahrspuren oder Abbiegebeschränkungen an vielen Stellen fehlen oder unvollständig sind.
- **Fehlende Ampel- und Signalsteuerungsdaten:** OSM enthält in der Regel keine vollständigen Angaben zu Ampelphasen, Umlaufzeiten oder koordinierter Schaltung. SUMO kann zwar aus heuristischen Annahmen Standardampeln generieren, diese weichen jedoch potenziell stark von der realen Steuerung ab.
- **Keine garantierte Netzvollständigkeit:** Besonders kleinere Straßen, private Zufahrten oder temporäre Baustellen sind häufig nicht oder nur unzureichend erfasst. Zudem treten beim Zuschnitt von Kartenausschnitten an den Netzrändern regelmäßig unvollständige Knoten oder isolierte Kanten auf.
- **Abweichende Modellierungskonzepte:** In OSM werden parallele Fahrbahnen oder getrennte Richtungsfahrbahnen oft als unabhängige Wege modelliert. Ohne geeignete Nachbearbeitung kann dies zu unnötigen Knoten und ineffizientem Verkehrsverhalten führen.
- **Abhängig von Typemap- und Importoptionen:** Die Interpretation der OSM-Tags erfolgt in SUMO durch sogenannte Typemaps, die z. B. Tempolimits und Spuranzahl je nach Straßentyp zuweisen. Ohne geeignete Typemap kann das Verhalten nicht der Realität entsprechen. [27]

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 Zähldaten, 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 (DTV[1]), 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

[1] DTV steht für *Durchschnittlicher Tagesverkehr* und bezeichnet die mittlere Anzahl an Fahrzeugen, die einen bestimmten Straßenabschnitt pro Tag passiere, 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.

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 Verkehrszähldaten:** 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 Zähldaten 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 Verkehrszähldaten

Für die Kalibrierung und Validierung der Verkehrssimulation wurden verschiedene reale Zähldatenquellen 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 MobiData BW ermöglichen eine Kontextualisierung der aktuellen Verkehrsbelastung. [25]
- **Zählstellen an äußeren Zufahrtsachsen:** Ergänzende Zähldaten aus dem Jahr 2023 an acht stark befahrenen Einfallstraßen (u. a. B10, B36, L605)[25] liefern Anhaltspunkte zur Verkehrsstärke an den Netzrändern.

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 Verkehrszähldaten (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 5000 s 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 Kaptitel 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 $v < 0.1\,\mathrm{m/s}$),
- Durchschnittsgeschwindigkeit pro Spur,
- aktuelle Ampelphase (TLS state),
- Dauer der aktuellen Phase,

- Binärmasken für die Phasenwechselbarkeit[3],
- 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 ($A = \{\texttt{keep}, \texttt{switch}\}$).
2. **Direktwahl-Modell:** Der Agent wählt direkt eine der möglichen Phasen ($A = \{\texttt{phase}_0, \ldots, \texttt{phase}_n\}$).

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.

Die Variante `realworld` berücksichtigt ausschließlich Metriken, die in realen Städten mit vergleichsweise einfacher Sensorik (z.,B. Induktionsschleifen, Kameras) erhoben werden können. Die übrigen Varianten (`diff-waiting-time`, `queue`, `emissions`) dienen der Evaluation unterschiedlicher Optimierungsziele und wurden in Vergleichsstudien eingesetzt (siehe Anhang B).

[3] Die 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 CO_2-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.

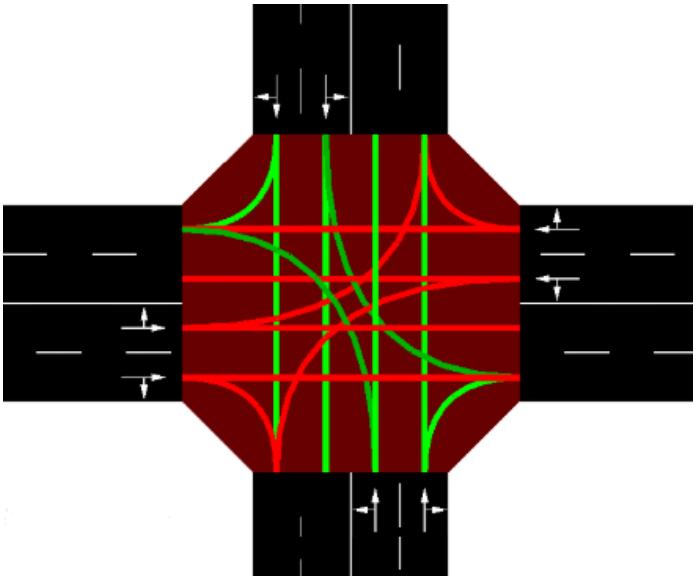


Abbildung 4: Visualisierung einer TLS-Kreuzung (Quelle nededit[28]).

- `<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=<tls_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.

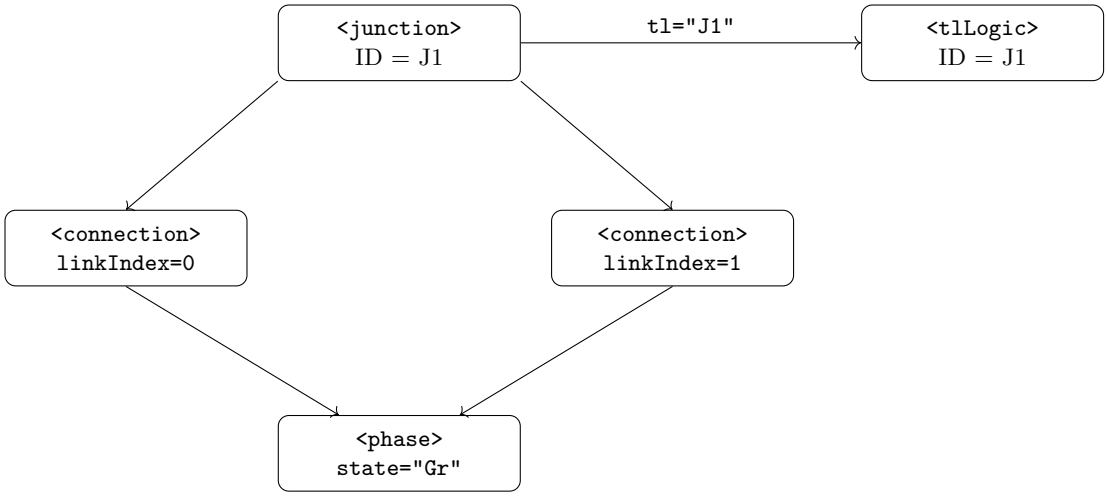


Abbildung 5: Zusammenspiel von Kreuzung, Verbindungen und Ampellogik in SUMO

4.4.2 Typische Fehlerquellen nach OSM-Import

Die automatische Ableitung von Ampelsteuerungen aus OSM ist unvollständig und fehleranfällig. Im Zusammenspiel mit `sumo-rl` 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-rl`, 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-rl` 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-rl` 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.

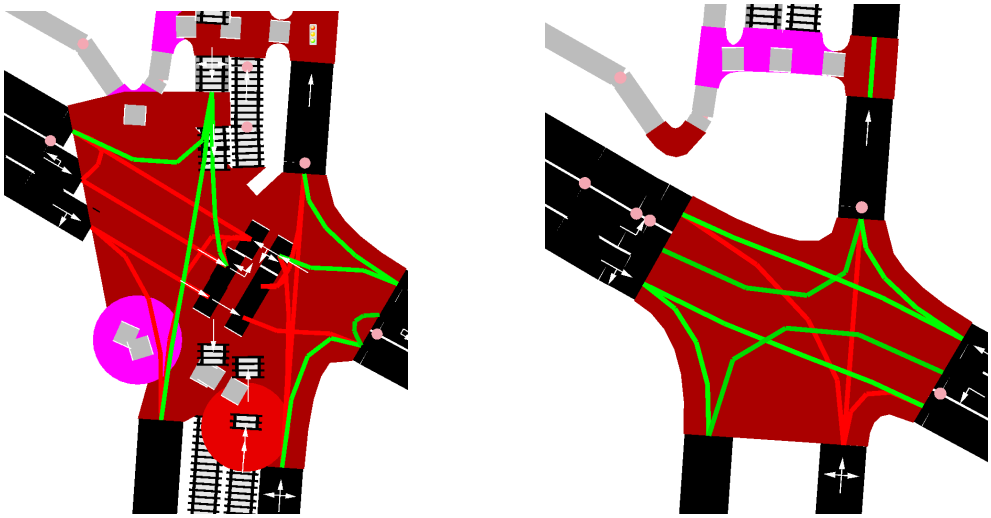
Beseitigung von Deadlocks Während der initialen Simulationen traten in SUMO wiederholt Deadlocks an komplexen Kreuzungen auf, die in der realen Verkehrsführung nicht vorkommen. Ursache waren insbesondere von OSM importierte Rampen- und Abbiegebeziehungen, die zu konfliktbehafteten Fahrbeziehungen führten.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



Vorher Nachher

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-rl` 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 CheckTLSLengths – Prüfung inkonsistenter Phasenlängen

```
1: function CheckTLSLengths(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 ← 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 ← Länge des state-Strings
12:            if actualLen ≠ 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 ← ID der Junction
6:         if tls_id in tls_signal_indices then
7:             expected_max ← Länge der Signalindizes für dieses TLS
8:             for all request-Elemente in Junction do
9:                 index ← Wert des index-Attributs
10:                if index ≥ 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

```
function FixRequests(net.xml)
    Lade XML-Baum mit Netzstruktur
    Initialisiere Dictionary tls_max_index für maximale Signalindices
    for all connection-Elemente do
        if TLS-ID und linkIndex vorhanden then
            Aktualisiere tls_max_index[tl] mit höchstem Index
        end if
    end for
    for all junction-Elemente do
        Hole TLS-ID
        if TLS nicht in tls_max_index then
            continue
        end if
        Bestimme erlaubten Maximalindex (max_idx)
        for all request-Einträge do
            if Index > max_idx then
                Entferne ungültigen request
            end if
        end for
        for all tlLogic-Elemente mit passender TLS-ID do
            for all Phasen do
                if state-String ist zu lang then
                    Kürze state auf max_idx + 1
                end if
            end for
        end for
    end for
    Speichere modifizierte XML-Datei
    Gib Statistiken zu entfernten Requests und angepassten Phasen aus
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

```
function RepairTLSStates(net.xml, referenz_dictionary)
    Lade Netzstruktur aus net.xml
    for all tlLogic-Elemente im Netz do
        tls_id ← ID des Ampelknotens
        if tls_id nicht in referenz_dictionary then
            continue
        end if
        correctLen ← erwartete Zustandslänge aus Referenz
        for all Phasen des Knotens do
            state ← Zeichenkette der Phase
            if Länge(state) ≠ correctLen then
                Kürze oder ergänze state auf correctLen
                Markiere Netz als geändert
            end if
        end for
    end for
    if Netz wurde geändert then
        Speichere bereinigte Netzdatei als karlsruhe_fixed.net.xml
    else
        Gib Hinweis: Alle Phasen bereits korrekt
    end if
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

```
function StateCheck(net.xml)
    Lade XML-Baum aus der Netzdatei
    for all tlLogic-Elemente im Netz do
        tl_id ← ID des aktuellen TLS
        for all Phasen i in tlLogic do
            state ← Zustand der Phase
            if len(state) ≠ 57 then
                Gib Warnung mit tl_id, Phasenindex und tatsächlicher Länge aus
            end if
        end for
    end for
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

```
function TestTLS(tls_id)
    Initialisiere SumoEnvironment
    Setze ts_ids auf [tls_id]
    Versuche: env.reset()
    if kein Fehler then
        env.close()
        return True
    else
        Gib Fehlermeldung aus
        return False
    end if
end function
Initialisiere leere Liste all_tls
Versuche: Umgebung mit SumoEnvironment zu starten
if erfolgreich then
    Lese alle ts_ids
    Schließe Umgebung
else
    Gib Fehler aus
end if
Initialisiere leere Liste valid_tls
for all tls_id in all_tls do
    if TestTLS(tls_id) then
        Füge tls_id zu valid_tls hinzu
    end if
end for
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]

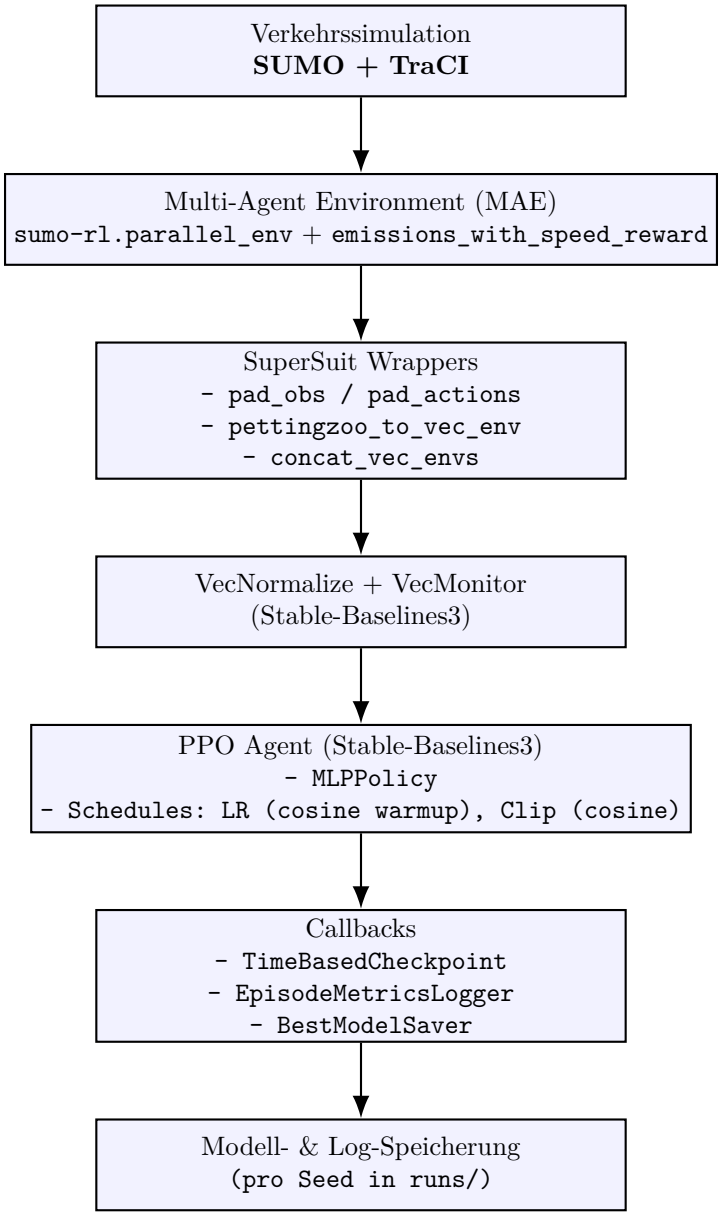


Abbildung 7: Architektur der RL-Trainingspipeline mit SUMO, MAE und Stable-Baselines3

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, CO_2-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 diente 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.

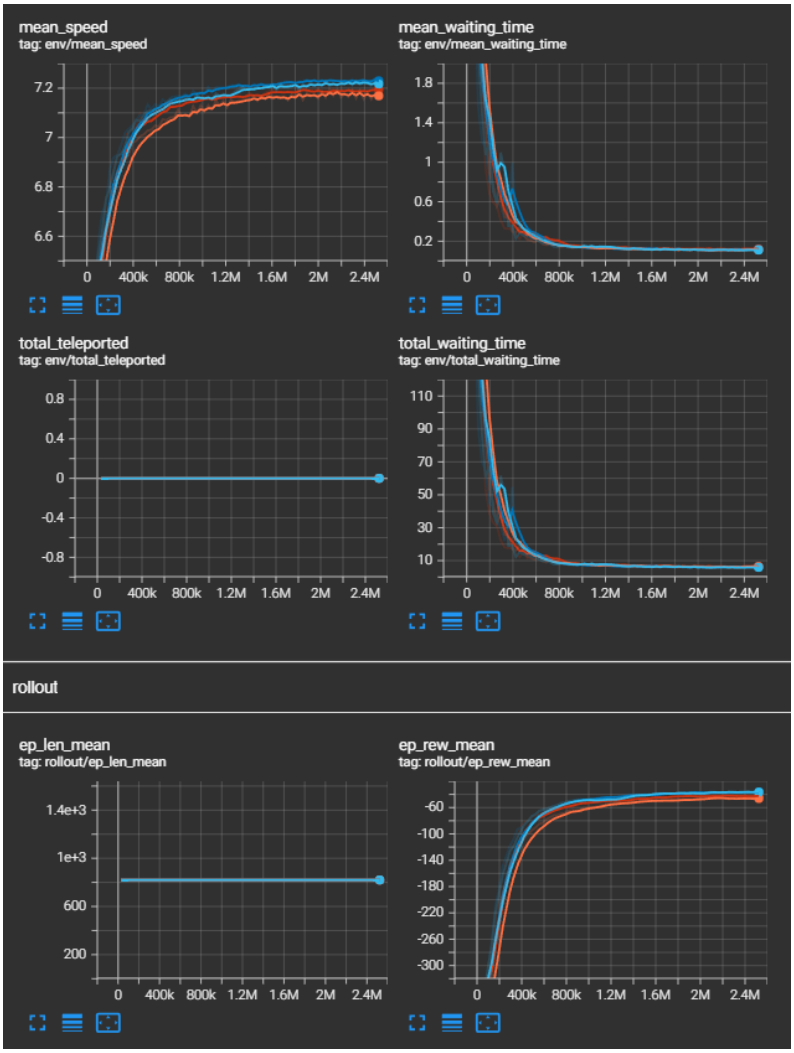


Abbildung 8: Beispielhafte Lernkurve eines Trainingslaufs, visualisiert mit TensorBoard. Der Reward stabilisiert sich nach einer Explorationsphase.

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 diente 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-Parallellisierung 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):

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

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 realtiv 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

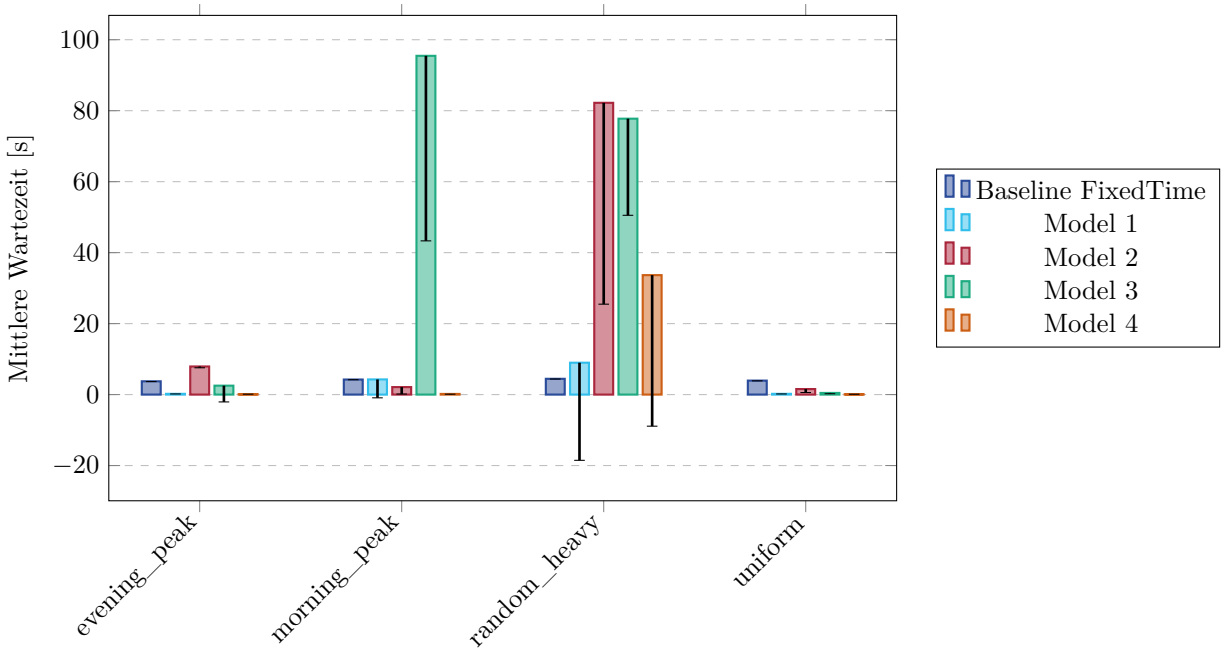


Abbildung 9: Mittlere Wartezeiten

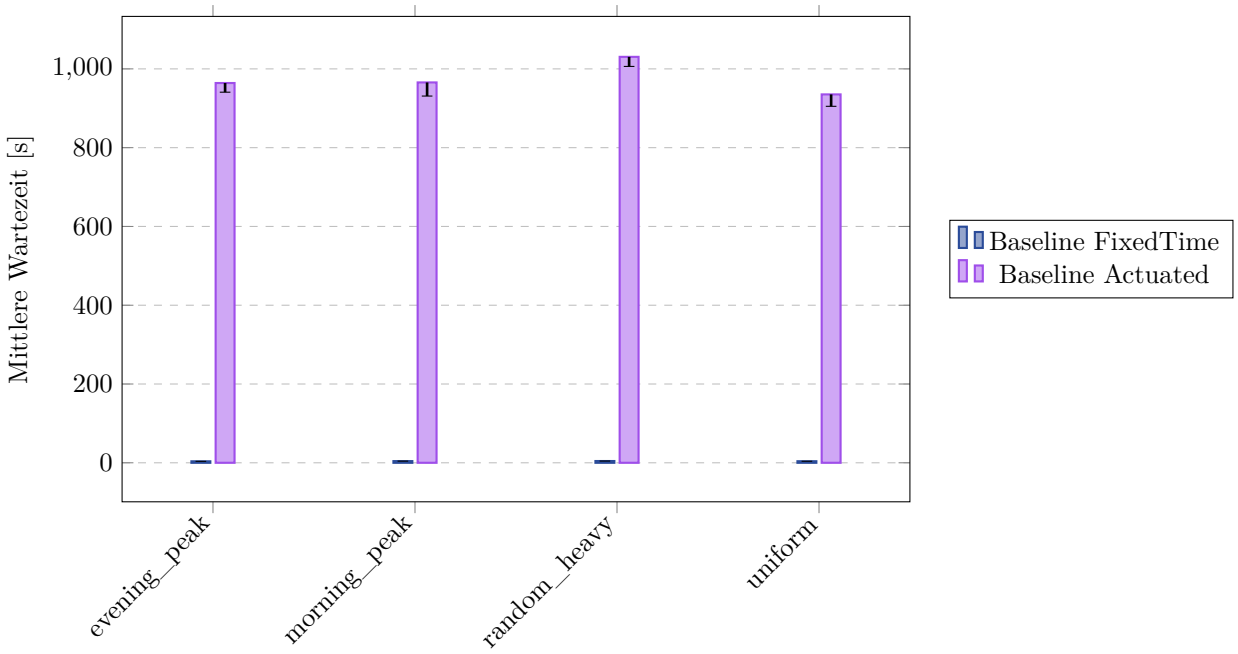


Abbildung 10: Mittlere Wartezeiten

Die mittlere Wartezeit für die Diff-Waiting-Time-Rewardfunktion zeigt erneut deutliche Unterschiede zwischen den Baselines und den trainierten Modellen.

Die Fixed-Time-Baseline erreicht im morning_peak 3.74 s, im evening_peak 4.20 s, im random_heavy 4.40 s und im uniform-Szenario 3.90 s. Diese Werte bilden eine stabile Referenz. Die Actuated-Baseline fällt dagegen erneut durch extrem hohe Wartezeiten auf: 964 s, 965 s, 1030 s und 935 s in den vier Szenarien.

Unter den trainierten Modellen zeigen sich differenzierte Muster. Modell 1 erreicht sehr geringe Werte mit 0.20 s, 4.27 s, 8.90 s und 0.21 s. Modell 2 weist im morning_peak mit 7.9 s sowie im random_heavy mit 82 s deutlich höhere Wartezeiten auf, während die Werte im evening_peak (2.1 s) und im uniform (1.5 s) niedrig bleiben. Modell 3 zeigt ebenfalls inkonsistente Ergebnisse: 2.5 s im morning_peak, aber 95 s im evening_peak und 77 s im random_heavy, während das uniform-Szenario mit 0.4 s vergleichsweise gut abschneidet. Modell 4 erzielt insgesamt die besten Resultate mit 0.11 s, 0.18 s, 33 s und 0.10 s, wenngleich auch hier im random_heavy ein Anstieg erkennbar ist.

Die erhöhten Mittelwerte in einzelnen Szenarien, insbesondere bei Modell 2 und Modell 3, gehen jeweils mit einer hohen Standardabweichung einher. Dies weist auf eine deutliche Instabilität zwischen den Runs hin und deutet darauf, dass die Modelle zwar vereinzelt effiziente Strategien entwickeln konnten, diese jedoch nicht konsistent reproduziert werden.

5.1.2 Anzahl stoppender Fahrzeuge

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

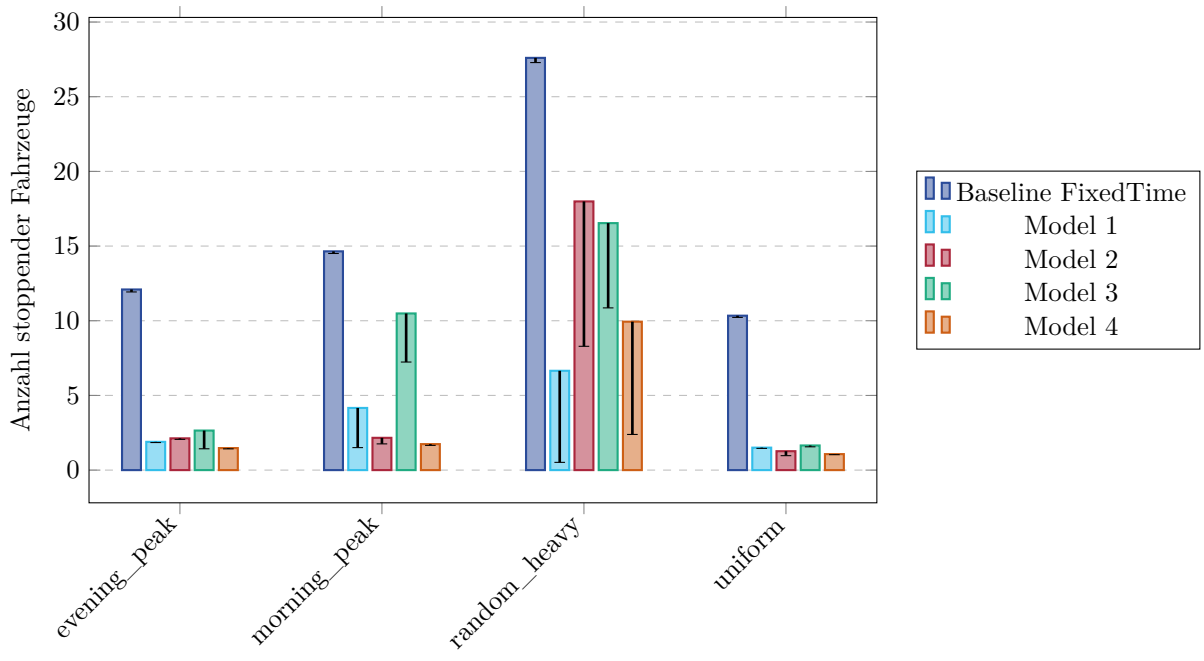


Abbildung 11: Anzahl stoppender Fahrzeuge

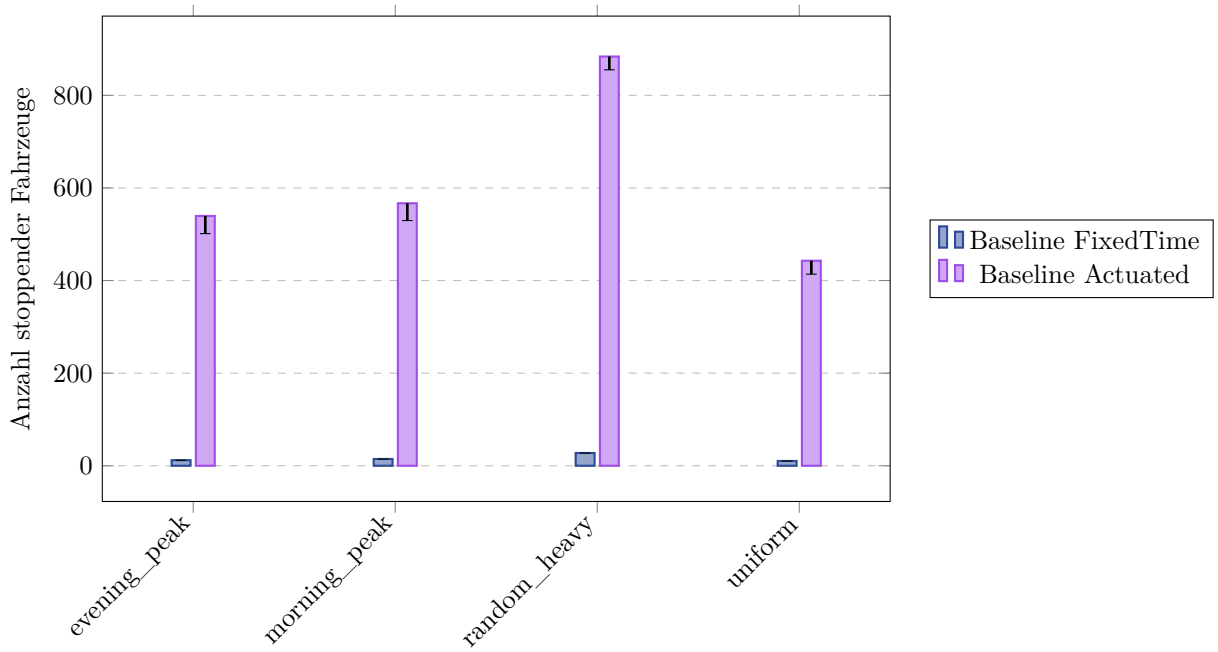


Abbildung 12: Anzahl stoppender Fahrzeuge

Die Ergebnisse zur Anzahl stoppender Fahrzeuge zeigen deutliche Unterschiede zwischen den Baselines und den trainierten Modellen.

Die Fixed-Time-Baseline erreicht im morning_peak durchschnittlich 12 Fahrzeuge, im evening_peak 14 Fahrzeuge, im random_heavy 27 Fahrzeuge und im uniform-Szenario 10 Fahrzeuge. Diese Werte bilden eine stabile und vergleichsweise effiziente Referenz.

Die Actuated-Baseline zeigt dagegen eine massiv erhöhte Zahl an Stopps. So ergeben sich im morning_peak 539 Fahrzeuge, im evening_peak 566 Fahrzeuge, im random_heavy 883 Fahrzeuge und im uniform 442 Fahrzeuge. Damit bestätigt sich auch in dieser Metrik die unzureichende Leistungsfähigkeit der Actuated-Steuerung.

Die trainierten Modelle liefern deutlich geringere Werte. Modell 1 erreicht 1.8, 4.1, 6.6 und 1.5 Fahrzeuge in den vier Szenarien. Modell 2 erzielt mit 2.12, 2.1, 17 und 1.2 Fahrzeugen ebenfalls niedrige Werte, allerdings mit einem deutlichen Anstieg im random_heavy-Szenario. Modell 3 weist Werte von 2.64, 10, 16 und 1.6 Fahrzeugen auf, wobei insbesondere im evening_peak und im random_heavy eine Verschlechterung gegenüber den übrigen Szenarien erkennbar ist. Modell 4 erzielt mit 1.4, 1.7, 9.9 und 1.06 Fahrzeugen die insgesamt besten Resultate und bleibt in allen Szenarien unterhalb der Fixed-Time-Baseline.

Die erhöhten Werte bei Modell 2 und Modell 3 im random_heavy-Szenario gehen mit einer hohen Standardabweichung einher, was auf starke Schwankungen zwischen den Runs hindeutet. Dies verdeutlicht, dass die Modelle zwar in der Lage sind, den Verkehrsfluss erheblich zu verbessern, ihre Robustheit unter unregelmäßigen Verkehrslasten jedoch eingeschränkt bleibt.

5.1.3 Anzahl ankommender Fahrzeuge

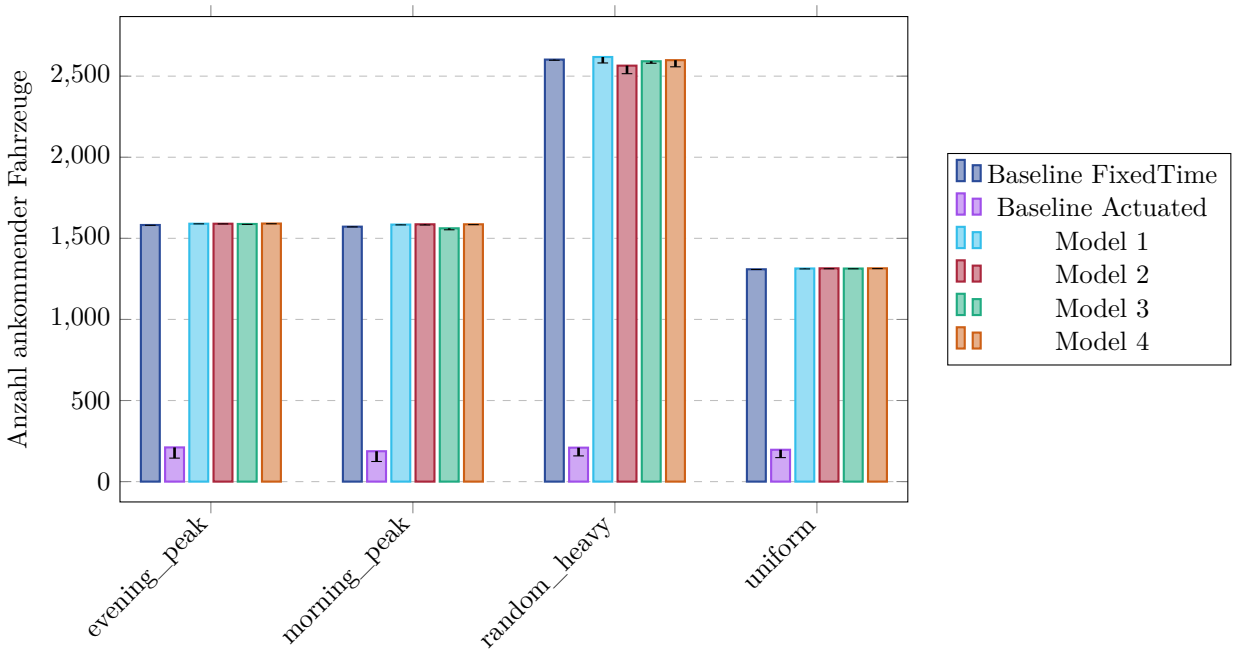


Abbildung 13: Anzahl ankommender Fahrzeuge

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

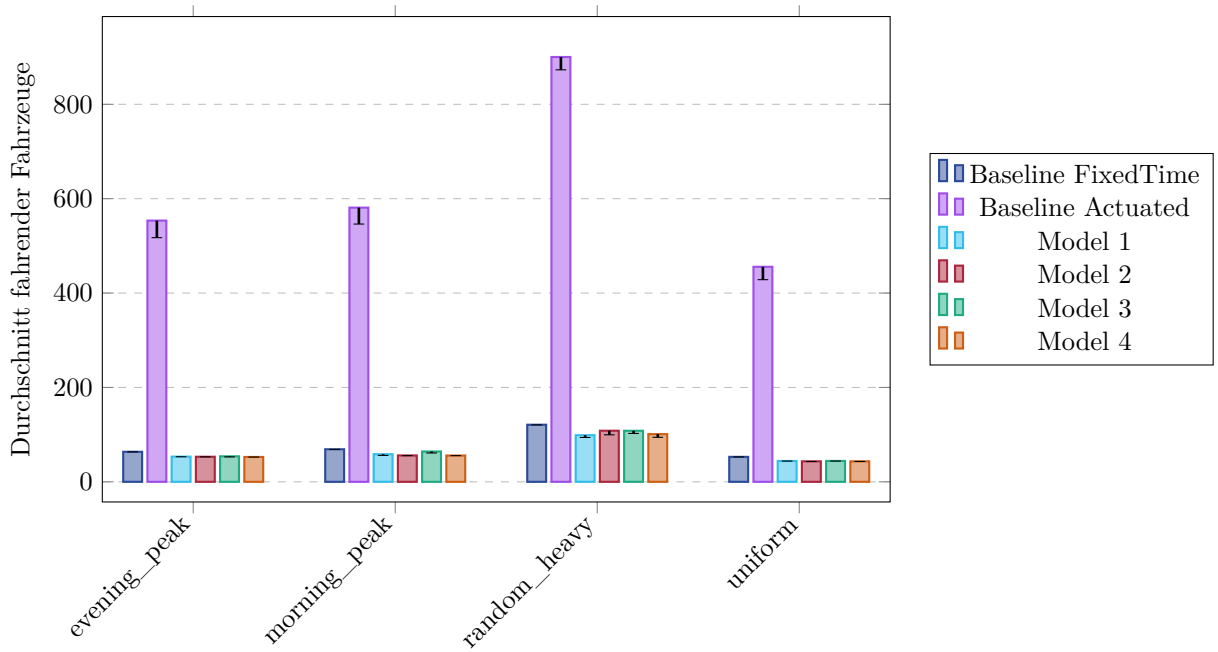


Abbildung 14: Durchschnitt fahrender Fahrzeuge

Die Auswertung des Durchschnitts fahrender Fahrzeuge (weniger ist besser) zeigt deutliche Unterschiede zwischen den Baselines und den trainierten Modellen.

Die Fixed-Time-Baseline erreicht im morning_peak durchschnittlich 63 Fahrzeuge, im evening_peak 69 Fahrzeuge, im random_heavy 121 Fahrzeuge und im uniform-Szenario 52 Fahrzeuge. Diese Werte bilden eine stabile Referenz.

Die Actuated-Baseline weist deutlich höhere Werte auf, mit 553 Fahrzeugen im morning_peak, 481 Fahrzeugen im evening_peak, 900 Fahrzeugen im random_heavy und 455 Fahrzeugen im uniform-Szenario. Damit bestätigt sich das schwache Abschneiden dieser Steuerung auch in dieser Metrik.

Die trainierten Modelle erreichen insgesamt Werte, die sehr nahe an der Fixed-Time-Baseline liegen. Modell 1 erzielt 53, 58, 98 und 44 Fahrzeuge. Modell 2 liegt bei 53, 55, 108 und 43 Fahrzeugen. Modell 3 weist 54, 64, 108 und 44 Fahrzeuge auf, während Modell 4 mit 52, 55, 101 und 43 Fahrzeugen die niedrigsten Werte liefert.

Insgesamt zeigen die Modelle konsistent bessere oder vergleichbare Ergebnisse zur Fixed-Time-Baseline. Auffällig ist, dass im random_heavy-Szenario die Werte von Modell 2 und Modell 3 leicht über der Referenz liegen, was mit einer erhöhten Standardabweichung einhergeht und auf Instabilität in komplexen Verkehrslagen hinweist.

5.1.5 Durchschnittsgeschwindigkeiten

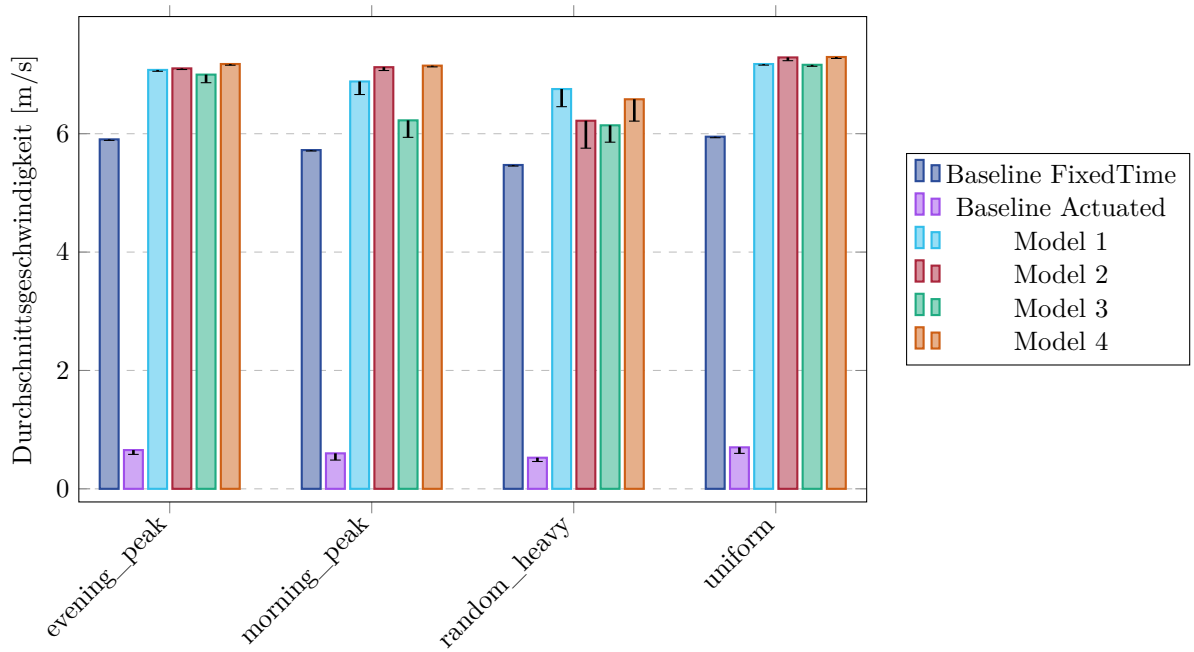


Abbildung 15: Durchschnittsgeschwindigkeiten

Die Analyse der Durchschnittsgeschwindigkeiten verdeutlicht klare Unterschiede zwischen den Baselines und den trainierten Modellen.

Die Fixed-Time-Baseline erreicht im morning_peak durchschnittlich 5.9 m/s, im evening_peak 5.7 m/s, im random_heavy 5.4 m/s sowie im uniform-Szenario 5.9 m/s. Damit liefert sie konsistente, aber nicht optimale Werte.

Die Actuated-Baseline weist durchgehend extrem niedrige Geschwindigkeiten auf: 0.6, 0.5, 0.5 und 0.7 m/s in den vier Szenarien. Diese Werte liegen um eine Größenordnung unterhalb der Fixed-Time-Baseline und bestätigen die unzureichende Leistungsfähigkeit dieser Steuerung.

Die trainierten Modelle übertreffen die Fixed-Time-Baseline deutlich. Modell 1 erreicht 7.0, 6.8, 6.7 und 7.1 m/s. Modell 2 erzielt sehr stabile Werte mit 7.1, 7.1, 6.2 und 7.2 m/s. Modell 3 liegt mit 6.9, 6.2, 6.1 und 7.1 m/s insgesamt ebenfalls über der Fixed-Time-Baseline, zeigt jedoch im evening_peak und random_heavy leicht reduzierte Ergebnisse. Modell 4 liefert mit 7.1, 7.1, 6.5 und 7.2 m/s die besten Resultate, insbesondere durch die hohe Stabilität über alle Szenarien hinweg.

Auffällig ist, dass die Modelle 2 und 4 durchgehend eine nahezu konstante Verbesserung gegenüber der Fixed-Time-Baseline erzielen, während Modell 3 in den Szenarien evening_peak und random_heavy leichten Performanceverlust zeigt. In diesen Fällen geht die Verschlechterung mit einer erhöhten Standardabweichung einher, was auf eine weniger stabile Performanz hindeutet.

5.1.6 Anzahl teleportierender Fahrzeuge

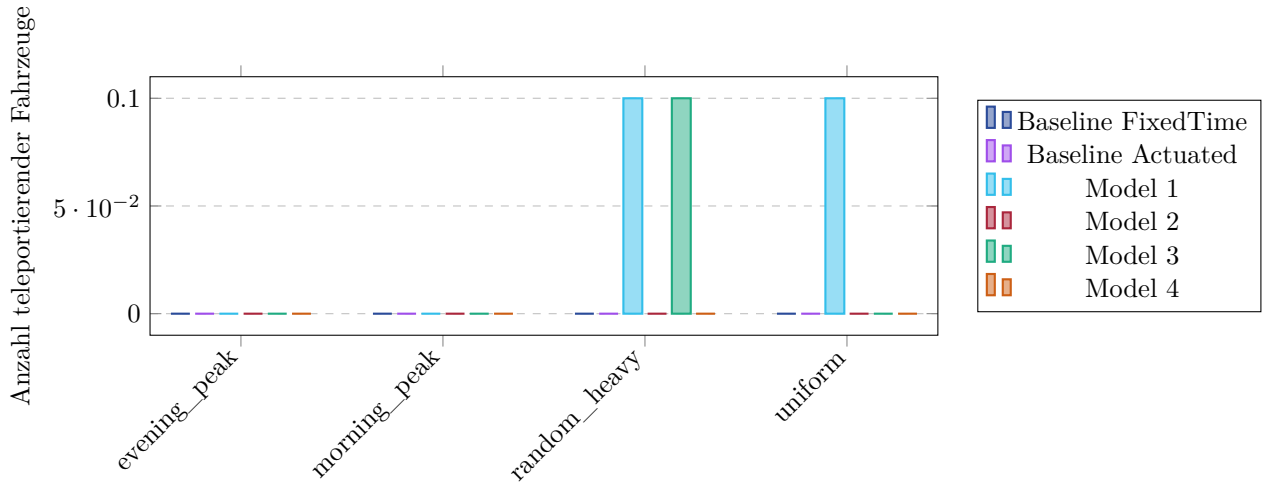


Abbildung 16: Anzahl teleportierender Fahrzeuge

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 Zahrzeuge

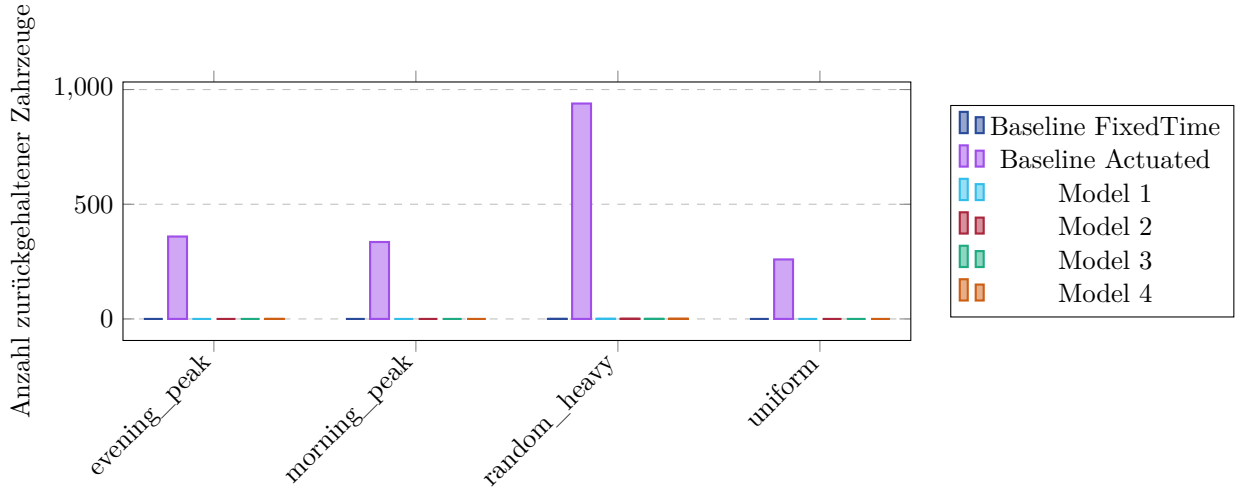


Abbildung 17: Anzahl zurückgehaltener Zahrzeuge

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

5.1.8 Einstufung

Die Modelle, die mit der Diff-Waiting-Time-Rewardfunktion trainiert wurden, zeigen in den meisten Metriken eine klare Überlegenheit gegenüber den Baselines. Die Fixed-Time-Baseline dient als stabile Referenz, wird jedoch in nahezu allen Szenarien von den Modellen deutlich übertroffen. Die Actuated-Baseline bestätigt auch hier ihre Schwäche und liefert durchweg die schlechtesten Ergebnisse.

Besonders auffällig ist die deutliche Reduktion der mittleren Wartezeiten und der Anzahl stoppender Fahrzeuge. Modelle 1 und 4 erreichen dabei die insgesamt besten Resultate und bleiben sowohl in regulären als auch in stark belasteten Szenarien konsistent unterhalb der Fixed-Time-Baseline. Modell 2 und Modell 3 zeigen ebenfalls Verbesserungen, jedoch treten in Szenarien mit hoher Verkehrslast (random_heavy, evening_peak) teils deutlich erhöhte Werte und zugleich hohe Standardabweichungen auf. Diese Schwankungen weisen auf eine geringere Stabilität der Strategien hin.

Hinsichtlich der Durchschnittsgeschwindigkeit erzielen alle Modelle höhere Werte als die Fixed-Time-Baseline, wobei insbesondere Modell 2 und Modell 4 durch ihre gleichmäßige Performance hervorstechen. Modell 3 fällt in einzelnen Szenarien leicht zurück, bleibt aber insgesamt dennoch über der Referenz.

Die Metriken zu Teleportationen und zurückgehaltenen Fahrzeugen bestätigen, dass das Netz nicht an seine absolute Kapazitätsgrenze gelangte. Einzelne Teleportationen bei Modell 1 und Modell 3 sind als Randereignisse zu werten und beeinflussen die Gesamteinstufung nicht maßgeblich.

Insgesamt lässt sich festhalten, dass die Diff-Waiting-Time-Rewardfunktion robuste Modelle hervorbringt, die klassische Steuerungen klar übertreffen. Modelle 1 und 4 überzeugen durchgängig mit stabiler Performance, während Modell 2 und Modell 3 unter hoher Verkehrslast anfälliger für Leistungseinbrüche und stärkere Varianz sind.

5.2 Reward: Queue

Die zweite Gruppe zielt auf die Minimierung der Warteschlangenlänge.

5.2.1 Mittlere Wartezeiten

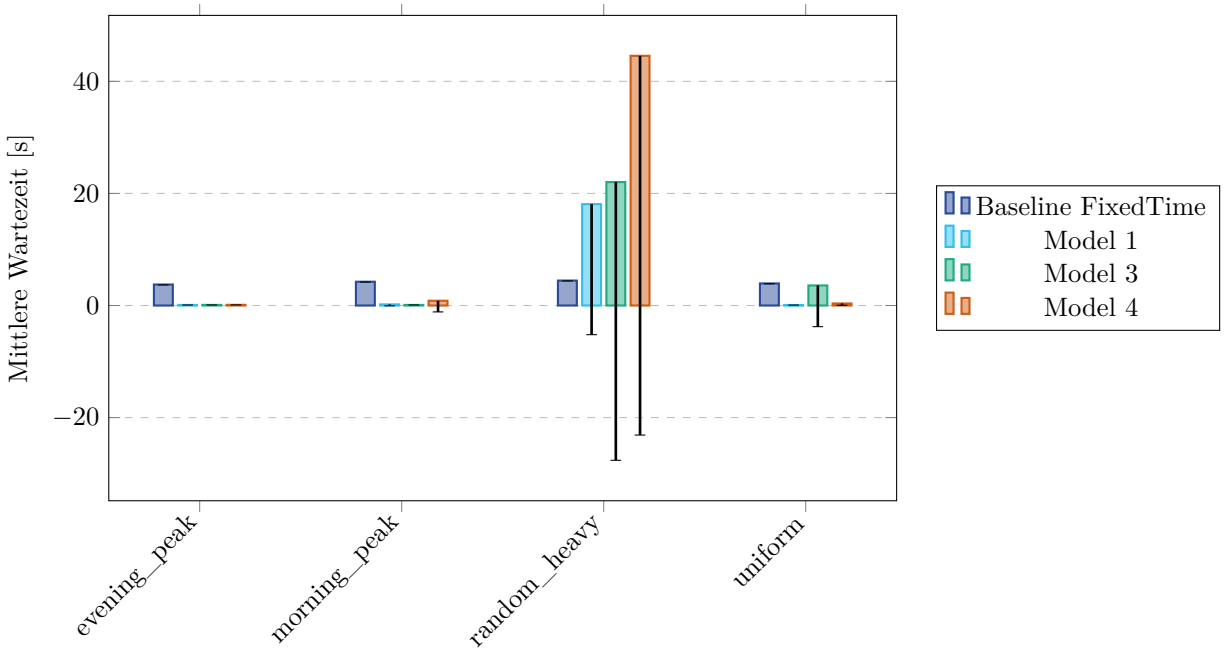


Abbildung 18: Mittlere Wartezeiten

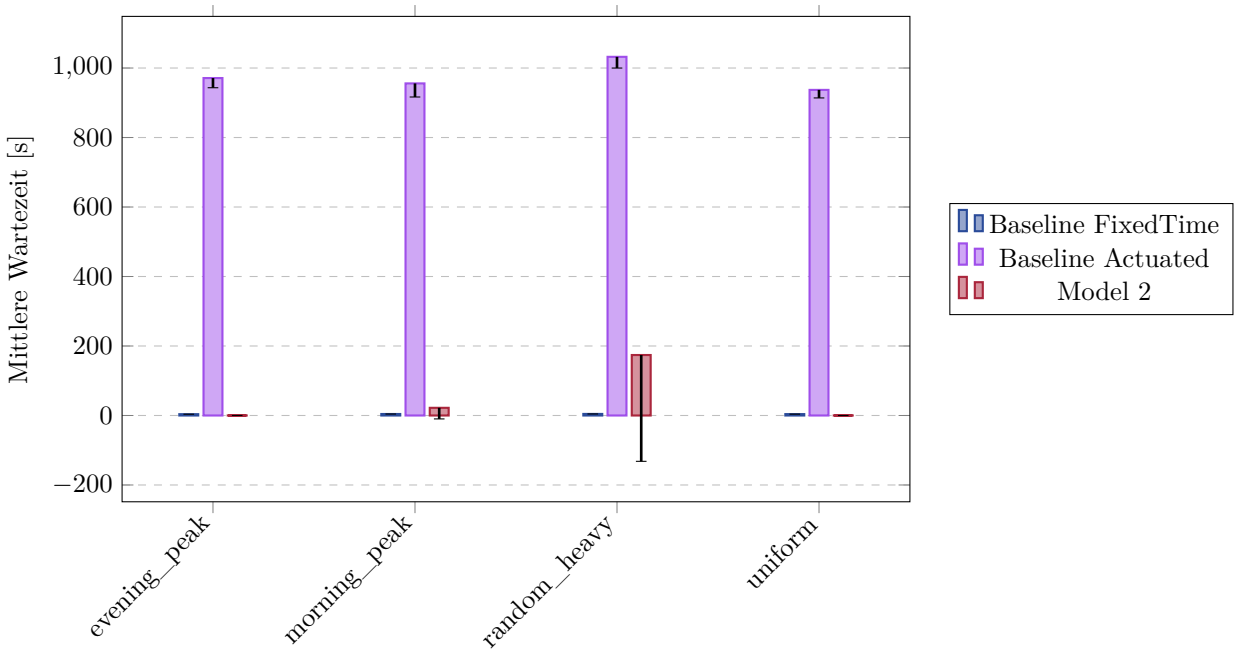


Abbildung 19: Mittlere Wartezeiten

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.

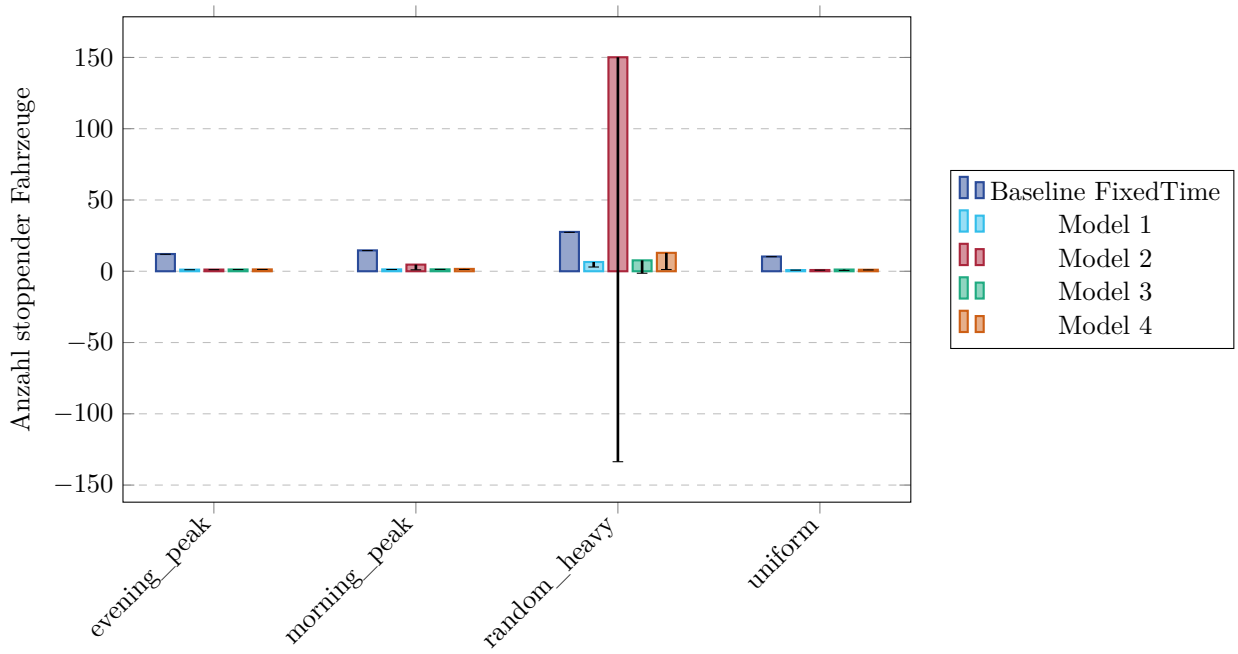


Abbildung 20: Anzahl stoppender Fahrzeuge

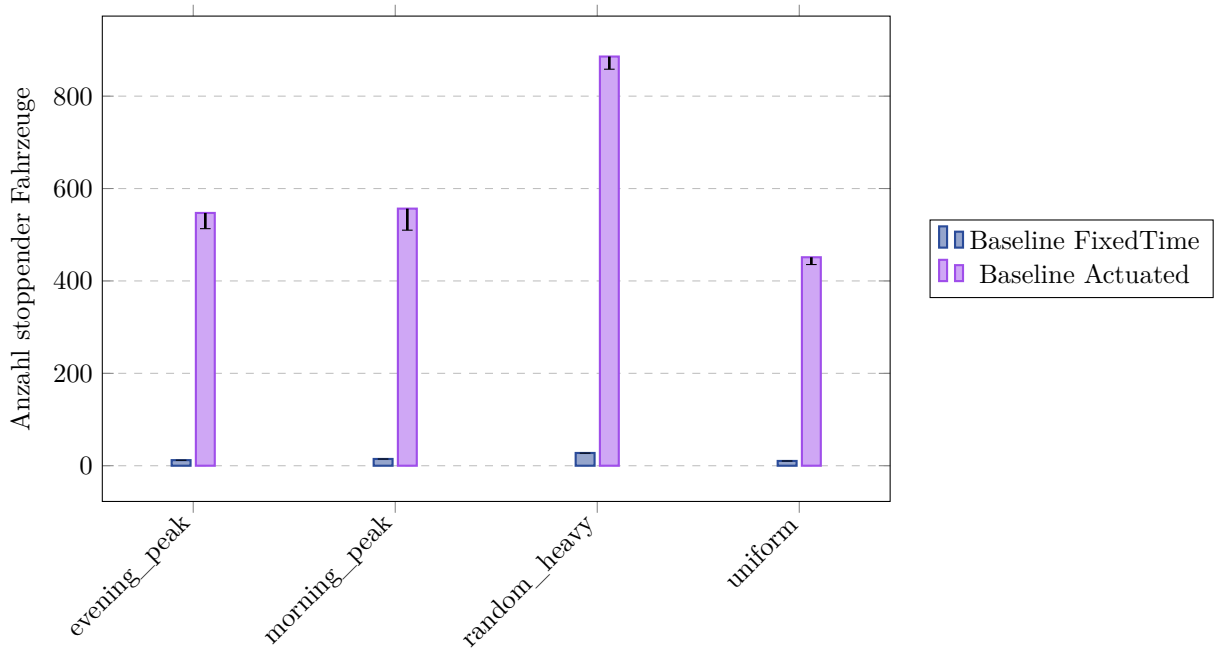


Abbildung 21: Anzahl stoppender Fahrzeuge

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

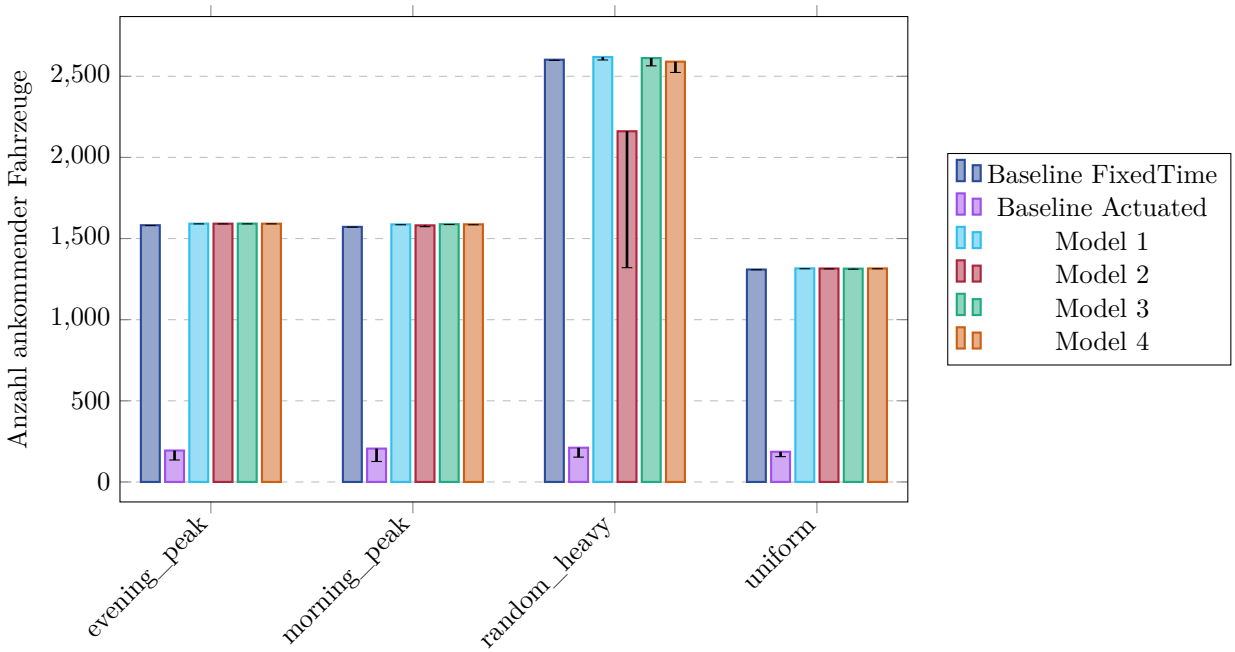


Abbildung 22: Anzahl ankommender Fahrzeuge

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

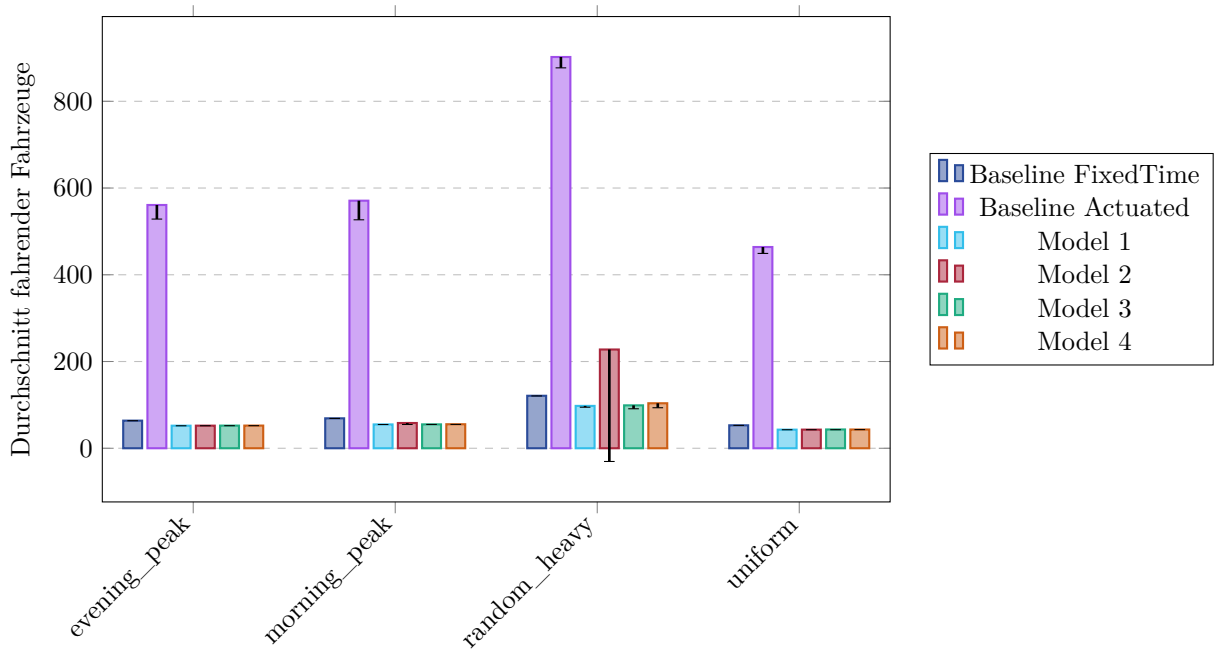


Abbildung 23: Durchschnitt fahrender Fahrzeuge

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

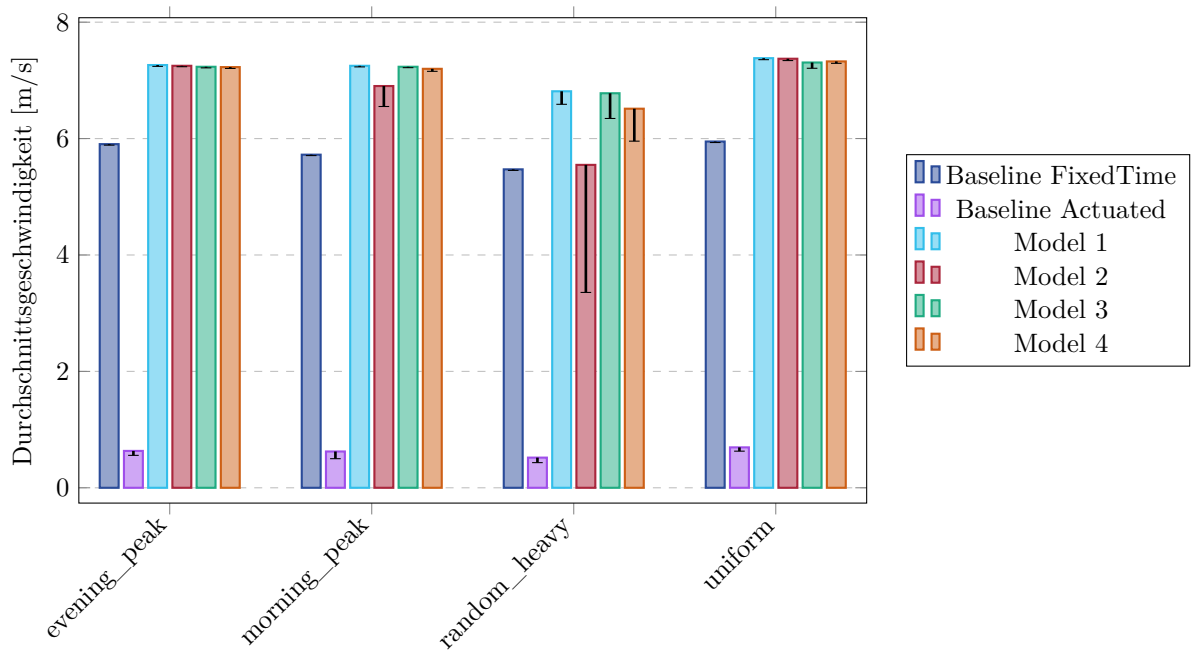


Abbildung 24: Durchschnittsgeschwindigkeiten

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

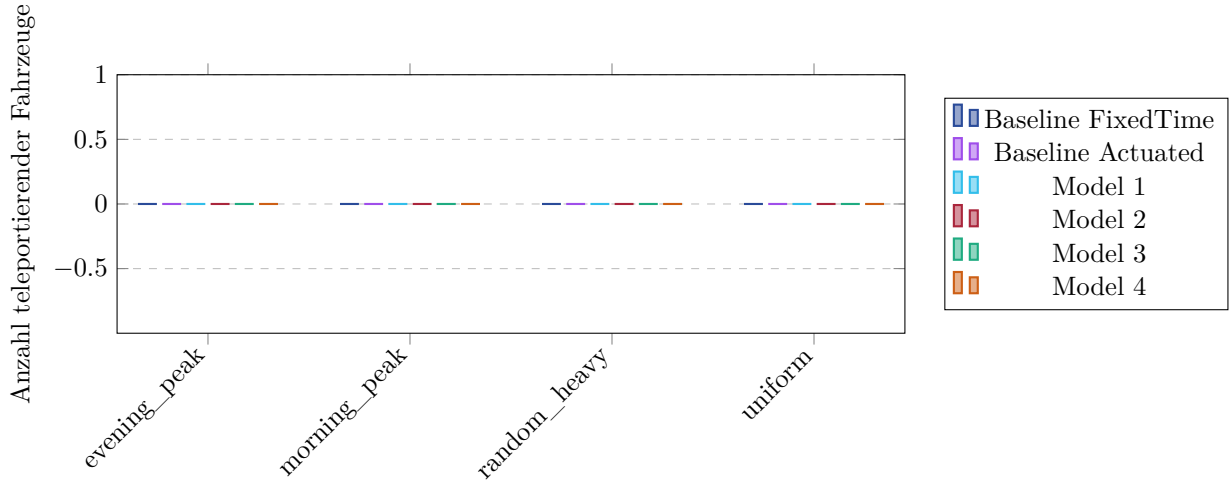


Abbildung 25: Anzahl teleportierender Fahrzeuge

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

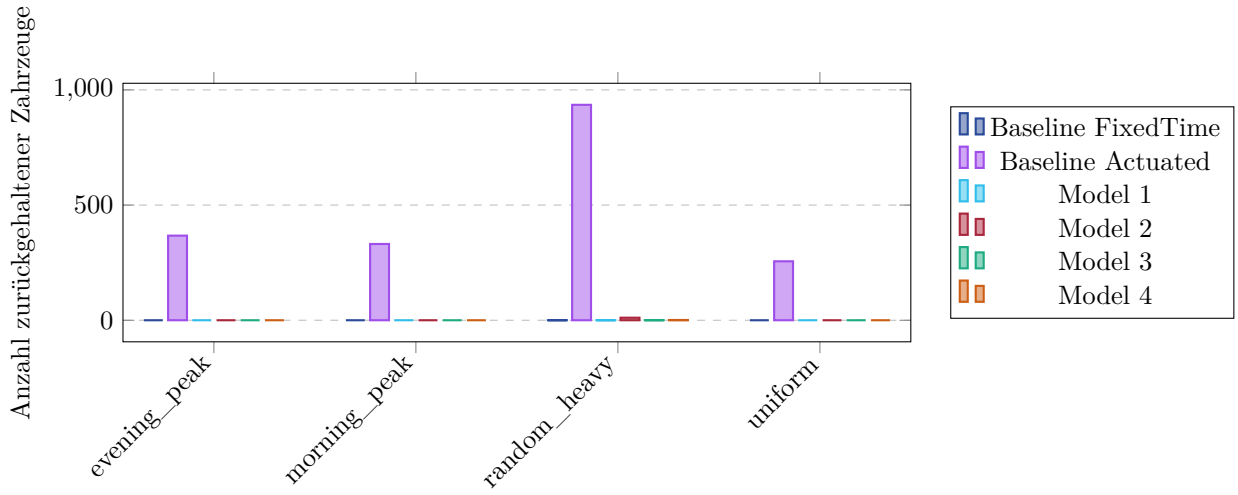


Abbildung 26: Anzahl zurückgehaltener Zahrzeuge

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 Rückstaueffekte 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 ziel auf ein kombiniertes Minimieren der Wartezeiten, Anzahl an Phasenwechsel und Staus.

5.3.1 Mittlere Wartezeiten

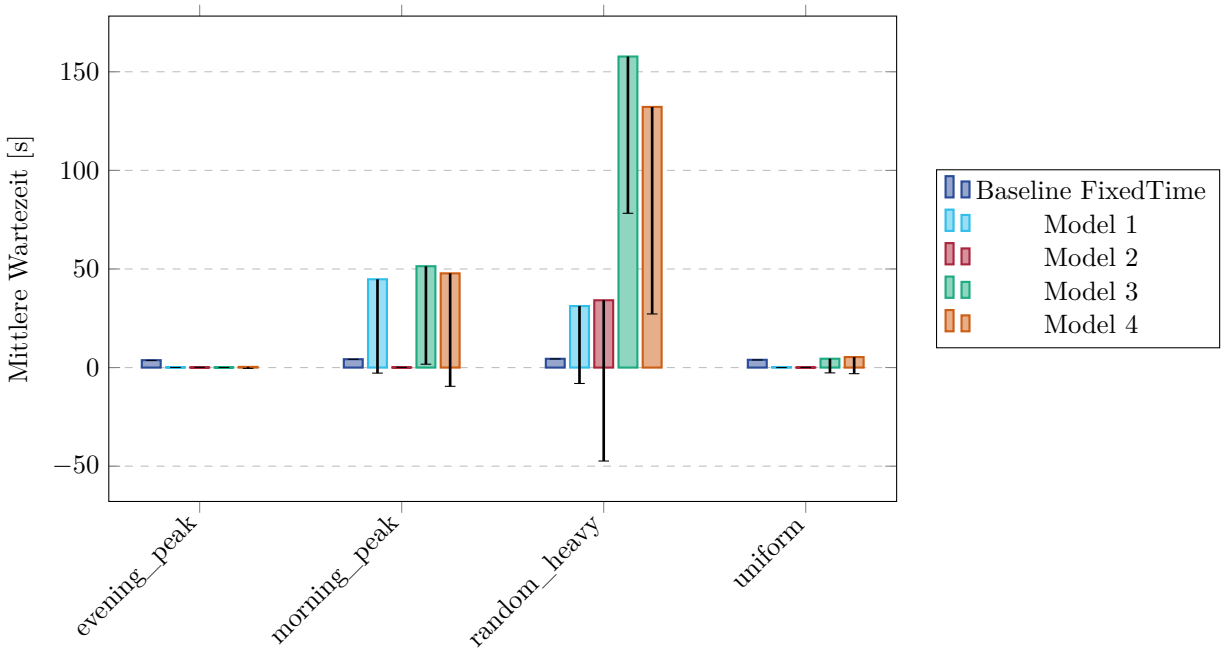


Abbildung 27: Mittlere Wartezeiten

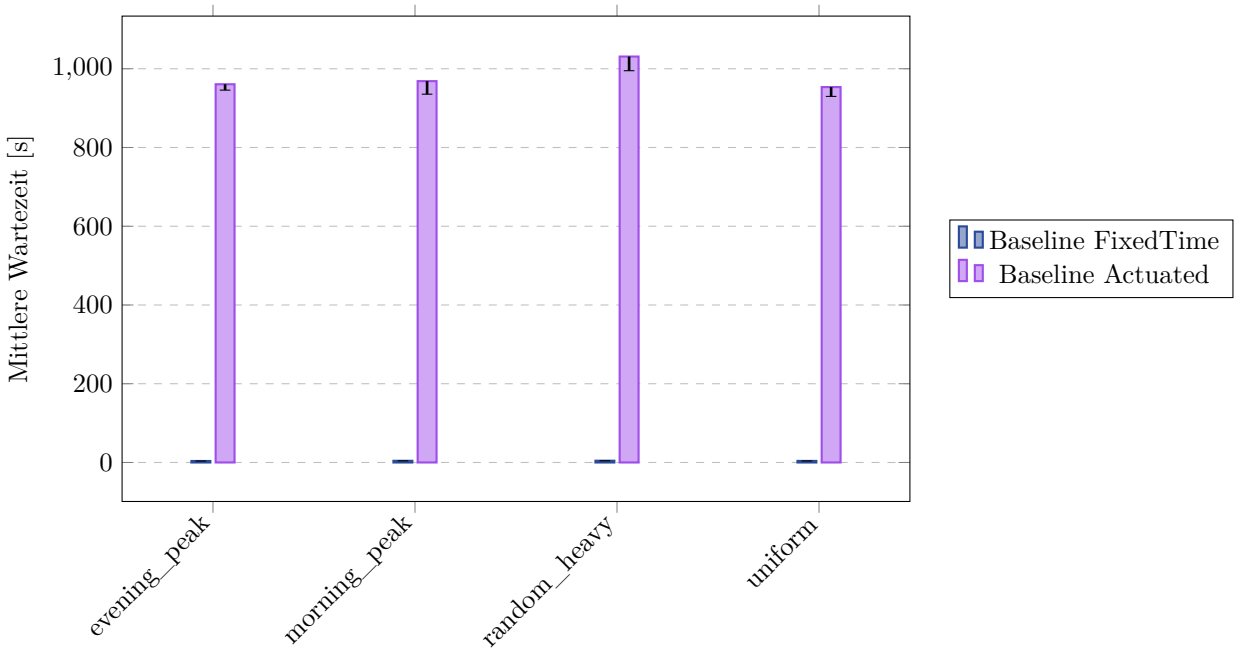


Abbildung 28: Mittlere Wartezeiten

Die Ergebnisse zur mittleren Wartezeit sind in Abbildung X dargestellt. Die FixedTime-Baseline erreicht im Szenario morning_peak einen Mittelwert von 3.74 s sowie im Szenario evening_peak 4.24 s. In den Szenarien random_heavy und uniform liegen die Werte bei 4.44 s bzw. 3.93 s, womit die Methode insgesamt stabile Resultate erzielt.

Die Actuated-Baseline weist demgegenüber deutlich höhere Wartezeiten auf. So werden im morning_peak durchschnittlich 960 s und im evening_peak 968 s gemessen. Besonders ausgeprägt sind die Werte in den Szenarien random_heavy (1031 s) und uniform (953 s), was auf erhebliche Ineffizienz dieser Steuerungsstrategie hindeutet.

Unter den trainierten Modellen zeigt Modell 1 in morning_peak mit 0.17 s sowie in uniform mit 0.25 s die niedrigsten Wartezeiten aller Verfahren. Im evening_peak werden 44.78 s und im random_heavy 31.19 s erreicht. Modell 2 erzielt Werte von 0.10 s (morning_peak), 0.10 s (evening_peak), 34.13 s (random_heavy) und 0.08 s (uniform). Bei Modell 3 zeigen sich mit 0.12 s (morning_peak), 51.40 s (evening_peak), 157.72 s (random_heavy) und 4.48 s (uniform) deutlich größere Unterschiede zwischen den Szenarien. Modell 4 verhält sich ähnlich und erreicht 0.34 s (morning_peak), 47.77 s (evening_peak), 132.16 s (random_heavy) sowie 5.32 s (uniform).

Auffällig ist, dass insbesondere bei Modell 3 und Modell 4 die deutlich erhöhten Mittelwerte in den Szenarien random_heavy und evening_peak jeweils mit einer hohen Standardabweichung einhergehen. Dies weist auf eine ausgeprägte Varianz zwischen den einzelnen Evaluationsläufen hin und deutet darauf, dass die Modelle in manchen Episoden sehr niedrige, in anderen jedoch stark erhöhte Wartezeiten produzierten.

5.3.2 Anzahl stoppender Fahrzeuge

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

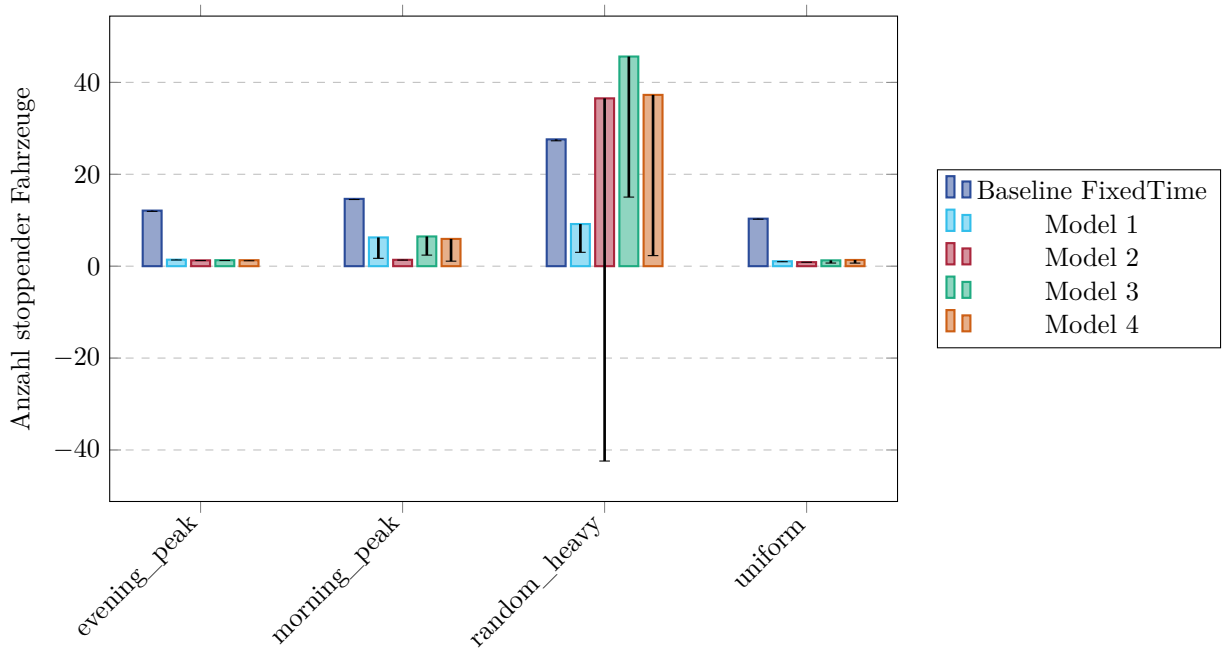


Abbildung 29: Anzahl stoppender Fahrzeuge

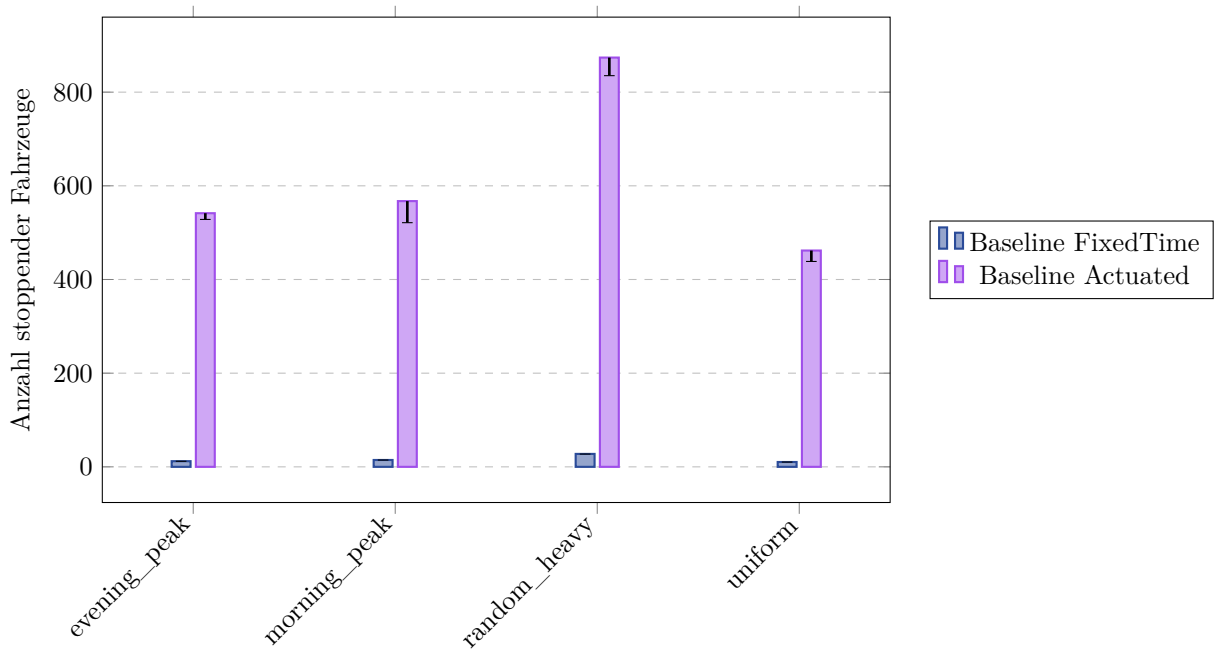


Abbildung 30: Anzahl stoppender Fahrzeuge

Die mittlere Anzahl stoppender Fahrzeuge unterscheidet sich deutlich zwischen den Baselines und den trainierten Modellen. Die Fixed-Time-Baseline erreicht im Szenario morning_peak durchschnittlich 12.09 Fahrzeuge, im evening_peak 14.64 Fahrzeuge,

im random_heavy 27.60 Fahrzeuge und im uniform-Szenario 10.33 Fahrzeuge.

Die Actuated-Baseline weist deutlich höhere Werte auf. Im morning_peak ergeben sich im Mittel 541.53 Fahrzeuge, im evening_peak 567 Fahrzeuge, im random_heavy 873 Fahrzeuge und im uniform 461 Fahrzeuge. Diese Resultate bestätigen das schwache Abschneiden der Actuated-Baseline auch in dieser Metrik.

Die trainierten Modelle zeigen demgegenüber eine deutliche Reduktion der Stopps. Modell 1 erreicht Werte von 1.40 (morning_peak), 6.25 (evening_peak), 9.17 (random_heavy) und 1.07 (uniform). Modell 2 erzielt 1.26, 1.38, 36.52 und 0.87 Fahrzeuge in den entsprechenden Szenarien. Bei Modell 3 liegen die Mittelwerte bei 1.30 (morning_peak), 6.46 (evening_peak), 45.62 (random_heavy) und 1.26 (uniform). Modell 4 erreicht 1.29, 5.93, 37.27 und 1.34 Fahrzeuge.

Es zeigt sich, dass insbesondere die Modelle 2-4 im Szenario random_heavy deutlich erhöhte Werte im Vergleich zu den übrigen Szenarien aufweisen. In diesen Fällen geht der Anstieg jeweils mit einer sehr hohen Standardabweichung einher, was auf starke Schwankungen zwischen den einzelnen Evaluationsläufen hinweist. Dies legt nahe, dass die Modelle teilweise sehr effiziente Steuerungsstrategien erlernen konnten, während in anderen Episoden suboptimale Strategien dominierten.

5.3.3 Anzahl ankommender Fahrzeuge

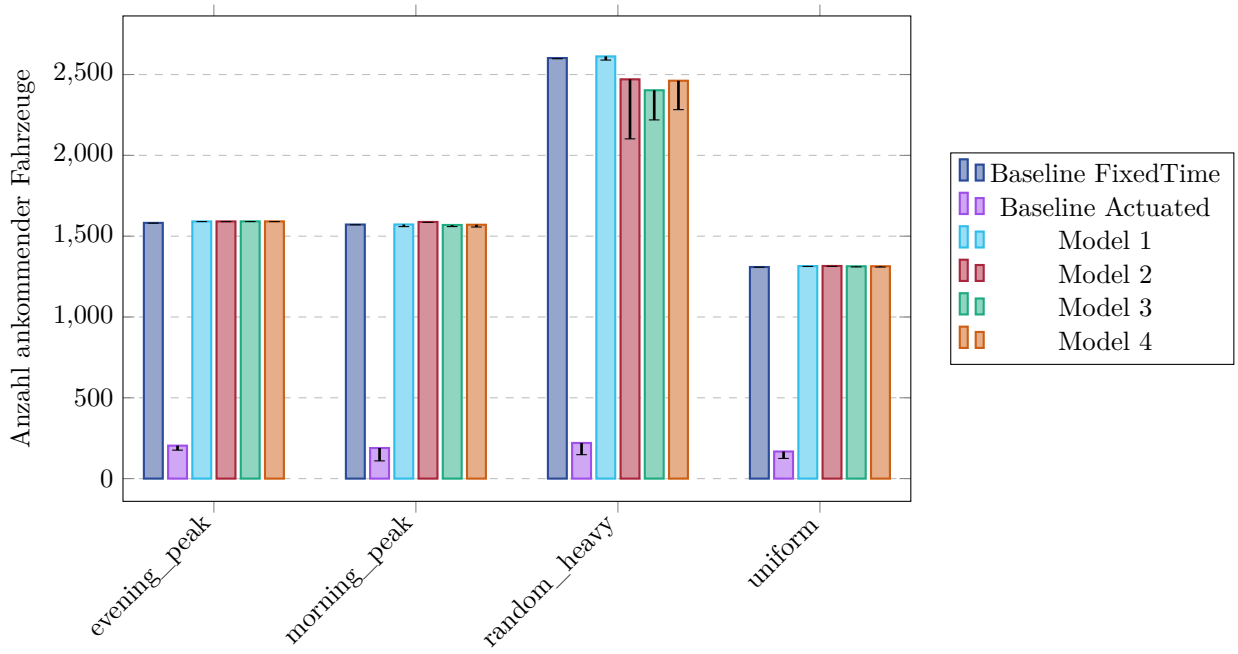


Abbildung 31: Anzahl ankommender Fahrzeuge

Die Auswertung der Anzahl ankommender Fahrzeuge zeigt über alle Szenarien hinweg sehr ähnliche Ergebnisse für die Fixed-Time-Baseline und die trainierten Modelle. Sowohl die Baseline als auch die Modelle erreichen in den meisten Szenarien nahezu das Maximum, was darauf hindeutet, dass der Verkehrsfluss grundsätzlich zuverlässig abgewickelt wird.

Auffällig ist lediglich, dass in random_heavy einzelne Modelle eine leicht geringere Performance aufweisen als die Fixed-Time-Baseline. Dieser Rückgang bleibt jedoch

moderat, und die Anzahl ankommender Fahrzeuge liegt weiterhin auf einem hohen Niveau. In den übrigen Szenarien (morning_peak, evening_peak, uniform) stimmen die Resultate nahezu exakt mit der Fixed-Time-Baseline überein.

Die Actuated-Baseline bestätigt erneut ihre Schwäche und fällt in allen Szenarien deutlich ab. Das deutliche Defizit dieser Steuerungsstrategie kontrastiert stark mit den stabil hohen Werten der Fixed-Time-Baseline und der Modelle.

5.3.4 Durchschnitt fahrender Fahrzeuge

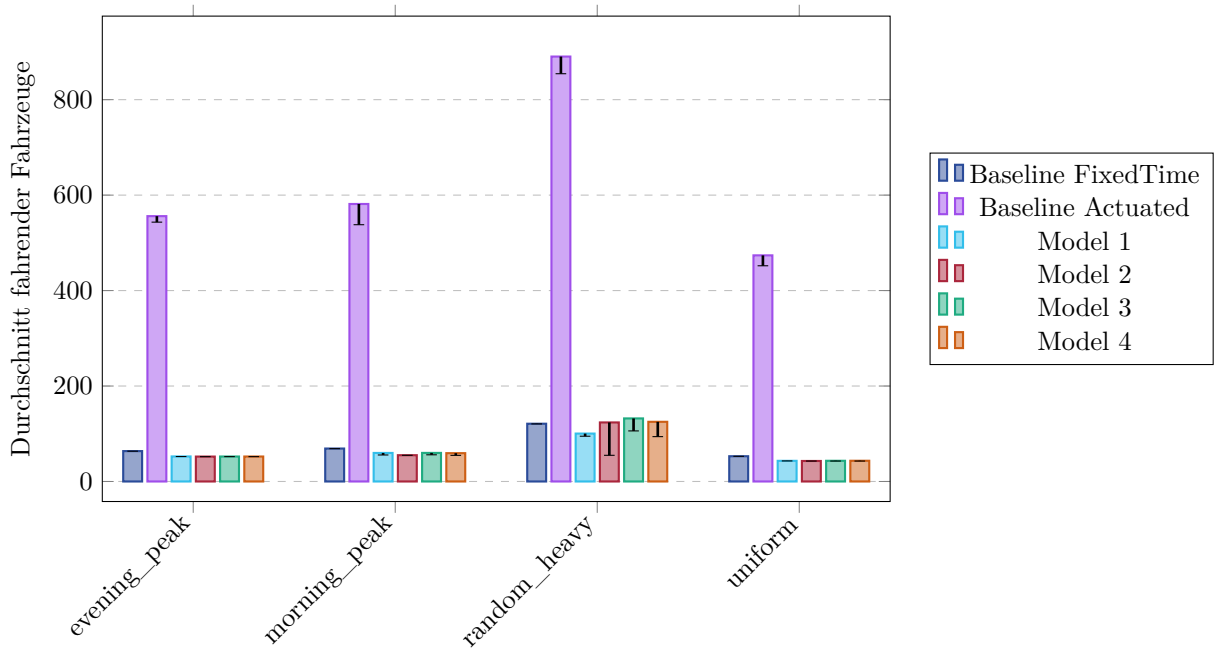


Abbildung 32: Durchschnitt fahrender Fahrzeuge

Die Ergebnisse zum Durchschnitt der sich im Netz befindlichen Fahrzeuge zeigen deutliche Unterschiede zwischen den Verfahren. Ein geringerer Wert ist hierbei positiv zu bewerten, da er auf eine schnellere Abwicklung des Verkehrs hinweist.

Die Fixed-Time-Baseline erreicht im morning_peak durchschnittlich 63.64 Fahrzeuge, im evening_peak 69 Fahrzeuge, im random_heavy 121 Fahrzeuge und im uniform-Szenario 52 Fahrzeuge. Damit ergibt sich ein insgesamt konsistentes Niveau mit moderaten Unterschieden zwischen den Szenarien.

Die Actuated-Baseline fällt erneut deutlich ab und erreicht im morning_peak 555 Fahrzeuge, im evening_peak 581 Fahrzeuge, im random_heavy 890 Fahrzeuge und im uniform 473 Fahrzeuge. Diese Werte liegen um ein Vielfaches über denen der Fixed-Time-Baseline und verdeutlichen die unzureichende Leistungsfähigkeit der Methode.

Die trainierten Modelle zeigen durchweg Werte, die sehr nah an der Fixed-Time-Baseline liegen. Modell 1 erreicht 52, 59, 100, 43 Fahrzeuge, Modell 2 weist 52, 55, 123, 42 Fahrzeuge auf, während Modell 3 mit 52, 59, 132, 43 Fahrzeuge und Modell 4 mit 51, 59, 125, 43 Fahrzeuge vergleichbare Resultate erzielen.

Auffällig ist, dass insbesondere im Szenario random_heavy einzelne Modelle (Modell 2-4) leicht erhöhte Werte im Vergleich zur Fixed-Time-Baseline aufweisen. Diese

Abweichungen gehen zugleich mit einer erhöhten Standardabweichung einher, was auf instabilere Performanz in diesem Szenario schließen lässt. In den übrigen Szenarien zeigen alle Modelle nahezu identische Ergebnisse zur Fixed-Time-Baseline.

5.3.5 Durchschnittsgeschwindigkeiten

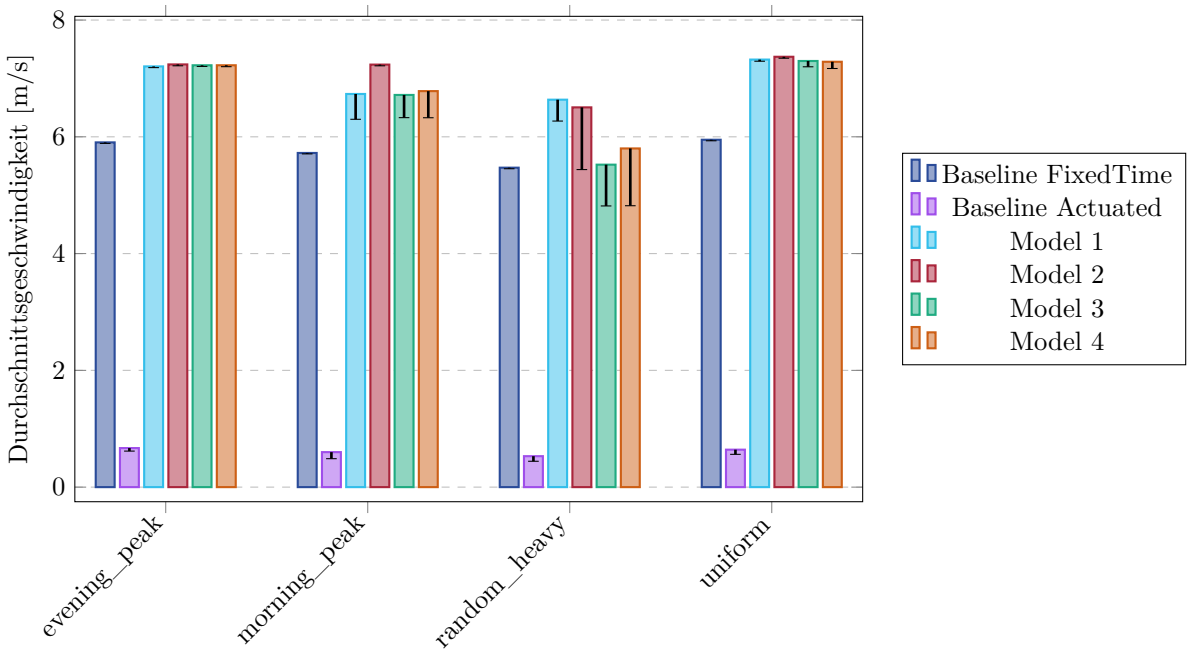


Abbildung 33: Durchschnittsgeschwindigkeiten

Die Analyse der Durchschnittsgeschwindigkeiten verdeutlicht klare Unterschiede zwischen den Baselines und den trainierten Modellen. Höhere Werte sind hierbei positiv zu bewerten, da sie auf einen effizienteren Verkehrsfluss hinweisen.

Die Fixed-Time-Baseline erreicht im morning_peak durchschnittlich 5.9 m/s, im evening_peak 5.7 m/s, im random_heavy 5.4 m/s sowie im uniform-Szenario 5.9 m/s. Damit liefert sie insgesamt solide Ergebnisse, die in allen Szenarien oberhalb der Actuated-Baseline liegen.

Die Actuated-Baseline weist durchgehend die niedrigsten Geschwindigkeiten auf: 0.6 m/s (morning_peak), 0.5 m/s (evening_peak), 0.52 m/s (random_heavy) und 0.6 m/s (uniform). Diese Werte sind um eine Größenordnung geringer als die der Fixed-Time-Baseline und bestätigen erneut die unzureichende Leistungsfähigkeit dieses Ansatzes.

Die trainierten Modelle erzielen durchweg höhere Durchschnittsgeschwindigkeiten als die Fixed-Time-Baseline. Modell 1 erreicht 7.2, 6.7, 6.6 und 7.3 m/s in den vier Szenarien. Modell 2 weist mit 7.2, 7.2, 6.5 und 7.3 m/s ein sehr stabiles Niveau auf. Modell 3 zeigt Werte von 7.2, 6.7, 5.5 und 7.2 m/s, wobei insbesondere im random_heavy-Szenario ein deutlicher Rückgang erkennbar ist. Modell 4 erzielt vergleichbare Resultate mit 7.2, 6.7, 5.8 und 7.2 m/s.

Besonders im Szenario random_heavy fällt auf, dass Modell 3 und Modell 4 geringere Durchschnittsgeschwindigkeiten erreichen als die übrigen Modelle. Auch hier zeigt sich eine erhöhte Standardabweichung, die auf instabile Ergebnisse zwischen

den Evaluationsläufen hindeutet. In den übrigen Szenarien liegen alle Modelle konsistent über der Fixed-Time-Baseline und markieren eine deutliche Verbesserung des Verkehrsflusses.

5.3.6 Anzahl teleportierender Fahrzeuge

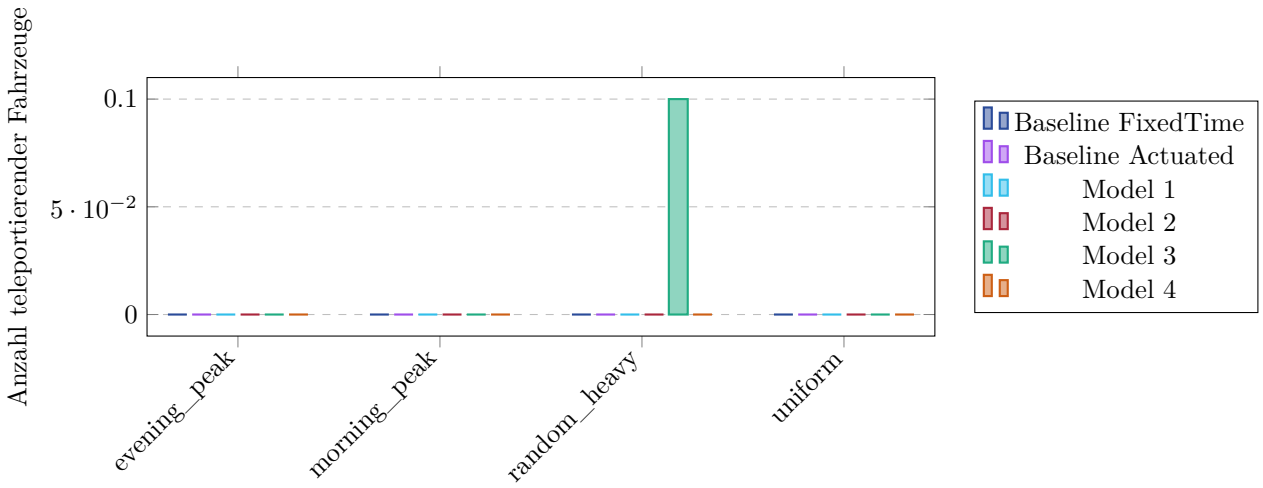


Abbildung 34: Anzahl teleportierender Fahrzeuge

Die Auswertung der Teleportationen zeigt, dass in nahezu allen Szenarien keine Fahrzeuge teleportiert werden mussten. Dies gilt sowohl für die beiden Baselines als auch für die trainierten Modelle. Eine Ausnahme bildet das Szenario random_heavy bei Modell 3, in dem vereinzelt Teleportationen auftraten. Dieses Ergebnis steht im Einklang mit den zuvor beobachteten Schwächen desselben Modells in diesem Szenario.

5.3.7 Anzahl zurückgehaltener Fahrzeuge

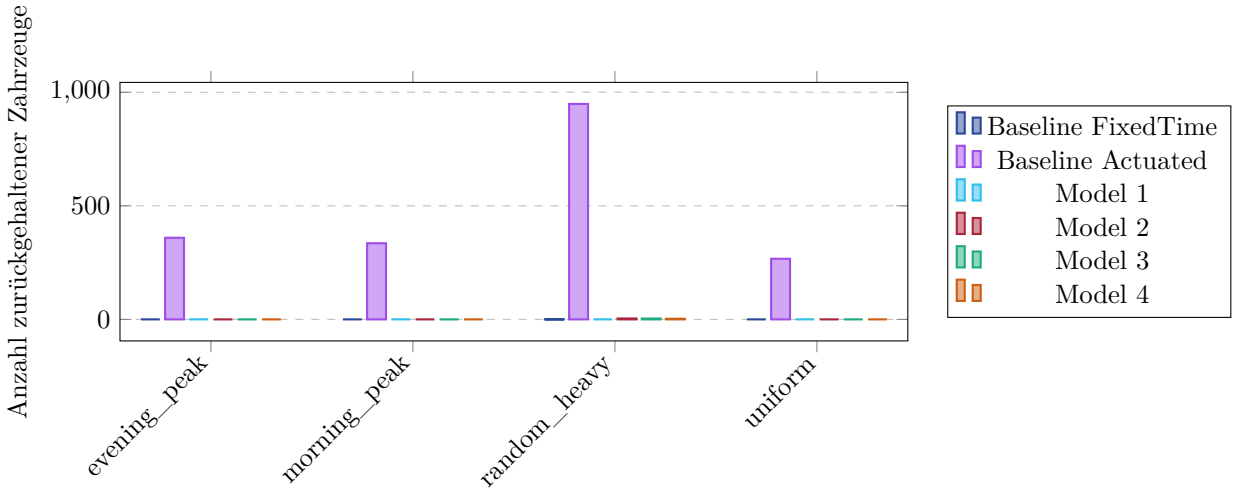


Abbildung 35: Anzahl zurückgehaltener Zahrzeuge

n allen Szenarien zeigen die vier Modelle sowie die Fixed-Time-Baseline keine zurückgehaltenen Fahrzeuge. Lediglich die Actuated-Baseline weist in sämtlichen Szenarien deutliche Werte auf, die mit der insgesamt schwachen Performance dieser Methode konsistent sind. Dieses Ergebnis bestätigt, dass ausschließlich die Actuated-Steuerung Fahrzeuge im Netz blockiert, während alle anderen Verfahren einen stabilen Verkehrsfluss ohne Zurückhalten sicherstellen konnten.

5.3.8 Einstufung

Die Auswertung der verschiedenen Metriken zeigt, dass die trainierten Modelle die klassischen Baselines insgesamt deutlich übertreffen. Während die Actuated-Baseline durchgehend schwache Ergebnisse liefert und selbst von der Fixed-Time-Steuerung klar geschlagen wird, gelingt es den Modellen in nahezu allen Szenarien, sowohl die mittlere Wartezeit als auch die Anzahl stoppender Fahrzeuge deutlich zu reduzieren und gleichzeitig höhere Durchschnittsgeschwindigkeiten zu erreichen.

Besonders deutlich wird der Vorteil der Modelle in den Szenarien mit regulärer oder gleichmäßiger Verkehrslast, wo sie konsistent nahe am Optimum operieren. Auch die Anzahl ankommender Fahrzeuge bleibt in diesen Fällen auf dem maximalen Niveau, sodass die Effizienzsteigerung nicht mit einem Verlust an Durchsatz erkauft wird.

Einschränkungen zeigen sich jedoch im random_heavy-Szenario: hier treten bei mehreren Modellen signifikante Verschlechterungen auf, die zugleich mit einer hohen Standardabweichung verbunden sind. Dies weist auf eine eingeschränkte Robustheit unter komplexeren und schwer vorhersagbaren Verkehrssituationen hin. Besonders ausgeprägt sind diese Schwächen bei einem Modell, das zusätzlich vereinzelt Teleportationen aufweist und damit strukturelle Instabilitäten erkennen lässt.

Insgesamt lässt sich festhalten, dass die lernbasierten Steuerungsansätze das Potenzial besitzen, klassische Verfahren im Hinblick auf Wartezeiten, Staus und Geschwindigkeiten deutlich zu übertreffen. Gleichzeitig verdeutlichen die Ergebnisse, dass die Generalisierungsfähigkeit insbesondere in Szenarien mit unregelmäßiger und schwer prognostizierbarer Verkehrslast eine zentrale Herausforderung bleibt.

5.4 Reward: CO_2-Emissionen

Die vierte Gruppe zielt auf eine Minimierung der CO_2-Emissionen, während gleichzeitig versucht wird die Stabilität aufrecht zu erhalten. Für diese Emissions-basierten Modelle wird zusätzlich die CO_2-Emission als zentrale Metrik betrachtet.

5.4.1 CO_2-Emissionen

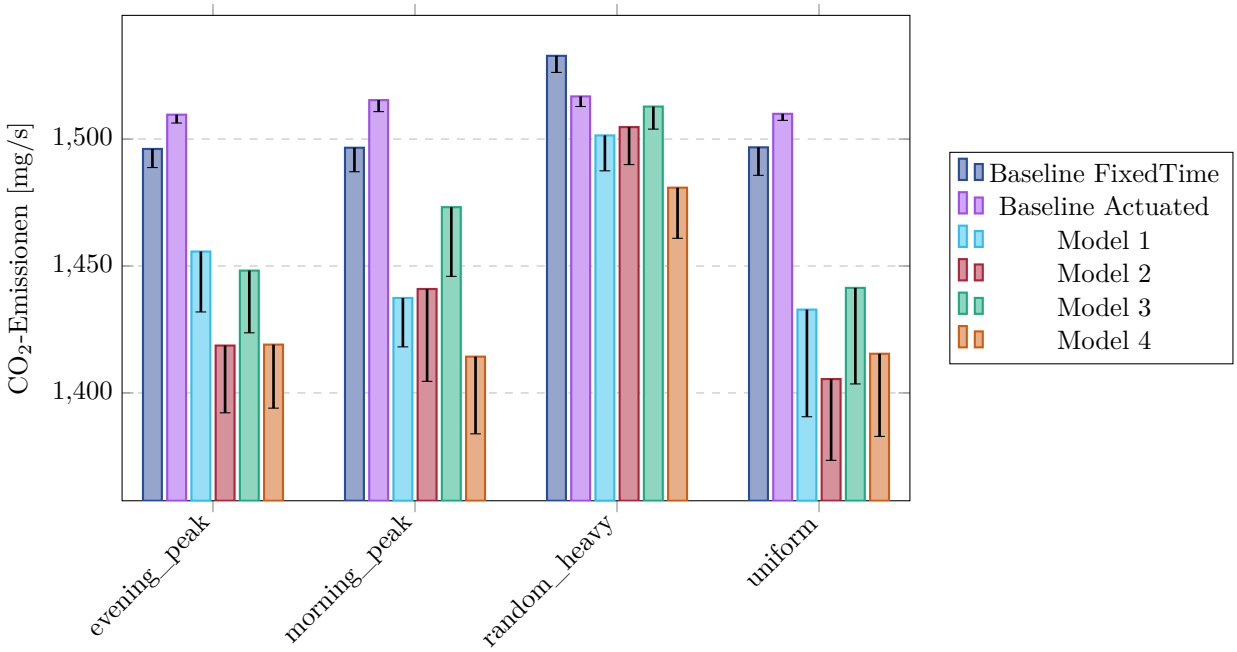


Abbildung 36: CO_2-Emissionen

Die Analyse der CO_2-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

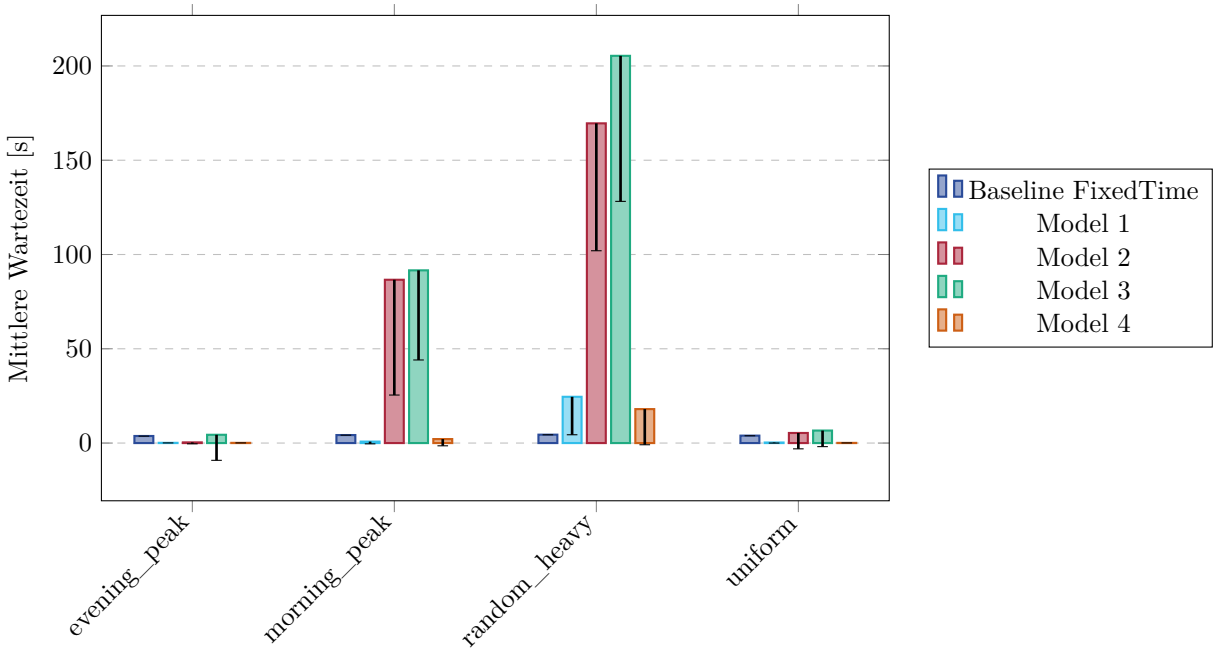


Abbildung 37: Mittlere Wartezeiten

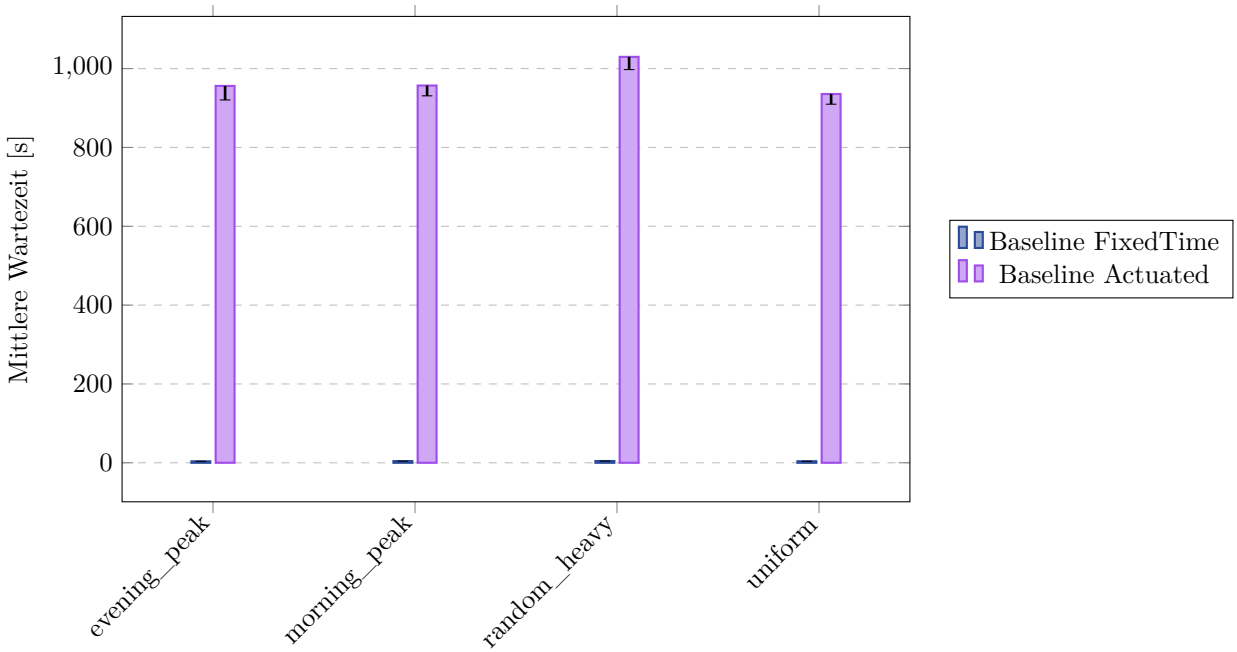


Abbildung 38: Mittlere Wartezeiten

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.

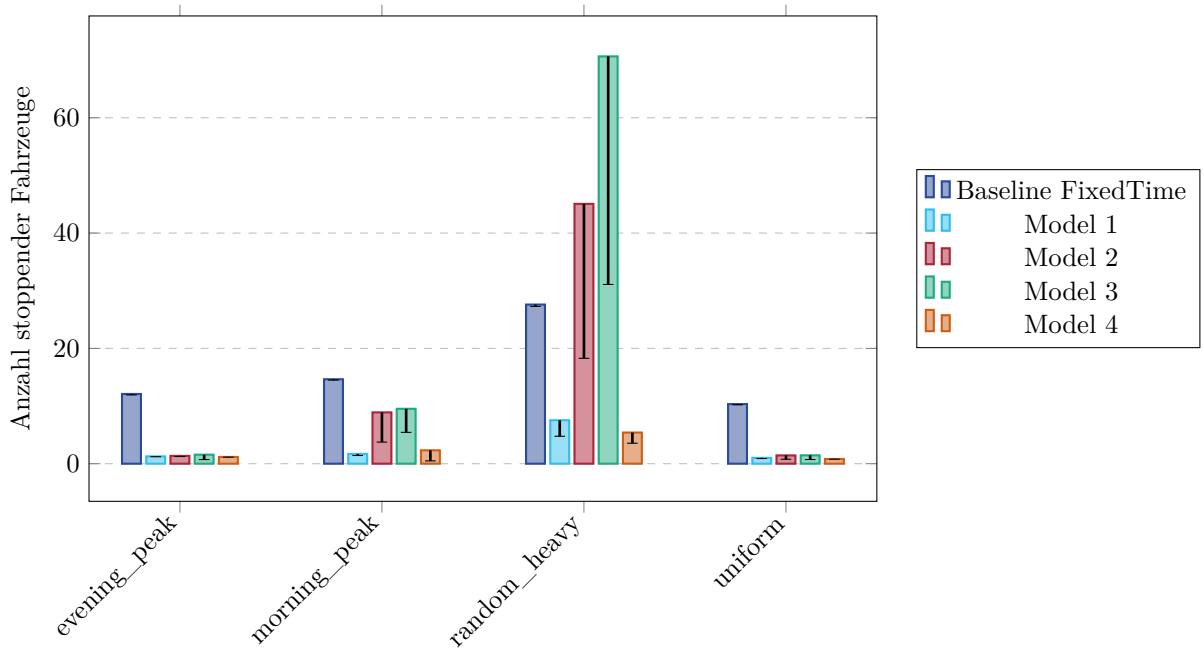


Abbildung 39: Anzahl stoppender Fahrzeuge

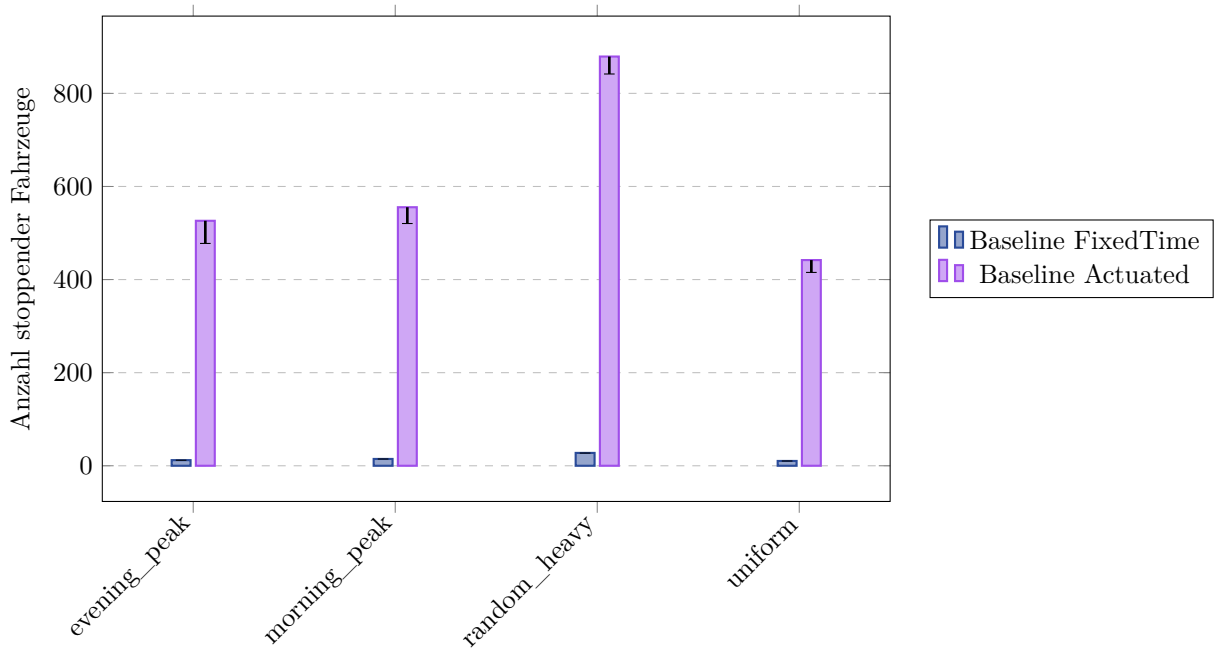


Abbildung 40: Anzahl stoppender Fahrzeuge

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

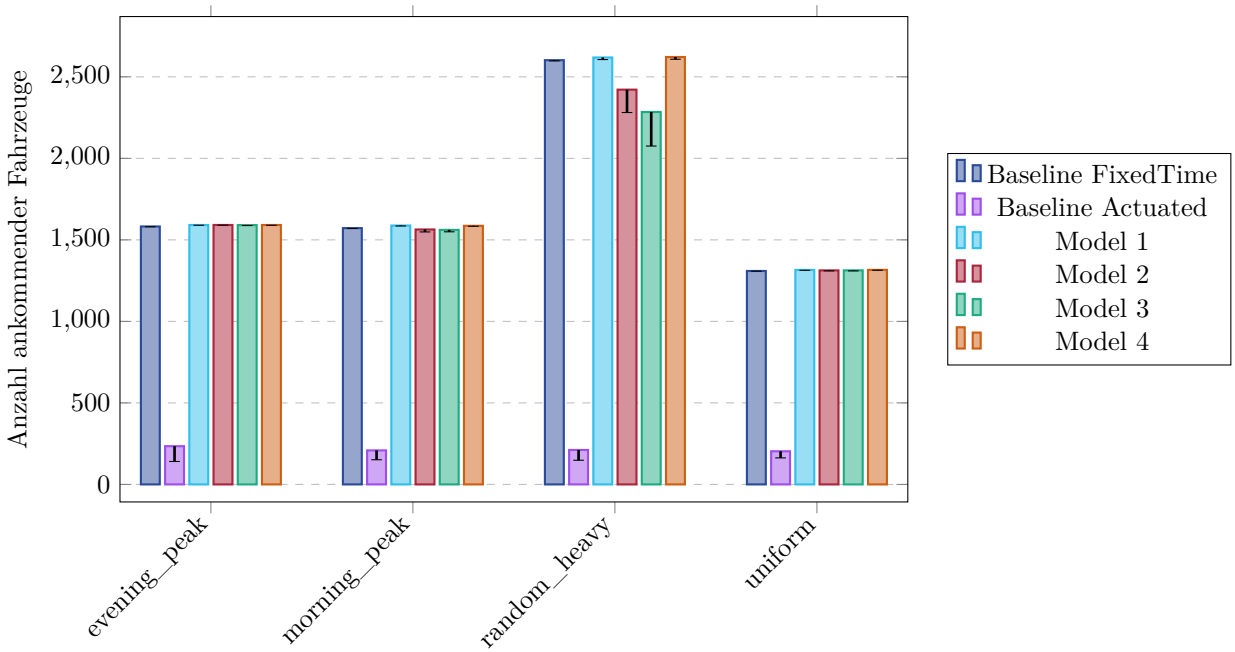


Abbildung 41: Anzahl ankommender Fahrzeuge

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

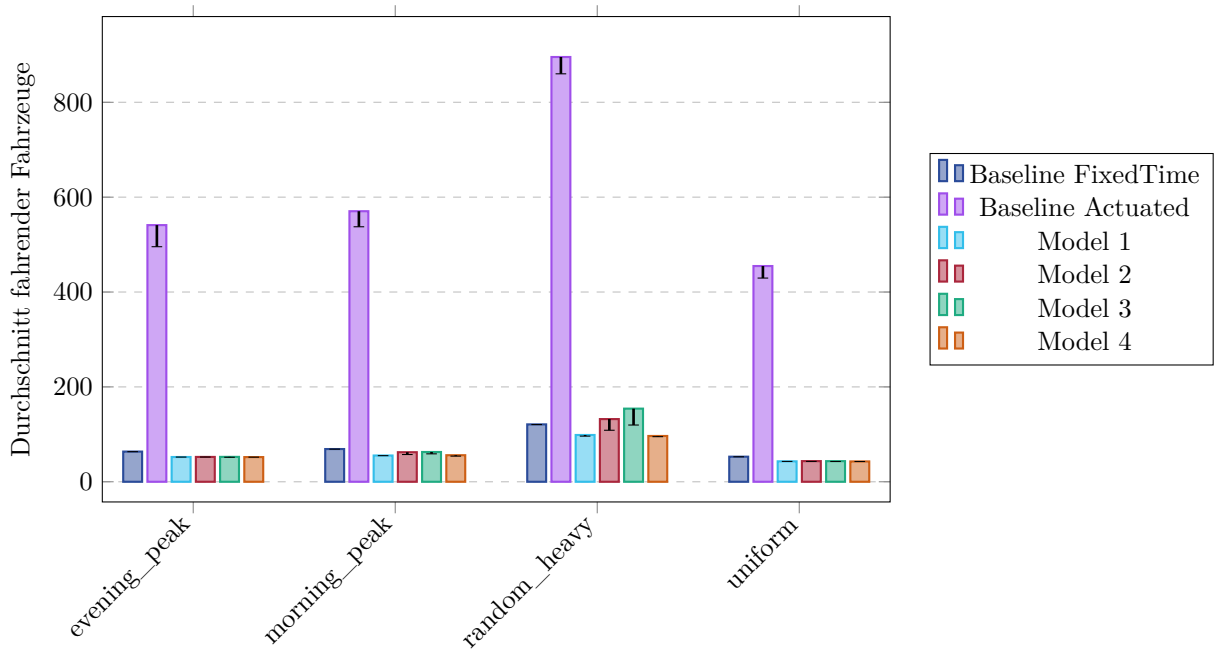


Abbildung 42: Durchschnitt fahrender Fahrzeuge

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

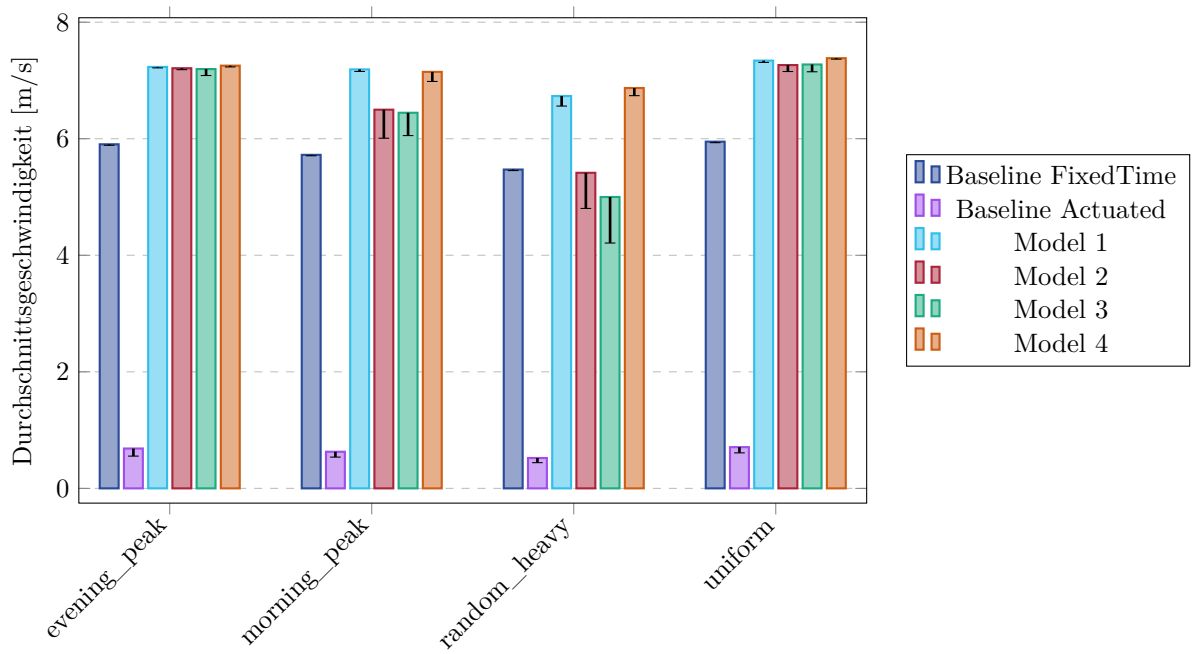


Abbildung 43: Durchschnittsgeschwindigkeiten

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

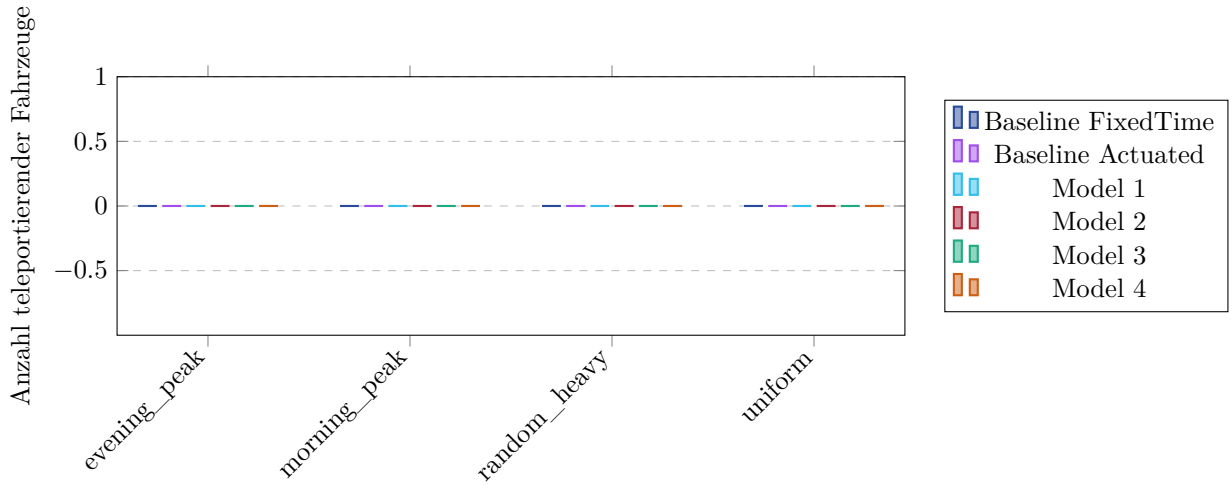


Abbildung 44: Anzahl teleportierender Fahrzeuge

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

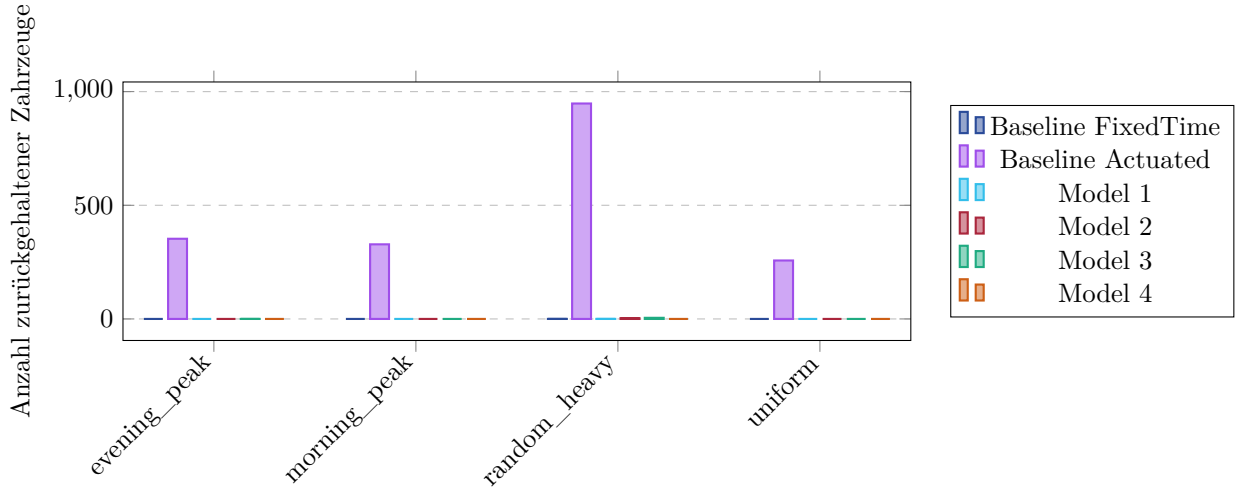


Abbildung 45: Anzahl zurückgehaltener Zahrzeuge

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

staueffekten. 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 CO_2-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 Rückstaueffekte 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

Ein zentrales Ziel der Evaluation bestand darin, nicht nur die absolute Leistungsfähigkeit der Modelle zu messen, sondern auch deren **Robustheit** und **Replikationsfähigkeit** zu bewerten. Unter Robustheit wird hier die Fähigkeit verstanden, auch unter variierenden Verkehrsbedingungen (verschiedene Szenarien) und unterschiedlichen Initialisierungen (Seeds) konsistente Resultate zu erzielen. Replikationsfähigkeit bezeichnet hingegen die Eigenschaft, dass ein Modell mit gleicher Konfiguration über mehrere Trainingsläufe hinweg vergleichbare Ergebnisse liefert.

Methodisches Vorgehen Für die Robustheitsanalyse wurden alle Modelle sowie die beiden Baselines in den vier Szenarien (morning_peak, evening_peak, uniform, random_heavy) jeweils über zehn Episoden evaluiert. Unterschiedliche Seeds stellten dabei sicher, dass sowohl Zufallseinflüsse im Verkehrsfluss als auch in der Modellinitialisierung berücksichtigt wurden. Pro Reward-Funktion lagen vier unabhängige Trainingsseeds vor, sodass neben der Szenario-Robustheit auch die Replikationsfähigkeit geprüft werden konnte.

Die Evaluationsumgebung protokollierte sowohl *mittlere Metriken* (z. B. durchschnittliche Wartezeit, mittlere Geschwindigkeit) als auch *Totals* (z. B. Anzahl ankommender Fahrzeuge, Gesamtemissionen) pro Episode. Diese Auswertung erlaubte es, systematische Leistungsunterschiede von stochastischen Schwankungen zu trennen. Besonders aufschlussreich war dabei die Betrachtung der Standardabweichungen: hohe

Varianz bei gleicher Reward-Funktion weist auf eine eingeschränkte Replikationsfähigkeit hin.

Ergebnisse Über alle Reward-Funktionen hinweg zeigte sich ein konsistentes Muster:

- **Baselines:** Die Actuated-Baseline erwies sich in allen Szenarien als instabil und deutlich unterlegen. Die Fixed-Time-Baseline blieb robust, konnte jedoch nur in Szenarien mit regulärer Last überzeugen.
- **Diff-Waiting-Time:** Modelle auf Basis dieser Reward-Funktion waren insgesamt robust, insbesondere Modell 1 und Modell 4. Modelle 2 und 3 zeigten dagegen in random_heavy erhöhte Varianz und teilweise deutliche Leistungseinbrüche, was die eingeschränkte Replikationsfähigkeit einzelner Seeds verdeutlicht.
- **Queue:** Die Queue-basierten Modelle erzielten eine durchgängig hohe Stabilität und reduzierten Wartezeiten und Stopps stark. Lediglich Modell 2 zeigte in random_heavy und morning_peak eine geringere Robustheit.
- **Real-World:** Diese Reward-Funktion lieferte in allen Szenarien außer random_heavy sehr stabile und nahe am Optimum liegende Ergebnisse. Unter hoher und unregelmäßiger Last traten jedoch erhöhte Standardabweichungen auf, was auf eingeschränkte Robustheit hindeutet.
- **CO_2-Emissionen:** Die Emissions-basierten Modelle kombinierten hohe Effizienz mit ökologischer Optimierung und erwiesen sich als besonders stabil. Lediglich in Hochlastszenarien (insb. Modell 2 und 3) zeigten sich Replikationsprobleme, die sich jedoch auf ein Niveau im Bereich der Fixed-Time-Baseline einpendelten.

Schlussfolgerung Die Analyse zeigt, dass die Robustheit und Replikationsfähigkeit stark von der gewählten Reward-Funktion abhängen. Während Queue- und CO_2-basierte Ansätze eine besonders stabile Performanz liefern, neigen Diff-Waiting-Time- und Real-World-Rewards unter Hochlastbedingungen zu Instabilitäten. In allen Fällen bleiben die Modelle den klassischen Baselines jedoch überlegen. Damit bestätigt sich, dass Deep-RL-basierte Verkehrssteuerungen grundsätzlich robuste Strategien hervorbringen, deren Generalisierungsfähigkeit sich jedoch insbesondere in unregelmäßigen Verkehrssituationen weiter verbessern muss.

5.6 Gesamtevaluierung und Schlussfolgerung

Die durchgeführte Evaluation verdeutlicht, dass alle untersuchten Reinforcement-Learning-Ansätze die klassischen Baselines (Fixed-Time und Actuated) in nahezu allen Metriken deutlich übertreffen. Insbesondere die Actuated-Baseline bestätigt sich über sämtliche Szenarien hinweg als instabil und ineffizient. Die Fixed-Time-Baseline hingegen zeigt, wie bereits aus der realen Welt zu erwarten, sehr gute und robuste Resultate, da diese Steuerungsstrategie seit Jahrzehnten in vielen Städten standardmäßig eingesetzt wird. Sie stellt damit eine starke Referenz dar, die von den RL-Modellen erst übertroffen werden musste.

Über alle Rewardfunktionen hinweg konnten die Modelle signifikante Verbesserungen in Bezug auf *mittlere Wartezeit*, *Anzahl stoppender Fahrzeuge* und *Durchschnittsgeschwindigkeit* erzielen. Ebenso wurde in fast allen Szenarien die maximale Anzahl ankommender Fahrzeuge erreicht, sodass die Effizienzsteigerungen nicht mit einem reduzierten Gesamtdurchsatz einhergingen.

Gleichzeitig zeigen die Ergebnisse, dass die Wahl der Rewardfunktion maßgeblichen Einfluss auf die Robustheit und Stabilität der Modelle hat:

- Mit der **Diff-Waiting-Time**-Funktion wurden Modelle generiert, die klassische Verfahren in den meisten Szenarien klar übertreffen. Allerdings weisen einzelne Varianten unter hoher Last (*random_heavy*, *evening_peak*) starke Varianz und Leistungseinbrüche auf.
- Die **Queue**-Funktion führte zu besonders konsistenten Resultaten, bei denen Wartezeiten und Rückstaus nahezu vollständig reduziert werden konnten. Die Stabilität über verschiedene Seeds hinweg war hier am höchsten.
- Die **Real-World**-Funktion ermöglichte praxisnahe Modelle, die in regulären Szenarien sehr effizient arbeiteten. Unter unregelmäßigen und schwer prognostizierbaren Lastbedingungen zeigte sich jedoch eine eingeschränkte Generalisierungsfähigkeit.
- Mit der **CO_2-Emissions**-Funktion konnten neben Effizienzgewinnen auch ökologische Verbesserungen erzielt werden. Diese Modelle kombinierten niedrige Emissionen mit hoher Verkehrseffizienz und erwiesen sich insgesamt als besonders stabile Variante, mit Ausnahme einzelner Schwächen in Hochlastszenarien.

Zusammenfassend lässt sich feststellen, dass Deep-RL-gestützte Steuerungsstrategien ein hohes Potenzial zur Optimierung urbaner Verkehrsflüsse besitzen. Während sich je nach Rewardfunktion unterschiedliche Stärken und Schwächen zeigen, übertreffen alle untersuchten Ansätze die etablierten Baselines teils deutlich. Zugleich wird ersichtlich, dass die Fixed-Time-Steuerung, trotz ihres Alters und ihrer vereinfachten Logik, nach wie vor ein leistungsstarker und praxisrelevanter Vergleichsmaßstab ist. Die Ergebnisse unterstreichen jedoch zugleich die Notwendigkeit, die Generalisierungsfähigkeit unter variierenden und komplexen Lastbedingungen weiter zu verbessern, um den Übergang von simulationsbasierten Experimenten hin zu robusten Realweltanwendungen zu ermöglichen.

6 Herausforderungen und Limitationen

Obwohl die Ergebnisse der Evaluation die Leistungsfähigkeit von Deep-RL-Ansätzen zur Verkehrsflussoptimierung verdeutlichen, bestehen verschiedene technische, methodische und konzeptionelle Einschränkungen. Im Folgenden werden die zentralen Herausforderungen systematisch dargestellt.

6.1 Technische und methodische Hürden

Ein wesentliches Problem ergab sich aus der verwendeten Simulationsumgebung SUMO. Da SUMO nicht *thread-sicher* ist, konnte das Training nicht parallel auf mehreren Instanzen durchgeführt werden. Dies führte dazu, dass ausschließlich CPU-basiertes Training möglich war, was die Trainingszeiten erheblich verlängerte. Ein GPU-basiertes oder verteiltes Training hätte hier deutliche Effizienzgewinne ermöglicht.

Darüber hinaus erforderte die Abstimmung der Hyperparameter (z. B. Lernrate, Discount-Faktor, Explorationsparameter) umfangreiche manuelle Experimente. Zwar konnten stabile Konfigurationen gefunden werden, doch bleibt der Prozess zeitaufwändig und fehleranfällig. Eine *automatisierte Hyperparameter-Optimierung* (z. B. mittels Bayesian Optimization oder Population-Based Training) könnte diesen Aufwand in zukünftigen Arbeiten erheblich reduzieren.

6.2 Repräsentativität und Qualität der Daten

Die zugrundeliegenden Straßennetze basierten auf OpenStreetMap-Daten. Diese Daten sind zwar frei verfügbar und weltweit verfügbar, weisen jedoch erhebliche Schwächen auf. Dazu zählen fehlerhafte oder unvollständige Geometrien, fehlende Informationen zu Fahrstreifen, Tempolimits oder Ampelphasen sowie Inkonsistenzen im Datenmodell. Um die Netze nutzbar zu machen, war daher eine umfangreiche manuelle Nachbearbeitung notwendig, was den Arbeitsaufwand spürbar erhöhte.

Zudem berücksichtigen die verwendeten Netze ausschließlich den motorisierten Individualverkehr. Wichtige Verkehrsteilnehmer wie Fußgänger, Radfahrende oder öffentlicher Nahverkehr (Bus, Bahn) konnten nicht integriert werden, da die entsprechenden Netzelemente in OSM oft unvollständig vorliegen und in SUMO nur eingeschränkt realistisch abgebildet werden können. Dies reduziert die Übertragbarkeit der Ergebnisse auf komplexe urbane Verkehrsszenarien.

6.3 Realitätsnähe der Simulation

SUMO bildet den Straßenverkehr stark vereinfacht ab. Fahrzeuge folgen deterministischen Mikroskopiemodellen und zeigen kein adaptives Verhalten, wie es in der realen Welt vorkommt. So können Fahrzeuge beispielsweise bei Hindernissen nicht eigenständig ausweichen oder zurücksetzen, was teilweise zu *Deadlocks* führt. Auch dynamische Aspekte wie Baustellen, Haltestellen, kurzfristige Störungen oder Unfallereignisse sind nicht modelliert.

Darüber hinaus unterscheiden sich die in SUMO verfügbaren Beobachtungsmetriken von realen Detektoren in Städten. Induktionsschleifen oder Kamerasysteme liefern in der Realität oftmals unvollständige oder verrauschte Daten, während die Simulation nahezu perfekte Werte bereitstellt. Die Übertragbarkeit der Modelle in reale Systeme ist daher eingeschränkt, solange keine Verfahren zur Modellierung von Messfehlern oder Unsicherheiten integriert werden.

6.4 Generalisierbarkeit der Ergebnisse

Die Modelle wurden auf einem spezifischen Netz (Karlsruhe) und unter definierten Szenarien (low_flow, medium_flow, high_flow) trainiert. Zwar zeigen die Ergebnisse deutliche Verbesserungen gegenüber den Baselines, doch ist nicht gesichert, dass diese Modelle ohne Weiteres auf andere Städte, Straßennetze oder Verkehrsmuster übertragbar sind.

Besonders das Szenario random_heavy offenbarte Schwächen einzelner Modelle. Hier traten teilweise drastisch erhöhte Wartezeiten und Stopps auf, die mit hohen Standardabweichungen verbunden waren. Dies zeigt, dass die Modelle in komplexen und unregelmäßigen Lastsituationen weniger robust sind und Generalisation eine der zentralen Herausforderungen bleibt.

7 Fazit und Ausblick

7.1 Zusammenfassung der wichtigsten Erkenntnisse

Die Arbeit hat gezeigt, dass Deep-Reinforcement-Learning-Ansätze ein erhebliches Potenzial zur Optimierung der Steuerung von Lichtsignalanlagen besitzen. Über alle betrachteten Reward-Funktionen hinweg konnten die Modelle die klassischen Baselines deutlich übertreffen. Insbesondere die Actuated-Baseline erwies sich in allen Szenarien als instabil und ineffizient, während die Fixed-Time-Baseline, wie aus ihrer weiten

Verbreitung in realen Städten zu erwarten, robuste und solide Ergebnisse lieferte. Dennoch gelang es den RL-Modellen, selbst diese starke Referenz in den meisten Metriken zu übertreffen.

Die Evaluation verdeutlichte, dass sich durch lernbasierte Steuerungen die mittlere Wartezeit reduzieren, die Anzahl stoppender Fahrzeuge verringern und die durchschnittliche Reisegeschwindigkeit erhöhen lässt. Gleichzeitig blieb der Gesamtdurchsatz nahezu immer auf maximalem Niveau, sodass Effizienzsteigerungen nicht mit einem Verlust an abgefertigten Fahrzeugen einhergingen. Besonders deutlich wurden die Vorteile in regulären und gleichmäßigen Verkehrsszenarien, in denen die Modelle konsistent nahe am Optimum operierten.

Gleichzeitig zeigten die Ergebnisse, dass die Robustheit stark von der gewählten Reward-Funktion abhängt. Während Queue- und CO_2-basierte Modelle eine sehr hohe Stabilität und Reproduzierbarkeit aufwiesen, traten bei Diff-Waiting-Time- und Real-World-basierten Modellen unter Hochlastbedingungen signifikante Schwankungen auf. Die Analysen unterstrichen zudem, dass besonders in unregelmäßigen Szenarien `random_heavy` die Generalisierungsfähigkeit eine wesentliche Herausforderung bleibt.

7.2 Mögliche Weiterentwicklungen

Aus den Ergebnissen ergeben sich mehrere konkrete Ansatzpunkte für weiterführende Arbeiten:

- **Verbesserte Reward-Funktionen:** Eine Kombination mehrerer Zielgrößen (z. B. Wartezeit, Emissionen, Durchsatz) in einem Multi-Objective- oder Curriculum-Learning-Ansatz könnte helfen, die Stärken einzelner Funktionen zu vereinen und die Robustheit zu erhöhen.
- **Automatisierte Hyperparameter-Optimierung:** Der hohe manuelle Abstimmungsaufwand könnte durch Frameworks wie `Optuna` oder `Ray Tune` deutlich reduziert werden. Diese Werkzeuge erlauben eine strukturierte Suche im Hyperparameter-Raum und steigern die Replizierbarkeit und Effizienz der Modellentwicklung.
- **Integration weiterer Verkehrsteilnehmer:** Zukünftige Arbeiten sollten auch Fußgänger, Radfahrende sowie den öffentlichen Nahverkehr berücksichtigen, um eine ganzheitlichere Optimierung urbaner Mobilität zu ermöglichen.
- **Realistischere Sensordaten:** Die Simulation sollte um Modelle erweitert werden, die Messfehler und Unsicherheiten realer Detektoren abbilden. Dadurch könnten die entwickelten Steuerungsstrategien praxisnäher validiert werden.
- **Skalierbarkeit und Verteilte Systeme:** Ein wichtiger Schritt besteht in der Übertragung der Ansätze auf größere Netze und den Einsatz verteilter Multi-Agent-Systeme, die ganze Stadtteile koordinieren können.
- **Kommunikation zwischen Agenten:** Ein weiterer Forschungspfad ist die Erweiterung hin zu kooperativen Steuerungen, bei denen die einzelnen Agenten Informationen untereinander austauschen. Dies könnte eine bessere globale Koordination ermöglichen und die Systemstabilität in hochdynamischen Situationen deutlich erhöhen.
- **Verbundene und autonome Fahrzeuge:** Mit Blick in die Zukunft könnte die Einbindung von Connected-Vehicle-Technologien und autonomen Fahrzeugen die Steuerung auf eine neue Ebene heben. RL-Agenten könnten direkt mit

Fahrzeugen interagieren, Fahrbefehle optimieren und so einen noch effizienteren Verkehrsfluss realisieren.

7.3 Relevanz für reale Verkehrsplanung

Die Ergebnisse der Arbeit verdeutlichen, dass lernbasierte Steuerungsverfahren das Potenzial haben, klassische Verfahren in der Verkehrssteuerung nicht nur zu ergänzen, sondern in vielen Aspekten zu übertreffen. Besonders die Fixed-Time-Steuerung, trotz ihrer weiten Verbreitung und Robustheit, konnte in zentralen Kennzahlen durch die RL-Modelle überboten werden. Dies zeigt, dass der Einsatz von Deep Reinforcement Learning in der Verkehrsplanung eine vielversprechende Perspektive darstellt.

Für die praktische Anwendung ist jedoch entscheidend, dass die Modelle eine hohe Robustheit und Generalisierungsfähigkeit erreichen. Nur wenn sich die Verfahren auch unter komplexen, dynamischen und teilweise unvorhersehbaren Bedingungen zuverlässig verhalten, ist ein Einsatz in urbanen Verkehrsnetzen realistisch. Die in dieser Arbeit aufgezeigten Schwächen in Szenarien mit hoher Last und unregelmäßigen Mustern verdeutlichen, dass hierfür noch Forschungs- und Entwicklungsbedarf besteht.

Darüber hinaus eröffnet die Arbeit Perspektiven im Kontext zukünftiger **Smart-City-Infrastrukturen**. Mit dem zunehmenden Einsatz von *Connected Vehicles* und autonomen Fahrzeugsystemen entsteht die Möglichkeit, RL-Agenten direkt mit Fahrzeugen und Sensornetzen interagieren zu lassen. Eine solche Integration würde es erlauben, nicht nur Ampelanlagen, sondern das gesamte Verkehrssystem koordiniert und adaptiv zu steuern. Besonders die Kommunikation zwischen Agenten und Fahrzeugen könnte einen entscheidenden Beitrag zur Vermeidung von Staus, zur Reduktion von Emissionen und zur Erhöhung der Verkehrssicherheit leisten.

Zusammenfassend liefert die Arbeit einen klaren Hinweis darauf, dass Deep-RL-gestützte Verkehrssteuerungen langfristig einen wichtigen Beitrag zur Reduktion von Staus, Wartezeiten und Emissionen leisten können. Damit stellen sie einen relevanten Baustein für die Gestaltung zukünftiger, nachhaltiger und effizienter urbaner Mobilität dar. Zentral bleibt jedoch die Weiterentwicklung in drei Bereichen: *Generalisierungsfähigkeit*, *Hyperparameter-Optimierung* und *praxisnahe Sensordaten*. Erst deren Kombination, in Verbindung mit der Einbindung von kooperativen Agentensystemen und vernetzten Fahrzeugen, wird den Übergang von simulationsbasierten Ansätzen zu robusten Realweltanwendungen im Sinne einer intelligenten und vernetzten Mobilität ermöglichen.

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`.

```python
# ====== Bibliotheken und Module ======
# Standard-Module für Dateiverwaltung, Zeit und Regex
import os
import re
import time
import datetime
import random

# SUMO-Interface (TraCI) für Simulation
import traci

# Mathematische und numerische Berechnungen
import numpy as np

# PyTorch für neuronale Netze und Reproduzierbarkeit
import torch

# Stable-Baselines3 (RL-Algorithmen, hier PPO)
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import VecNormalize, VecMonitor
from stable_baselines3.common.callbacks import BaseCallback, CallbackList

# SUMO-RL-Umgebung (PettingZoo-kompatibel)
from sumo_rl.environment.env import parallel_env

# SuperSuit - Hilfsfunktionen, um PettingZoo-Umgebungen mit SB3 zu verwenden
from supersuit import (
    pad_observations_v0,          # Padding für Beobachtungen, um feste Größe zu garantieren
    pad_action_space_v0,          # Padding für Aktionsraum
    pettingzoo_env_to_vec_env_v1, # Konvertierung zu SB3-kompatiblem Vektor-Env
    concat_vec_envs_v1            # Mehrere Envs parallel laufen lassen
)

# Gymnasium für RL-Umgebungs-Schnittstellen
import gymnasium as gym
from gymnasium import Wrapper


# ====== Trainings-Setup ======
SEEDS = [143534, 456, 635768, 13755]  # Verschiedene Zufalls-Seed-Werte für reproduzierbare Runs
```

```python
ROUTE_FILES = [
    "flows_low.rou.xml",
    "flows_medium.rou.xml",
    "flows_high.rou.xml",
]

# ====== Schedules für Hyperparameter-Anpassung ======
# (Funktionen, die während des Trainings den Wert z. B. von Lernrate oder Clip-Bereich dynamisch anpassen)
def adaptive_entropy_schedule(start=0.01):
    return lambda progress: max(0.001, start * (1 - progress))

def dynamic_clip_range(start=0.2, end=0.1):
    return lambda pr: end + (start - end) * pr

def cosine_clip(start=0.2, end=0.1):
    return lambda pr: end + (start - end) * 0.5 * (1 + np.cos(np.pi * (1 - pr)))

def linear_schedule(start):
    return lambda progress: start * (1 - progress)

def cosine_warmup_floor(start=3e-4, warmup_frac=0.05, min_lr_frac=0.1):
    """
    Lernrate: Erst linear hochfahren (Warmup), dann mit Cosinus auf Minimalwert absenken.
    """
    min_lr = start * min_lr_frac
    warmup_frac = max(0.0, min(0.5, warmup_frac))
    def schedule(progress_remaining: float) -> float:
        t = 1.0 - progress_remaining
        if t < warmup_frac:
            base = start * 0.1 + (start - start * 0.1) * (t / warmup_frac)
        else:
            tt = (t - warmup_frac) / max(1e-8, (1.0 - warmup_frac))
            cos_term = 0.5 * (1 + np.cos(np.pi * tt))
            base = min_lr + (start - min_lr) * cos_term
        return float(base)
    return schedule

# ====== Hilfsfunktionen und Callbacks ======
# (Modelle finden, Checkpoints speichern, Metriken loggen, bestes Modell sichern)
# ====== Letzten vollständigen Run finden ======
def find_latest_complete_run(base_dir="runs", prefix="ppo_sumo_"):
    """
    Sucht im 'runs'-Ordner nach dem neuesten Trainingslauf, der
    - eine gespeicherte VecNormalize-Instanz hat
    - und entweder ein finales Modell oder mindestens einen Checkpoint.
    Gibt die Pfade zu Run-Ordner, Modell und Normalisierungsdatei zurück.
    """
    subdirs = sorted(
        [d for d in os.listdir(base_dir) if d.startswith(prefix)],
        reverse=True
    )
    for d in subdirs:
        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+",
                ↪  x)[0]), reverse=True)
110             best_checkpoint = checkpoint_models[0]
111             return dir_path, os.path.join(dir_path, best_checkpoint),
                ↪  norm_path
112
113     return None
114
115 def make_env(seed, route_files):
116     def _init():
117         env = parallel_env(
118             net_file="map.net.xml",
119             route_file=route_files[0],  # Platzhalter
120             use_gui=False,
121             num_seconds=4096,
122             reward_fn="diff-waiting-time",
123             min_green=5,
124             max_depart_delay=100,
125             sumo_seed=seed,
126             add_system_info=True,
127             add_per_agent_info=False,
128         )
129         if hasattr(env, "seed"):
130             env.seed(seed)
131
132         orig_reset = env.reset
133         idx = {"i": -1}  # mutierbares Zähl-Objekt im Closure
134
135         def reset_with_round_robin(**kwargs):
136             idx["i"] = (idx["i"] + 1) % len(route_files)
137             new_route = route_files[idx["i"]]
138             env.route_file = new_route
139             if hasattr(env, "sumo_seed"):
140                 env.sumo_seed = seed
141             print(f"\n[DEBUG] Reset → Route: {new_route} | Seed: {seed}\n",
                ↪  flush=True)
142             return orig_reset(**kwargs)
143
144         env.reset = reset_with_round_robin
145         return env
146     return _init
147
```

```python

def shorten_key(orig_key: str) -> str:
    return orig_key.replace("system_", "")

# ====== Callback: Zeitbasiertes Speichern ======
class TimeBasedCheckpointCallback(BaseCallback):
    """
    Speichert Modell und Normalisierungsdaten in festen Zeitintervallen (Sekunden).
    """
    def __init__(self, save_interval_sec, save_path, name_prefix="ppo_sumo_model", verbose=0):
        super().__init__(verbose)
        self.save_interval_sec = save_interval_sec
        self.save_path = save_path
        self.name_prefix = name_prefix
        self.last_save_time = time.time()

    def _on_step(self) -> bool:
        return True  # Keine Aktion bei jedem einzelnen Step

    def _on_rollout_end(self) -> bool:
        # Am Ende eines Rollouts prüfen, ob das Zeitintervall abgelaufen ist
        current_time = time.time()
        if current_time - self.last_save_time >= self.save_interval_sec:
            timestep = self.num_timesteps
            filename = f"{self.name_prefix}_{timestep}_steps"
            self.model.save(os.path.join(self.save_path, filename + ".zip"))
            if hasattr(self.training_env, "save"):
                self.training_env.save(os.path.join(self.save_path, f"{filename}_vecnormalize.pkl"))
            print(f"[Checkpoint] Modell gespeichert bei {timestep} Schritten ({filename})")
            self.last_save_time = current_time
        return True


# ====== Callback: Metriken aus der Env loggen ======
class EpisodeMetricsLoggerCallback(BaseCallback):
    def __init__(self, prefix="episode", verbose=0):
        super().__init__(verbose)
        self.prefix = prefix
        self.verbose = verbose
        self.sums = {}
        self.counts = {}
        self.last_totals = {}

    def _on_step(self) -> bool:
        dones = self.locals.get("dones")
        infos = self.locals.get("infos")
        if infos is None:
            return True

        for i, info in enumerate(infos):
            if not isinstance(info, dict):
                continue

```

```
            if dones is not None and dones[i]:
                # --- Episode zu Ende ---
                fin = info.get("final_info") or info.get("terminal_info")
                if isinstance(fin, dict):
                    for k, v in fin.items():
                        if not isinstance(v, (int, float)) or not
                        ↪  np.isfinite(v):
                            continue
                        if k.startswith("system_mean_"):
                            self.sums[k] = self.sums.get(k, 0.0) + float(v)
                            self.counts[k] = self.counts.get(k, 0) + 1
                        elif k.startswith("system_total_"):
                            self.last_totals[k] = float(v)
            else:
                # --- Nur Zwischenschritt, solange Episode noch läuft ---
                for k, v in info.items():
                    if not isinstance(v, (int, float)) or not np.isfinite(v):
                        continue
                    if k.startswith("system_mean_") or k in [
                        "system_total_waiting_time",
                        "system_total_stopped",
                        "system_total_running",
                    ]:
                        self.sums[k] = self.sums.get(k, 0.0) + float(v)
                        self.counts[k] = self.counts.get(k, 0) + 1
                    elif k.startswith("system_total_"):
                        self.last_totals[k] = float(v)

        # Episode fertig → loggen
        if dones is not None and any(dones):
            for k, total in self.sums.items():
                mean_val = total / max(1, self.counts.get(k, 1))
                short_key = shorten_key(k)
                self.logger.record(f"{self.prefix}/{short_key}", mean_val)
                if self.verbose:
                    print(f"[EpisodeMetrics] {short_key} (mean) =
                    ↪  {mean_val:.3f}")

            for k, v in self.last_totals.items():
                short_key = shorten_key(k)
                self.logger.record(f"{self.prefix}/{short_key}", v)
                if self.verbose:
                    print(f"[EpisodeMetrics] {short_key} (total) = {v:.0f}")

            # Reset für nächste Episode
            self.sums.clear()
            self.counts.clear()
            self.last_totals.clear()

        return True

# ====== Callback: Bestes Modell speichern ======
class BestModelSaverCallback(BaseCallback):
    """
    Speichert das Modell mit dem bisher höchsten mittleren Episodenreward.
    """
    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,
                        ↪  "best_model_vecnormalize.pkl")
273                     self.model.env.save(norm_path)
274                 print(f"[AUTOLOG] Neuer Bestwert {mean_rew:.2f} → best_model
                    ↪  gespeichert!", flush=True)
275 
276 
277 # ====== Haupt-Trainingsschleife ======
278 for SEED in SEEDS:
279     # Reproduzierbarkeit sicherstellen
280     np.random.seed(SEED)
281     torch.manual_seed(SEED)
282 
283     # Log-Verzeichnis erstellen
284     now = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
285     log_dir = os.path.join("runs", f"ppo_sumo_{SEED}_{now}")
286     os.makedirs(log_dir, exist_ok=True)
287 
288     print(f"\n[INFO] Starte Training mit Seed: {SEED}")
289 
290     # SUMO-Umgebung initialisieren
291     env = make_env(SEED, ROUTE_FILES)()
292 
293     # Falls die Env einen seed()-Aufruf unterstützt
294     if hasattr(env, "seed"):
295         env.seed(SEED)
296 
297     # Anpassung der Beobachtungen und Aktionen an SB3
298     env = pad_observations_v0(env)
299     env = pad_action_space_v0(env)
300     env = pettingzoo_env_to_vec_env_v1(env)
301 
302     # WICHTIG: trotzdem concat_vec_envs_v1 mit num_vec_envs=1
303     env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=1,
        ↪  base_class="stable_baselines3")
304 
305 
306     # Logging und Normalisierung
307     env = VecMonitor(env, filename=os.path.join(log_dir, "monitor.csv"))
308     env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.0)
309 
```

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

```
    finally:
        try:
            env.save(os.path.join(log_dir, "vecnormalize.pkl"))
        except Exception as e:
            print(f"[WARNUNG] VecNormalize konnte nicht gespeichert werden:
            ↪ {e}")
        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.

```
import os
import re
import time
import datetime
import traci
import numpy as np
import torch
import json
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import VecNormalize, VecMonitor
from stable_baselines3.common.callbacks import BaseCallback, CallbackList
from sumo_rl.environment.env import parallel_env
from supersuit import (
    pad_observations_v0,
    pad_action_space_v0,
    pettingzoo_env_to_vec_env_v1,
    concat_vec_envs_v1
)
from gym import Wrapper

# ==== Seed setzen ====
SEED = 42
np.random.seed(SEED)
torch.manual_seed(SEED)

# ==== Adaptive Parameter-Schedules ====
def dynamic_clip_range(start=0.2):
    return lambda progress: max(0.1, start * (1 - 0.5 * progress))

def linear_schedule(start):
    return lambda progress: start * (1 - progress)

# ==== Finde letzten vollständigen Run ====
def find_latest_complete_run(base_dir="runs", prefix="ppo_sumo_"):
    subdirs = sorted(
        [d for d in os.listdir(base_dir) if d.startswith(prefix)],
        reverse=True
    )
    for d in subdirs:
        dir_path = os.path.join(base_dir, d)
        norm_path = os.path.join(dir_path, "vecnormalize.pkl")
        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+",
                ↪  x)[0]), reverse=True)
55              best_checkpoint = checkpoint_models[0]
56              return dir_path, os.path.join(dir_path, best_checkpoint),
                ↪  norm_path
57
58      return None
59
60  # ==== Zeitbasierter Checkpoint Callback ====
61  class TimeBasedCheckpointCallback(BaseCallback):
62      def __init__(self, save_interval_sec, save_path,
        ↪  name_prefix="ppo_sumo_model", verbose=0):
63          super().__init__(verbose)
64          self.save_interval_sec = save_interval_sec
65          self.save_path = save_path
66          self.name_prefix = name_prefix
67          self.last_save_time = time.time()
68
69      def _on_step(self) -> bool:
70          return True
71
72      def _on_rollout_end(self) -> bool:
73          current_time = time.time()
74          if current_time - self.last_save_time >= self.save_interval_sec:
75              timestep = self.num_timesteps
76              filename = f"{self.name_prefix}_{timestep}_steps"
77              self.model.save(os.path.join(self.save_path, filename + ".zip"))
78              if hasattr(self.training_env, "save"):
79                  self.training_env.save(os.path.join(self.save_path,
                    ↪  f"{filename}_vecnormalize.pkl"))
80              print(f"[Checkpoint] Modell gespeichert bei {timestep} Schritten
                ↪  ({filename})")
81              self.last_save_time = current_time
82          return True
83
84  # ==== Learning Rate Logger Callback ====
85  class LearningRateLoggerCallback(BaseCallback):
86      def __init__(self, verbose=0):
87          super().__init__(verbose)
88
89      def _on_step(self) -> bool:
90          lr = self.model.lr_schedule(self.num_timesteps /
            ↪  self.model._total_timesteps)
91          self.logger.record("train/learning_rate", lr)
92          return True
93
```

```
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,
    ↪  base_class="stable_baselines3")
121 env = VecMonitor(env)
122
123 # ==== Modell laden oder neu starten ====
124 result = find_latest_complete_run()
125 if result:
126     latest_run_dir, model_path, normalize_path = result
127     print("Fortsetzung wird gestartet mit:")
128     print(f"Verzeichnis : {latest_run_dir}")
129     print(f"Modell      : {model_path}")
130     print(f"Normalize   : {normalize_path}\n")
131
132     env = VecNormalize.load(normalize_path, env)
133     env.training = True
134     env.norm_reward = True
135
136     model = PPO.load(model_path, env=env, tensorboard_log=log_dir, verbose=1,
        ↪  device="cpu")
137     print(f"[INFO] Modell startet bei {model.num_timesteps} Timesteps.")
138 else:
139     print("[INFO] Kein vorheriges Modell gefunden. Starte frisches
        ↪  Training.\n")
140     env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.0)
141     model = PPO(
142         policy="MlpPolicy",      # Mehrschicht-Perzeptron-Policy
143         env=env,
144         verbose=1,               # Ausführliches Logging
145         tensorboard_log=log_dir, # TensorBoard-Pfad
146         batch_size=256,          # Minibatch-Größe für PPO
147         n_steps=2048,            # Rollout-Länge
```

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

```
            callback=callbacks,
        )
        model.save(os.path.join(log_dir, "model.zip"))
        env.save(os.path.join(log_dir, "vecnormalize.pkl"))
        print(f"\n[INFO] Training abgeschlossen. Modell gespeichert unter:
        ↪ {log_dir}")

    except KeyboardInterrupt:
        print("[ABBRUCH] Manuelles Beenden erkannt. Speichere aktuellen Stand...")
        model.save(os.path.join(log_dir, "model_interrupt.zip"))
        env.save(os.path.join(log_dir, "vecnormalize_interrupt.pkl"))

    except Exception as e:
        print(f"\n[FEHLER] Während des Trainings aufgetreten: {e}")

    finally:
        try:
            env.save(os.path.join(log_dir, "vecnormalize.pkl"))
        except Exception as e:
            print(f"[WARNUNG] VecNormalize konnte nicht gespeichert werden: {e}")
        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.

```
def diff_waiting_time_reward(traffic_signal):
    ts_wait = sum(traffic_signal.get_accumulated_waiting_time_per_lane()) /
    ↪ 100.0
    reward = traffic_signal.last_ts_waiting_time - ts_wait
    traffic_signal.last_ts_waiting_time = ts_wait
    return reward
```

B.2 `queue`

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

```
def queue_reward(traffic_signal):
    return -traffic_signal.get_total_queued()

def get_total_queued(traffic_signal) -> int:
    """Returns the total number of vehicles halting in the intersection."""
    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.

```python
def realworld_reward(traffic_signal):
    # Speed (0-7.5 m/s -> 0-1)
    avg_speed = traffic_signal.get_average_speed()
    speed_term = min(max(avg_speed, 0.0), 7.5) / 7.5

    # Queue (0-20 Fzg -> 0-1)
    total_queue = traffic_signal.get_total_queued()
    queue_term = min(max(total_queue, 0), 20) / 20.0

    # Mean waiting time (0-10 s -> 0-1)
    waits_per_lane = traffic_signal.get_accumulated_waiting_time_per_lane()
    mean_wait = sum(waits_per_lane) / len(waits_per_lane) if waits_per_lane
    ↪  else 0.0
    wait_term = min(max(mean_wait, 0.0), 20.0) / 10.0

    reward = speed_term - queue_term - wait_term
    return reward
```

B.4 `emissions`

Diese Variante erweitert den Reward zusätzlich um einen Term für die CO_2-Emissionen, sodass sowohl Verkehrsfluss als auch Nachhaltigkeit berücksichtigt werden.

```python
def emissions_reward(traffic_signal):
    env = getattr(traffic_signal, "env", None)
    if env is None or getattr(env, "sumo", None) is None:
        return 0.0
    if float(env.sim_step) >= float(env.sim_max_time):
        return 0.0

    sumo = env.sumo

    # Speed (0-7.5 m/s → 0..1)
    avg_speed = traffic_signal.get_average_speed()
    speed_term = min(max(avg_speed, 0.0), 7.5) / 7.5

    # Queue (0-20 → 0..1)
    total_queue = traffic_signal.get_total_queued()
    queue_term = min(max(total_queue, 0), 20) / 20.0

    # Wait (0-10 s → 0..1)
    waits = traffic_signal.get_accumulated_waiting_time_per_lane()
    mean_wait = (sum(waits) / len(waits)) if waits else 0.0
    wait_term = min(max(mean_wait, 0.0), 10.0) / 10.0

    # Emissionen
    try:
        lanes = getattr(traffic_signal, "lanes", [])
        total_co2 = sum(sumo.lane.getCO2Emission(lane) for lane in lanes)
        n_veh = sum(sumo.lane.getLastStepVehicleNumber(lane) for lane in
        ↪  lanes)
        BASELINE = 300.0 * max(1, len(lanes))
        CAP      = 2000.0 * max(1, len(lanes))
        co2_term = max(0.0, min(total_co2 - BASELINE, CAP - BASELINE)) / (CAP
        ↪  - BASELINE)
```

```
    except Exception:
        co2_term = 0.0

    reward = speed_term - queue_term - wait_term - co2_term
    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`.

```
import os, json, numpy as np
import glob
from stable_baselines3 import PPO
from stable_baselines3.common.vec_env import VecNormalize, VecMonitor
from stable_baselines3.common.logger import configure
from sumo_rl.environment.env import parallel_env
from supersuit import pad_observations_v0, pad_action_space_v0
from supersuit import pettingzoo_env_to_vec_env_v1, concat_vec_envs_v1

# ----- Config -----
RUNS = sorted(glob.glob(os.path.join("runs", "ppo_sumo_*")))
MODEL_NAME  = "best_model.zip"
N_EPISODES  = 10
EP_LENGTH_S = 4096
EP_SEEDS    = [12345, 67890, 13579, 24680, 11223, 44556, 77889, 99100, 31415,
↪  27182]
SCENARIOS   = [
    {"name": "morning_peak", "route_file": "flows_morning.rou.xml"},
    {"name": "evening_peak", "route_file": "flows_evening.rou.xml"},
    {"name": "uniform",      "route_file": "flows_uniform.rou.xml"},
    {"name": "random_heavy", "route_file": "flows_random_heavy.rou.xml"},
]

# ----- Env Factory -----
def make_env(route_file, sumo_seed):
    print(f"[DEBUG] Creating SUMO env with route={route_file},
    ↪  seed={sumo_seed}")
    env = parallel_env(
        net_file="map.net.xml",
        route_file=route_file,
        use_gui=False,
        num_seconds=EP_LENGTH_S,
        reward_fn=dummy_reward,
        min_green=5,
        max_depart_delay=100,
        sumo_seed=sumo_seed,
        add_system_info=True,
        add_per_agent_info=False,
```

```
    )
    env = pad_observations_v0(env)
    env = pad_action_space_v0(env)
    env = pettingzoo_env_to_vec_env_v1(env)
    env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=1,
    ↪  base_class="stable_baselines3")
    env = VecMonitor(env)
    return env

# ----- Model Loader -----
def load_model_and_norm(env, run_dir):
    vecnorm_path = os.path.join(run_dir, "vecnormalize.pkl")
    model_path   = os.path.join(run_dir, MODEL_NAME)

    #print(f"[DEBUG] Loading VecNormalize from {vecnorm_path}")
    env = VecNormalize.load(vecnorm_path, env)
    env.training = False
    env.norm_reward = False

    #print(f"[DEBUG] Loading PPO model from {model_path}")
    model = PPO.load(model_path, env=env, device="cpu")
    return model, env

# ----- Rollout -----
def rollout(model, env):
    obs = env.reset()
    dones = [False]

    sums = {}
    counts = {}
    last_totals = {}

    while True:
        action, _ = model.predict(obs, deterministic=True)
        obs, rewards, dones, infos = env.step(action)

        info = infos[0] if isinstance(infos, list) else infos
        if not isinstance(info, dict):
            info = {}

        # Wenn Episode zu Ende ist:
        if dones[0]:
            # Falls vorhanden, final_info/terminal_info verwenden
            fin = info.get("final_info") or info.get("terminal_info")
            if isinstance(fin, dict):
                # Mittelwerte vom finalen Step noch einrechnen
                for k, v in fin.items():
                    if k.startswith("system_mean_") and isinstance(v, (int,
                    ↪  float)) and np.isfinite(v):
                        sums[k] = sums.get(k, 0.0) + float(v)
                        counts[k] = counts.get(k, 0) + 1
                # Totals aus final_info (echte Endstände)
                for k, v in fin.items():
                    if k.startswith("system_total_") and isinstance(v, (int,
                    ↪  float)) and np.isfinite(v):
                        last_totals[k] = float(v)
            break
```

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

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

```python

def to_serializable(obj):
    if isinstance(obj, (np.integer,)):
        return int(obj)
    elif isinstance(obj, (np.floating,)):
        return float(obj)
    elif isinstance(obj, (np.ndarray,)):
        return obj.tolist()
    return str(obj)

# ----- Evaluation Loop -----
# ----- Evaluation Loop -----
def evaluate():
    results = []
    log_dir_root = os.path.join("evaluation", "logs")

    # Zählung: 2 Baselines + len(RUNS) RL pro (scenario × episode)
    total_episodes = (2 + len(RUNS)) * len(SCENARIOS) * N_EPISODES
    ep_counter = 0

    for sc in SCENARIOS:
        scen_log_dir = os.path.join(log_dir_root, f"eval_{sc['name']}")
        os.makedirs(scen_log_dir, exist_ok=True)
        logger = configure(scen_log_dir, ["tensorboard", "stdout"])

        print(f"[INFO] Evaluating scenario={sc['name']}")

        for ep in range(N_EPISODES):
            ep_seed = EP_SEEDS[ep]

            # --- 1) Fixed-Time ---
            env = make_env_baseline(sc["route_file"], sumo_seed=ep_seed,
            ↪  fixed_time=True)
            ep_counter += 1
            print(f"[PROGRESS] FixedTime | {sc['name']} | Ep
            ↪  {ep+1}/{N_EPISODES} "
                  f"({ep_counter}/{total_episodes})")
            m = rollout_baseline(env)
            m.update({
                "scenario": sc["name"],
                "episode": ep,
                "method": "Baseline_FixedTime"
            })
            results.append(m)

            # --- 2) Actuated ---
            env = make_env_baseline(sc["route_file"], sumo_seed=ep_seed,
            ↪  fixed_time=False)
            ep_counter += 1
            print(f"[PROGRESS] Actuated | {sc['name']} | Ep
            ↪  {ep+1}/{N_EPISODES} "
                  f"({ep_counter}/{total_episodes})")
            m = rollout_baseline(env)
            m.update({
                "scenario": sc["name"],
                "episode": ep,
                "method": "Baseline_Actuated"
```

```
            })
            results.append(m)

            # --- 3) RL-Modelle ---
            for run_dir in RUNS:
                env_raw = make_env(sc["route_file"], sumo_seed=ep_seed)
                model, env = load_model_and_norm(env_raw, run_dir)
                ep_counter += 1
                model_name = os.path.basename(run_dir)
                print(f"[PROGRESS] RL | {sc['name']} | {model_name} "
                      f"| Ep {ep+1}/{N_EPISODES}
                      ↪  ({ep_counter}/{total_episodes})")
                m = rollout(model, env)

                # Seed extrahieren (3. Teil vom Namen)
                parts = model_name.split("_")
                model_seed = parts[2] if len(parts) > 2 else "unknown"

                m.update({
                    "scenario": sc["name"],
                    "episode": ep,
                    "method": f"{model_name}_{model_seed}"
                })
                results.append(m)

            # --- Logging dieser Episode (Baselines + alle RL) ---
            for entry in results[-(2 + len(RUNS)):]:
                for k, v in entry.items():
                    if isinstance(v, (int, float)) and k not in ["episode",
                    ↪  "ep_seed"]:
                        short_key = shorten_key(k)
                        logger.record(f"{entry['method']}/{short_key}", v)
            logger.dump(step=ep)

    results_path = os.path.join("evaluation", "eval_results.json")
    os.makedirs(os.path.dirname(results_path), exist_ok=True)
    with open(results_path, "w") as f:
        json.dump(results, f, indent=2, default=to_serializable)

    print(f"[INFO] Evaluation abgeschlossen. Ergebnisse: {results_path}")

if __name__ == "__main__":
    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* × *Methode* und erzeugt sowohl eine aggregierte Gesamttabelle als auch separate CSV-Dateien pro Methode.

```python
import json
import pandas as pd
import os

# Pfade
json_path = "evaluation/eval_results.json"
raw_csv_path = "evaluation/eval_results_raw.csv"
agg_csv_path = "evaluation/eval_results_agg.csv"

# ------------------------------
# Schritt 1: JSON -> Raw CSV
# ------------------------------
print(f"Lese JSON-Datei: {json_path}")
with open(json_path, "r") as f:
    data = json.load(f)

df = pd.DataFrame(data)
df.to_csv(raw_csv_path, index=False)
print(f"Raw CSV geschrieben: {raw_csv_path}")

# ------------------------------
# Schritt 2: Aggregation
# ------------------------------
# numerische Spalten automatisch finden (alles außer scenario, method,
↪  episode)
numeric_cols = df.select_dtypes(include="number").columns.tolist()
numeric_cols = [c for c in numeric_cols if c not in ["episode"]]  # episode
↪  nicht mitteln

# Aggregationsdict
agg_dict = {}
for col in numeric_cols:
    agg_dict[f"{col}_mean"] = (col, "mean")
    agg_dict[f"{col}_std"] = (col, "std")

# Gruppieren nach Szenario + Methode
agg = df.groupby(["scenario", "method"]).agg(**agg_dict).reset_index()

# Gesamte Aggregation speichern
agg.to_csv(agg_csv_path, index=False)
print(f"Aggregierte Datei geschrieben: {agg_csv_path}")
print("Zeilen:", len(agg))

# ------------------------------
# Schritt 3: Pro-Methode CSVs
# ------------------------------
for method, df_method in agg.groupby("method"):
    safe_name = method.replace(" ", "_").replace("/", "_")
    out_path = f"evaluation/{safe_name}.csv"
    df_method.to_csv(out_path, index=False)
    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.

```
import xml.etree.ElementTree as ET

# === Konfiguration ===
net_file = "karlsruhe.net.xml"

# === Einlesen ===
tree = ET.parse(net_file)
root = tree.getroot()

# === Alle controlledLinks zählen ===
tls_controlled_links = {}
for connection in root.findall("connection"):
    if "tl" in connection.attrib and "linkIndex" in connection.attrib:
        tls_id = connection.attrib["tl"]
        tls_controlled_links.setdefault(tls_id,
        ↪  set()).add(int(connection.attrib["linkIndex"]))

# === Alle Phasen prüfen ===
def check_tls_lengths():
    print("Überprüfe alle TLS auf inkonsistente Phasenlängen...\n")
    any_issues = False
    for logic in root.findall("tlLogic"):
        tls_id = logic.attrib["id"]
        expected_len = len(tls_controlled_links.get(tls_id, []))

        if expected_len == 0:
            print(f" TLS '{tls_id}' hat keine controlledLinks (wird evtl.
            ↪  nicht gesteuert)")
            continue

        for i, phase in enumerate(logic.findall("phase")):
            actual_len = len(phase.attrib["state"])
            if actual_len != expected_len:
                print(f" Phase {i} von TLS '{tls_id}' hat Länge {actual_len},
                ↪  erwartet: {expected_len}")
                print(f"    → state=\"{phase.attrib['state']}\"")
                any_issues = True

    if not any_issues:
        print(" Alle TLS-Phasen stimmen mit ihren controlledLinks überein!")

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.

```
import xml.etree.ElementTree as ET

net_file = "karlsruhe.net.xml"
tree = ET.parse(net_file)
root = tree.getroot()

# Zähle für jedes TLS wie viele signal indices es gibt (controlled links)
tls_signal_indices = {}
for conn in root.findall("connection"):
    if "tl" in conn.attrib and "linkIndex" in conn.attrib:
        tls_id = conn.attrib["tl"]
        tls_signal_indices.setdefault(tls_id,
        ↪  set()).add(int(conn.attrib["linkIndex"]))

# Vergleiche mit den request-Elementen
print("Überprüfe request-Indizes gegen Signalindizes...\n")
any_issues = False
for junction in root.findall("junction"):
    tls_id = junction.attrib.get("id")
    requests = junction.findall("request")
    if tls_id in tls_signal_indices:
        expected_max = len(tls_signal_indices[tls_id])
        for req in requests:
            index = int(req.attrib["index"])
            if index >= expected_max:
                print(f"Junction '{tls_id}': request index {index} > max
                ↪  signal index {expected_max - 1}")
                any_issues = True

if not any_issues:
    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.

```
import xml.etree.ElementTree as ET

net_file = "karlsruhe.net.xml"
output_file = "karlsruhe_fixed_tls.net.xml"

tree = ET.parse(net_file)
root = tree.getroot()

# Finde maximal verwendete Signal-Indices pro TLS
tls_max_index = {}
```

```
for conn in root.findall("connection"):
    tl = conn.get("tl")
    idx = conn.get("linkIndex")
    if tl and idx:
        idx = int(idx)
        tls_max_index[tl] = max(tls_max_index.get(tl, -1), idx)

# Bereinigung
total_removed_requests = 0
total_adjusted_phases = 0
changed_tls = []

for junction in root.findall("junction"):
    tls_id = junction.get("id")
    if tls_id not in tls_max_index:
        continue

    max_idx = tls_max_index[tls_id]
    requests = list(junction.findall("request"))
    removed = 0

    for req in requests:
        req_idx = int(req.get("index"))
        if req_idx > max_idx:
            junction.remove(req)
            removed += 1

    if removed > 0:
        print(f"TLS '{tls_id}': {removed} ungültige <request>-Einträge
        ↪  entfernt.")
        total_removed_requests += removed
        changed_tls.append(tls_id)

    # Kürze zugehörige Phasen
    for tl in root.findall("tlLogic"):
        if tl.get("id") == tls_id:
            adjusted = 0
            for phase in tl.findall("phase"):
                state = phase.get("state")
                if len(state) > max_idx + 1:
                    old_len = len(state)
                    phase.set("state", state[:max_idx + 1])
                    adjusted += 1
            if adjusted > 0:
                print(f" TLS '{tls_id}': {adjusted} <phase>-Strings auf Länge
                ↪  {max_idx + 1} gekürzt.")
                total_adjusted_phases += adjusted
                if tls_id not in changed_tls:
                    changed_tls.append(tls_id)

# Speichern
tree.write(output_file, encoding="utf-8")
print("\n Reparatur abgeschlossen.")
print(f" Gesamt entfernte <request>-Einträge: {total_removed_requests}")
print(f" Gesamt angepasste <phase>-Einträge: {total_adjusted_phases}")
print(f" Betroffene TLS-IDs: {len(changed_tls)} Stück")
for tls in changed_tls:
```

```
    print(f"  - {tls}")
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.

```
from xml.etree import ElementTree as ET

# Manuell gepflegte Dictionary mit {TLS-ID: Anzahl controlledLinks}
controlled_links = {
    "1720933516": 6,
    "3538953167": 2,
    "3664415977": 10,
    "cluster_14795187_1720919996_2670370290_2670370291": 11,
    "cluster_14795804_55474925_6655074904_765746891_#1more": 49,
    "cluster_15431428_1719671850_1720917935": 20,
    "cluster_1590912233_3664415976_5083348337_5083348350": 11,
    "cluster_1692973685_1692973722_1718084055_1718084058_#11more": 36,
    "cluster_1729190097_3687504105": 8,
    "cluster_1744031943_5131521735": 10,
    "joinedS_1623835169_cluster_1137679587_1626739216_1728272870_1728272909_#17more": 33,
    "joinedS_309108716_cluster_11001804363_1125509937_12515596172_1784859792_#5more": 14,
    "joinedS_5092985445_cluster_1590912226_2911376263": 10,
    # ggf. mehr hinzufügen
}

tree = ET.parse("karlsruhe.net.xml")
root = tree.getroot()
changed = False

for logic in root.findall("tlLogic"):
    tl_id = logic.attrib["id"]
    if tl_id not in controlled_links:
        continue

    correct_len = controlled_links[tl_id]
    for phase in logic.findall("phase"):
        state = phase.attrib["state"]
        if len(state) != correct_len:
            new_state = state[:correct_len].ljust(correct_len, 'r')
            print(f" Fixing {tl_id}: {len(state)} → {correct_len}")
            phase.attrib["state"] = new_state
            changed = True

if changed:
    tree.write("karlsruhe_fixed.net.xml")
    print(" Bereinigte Datei gespeichert: karlsruhe_fixed.net.xml")
else:
```

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

```

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

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

```
from xml.etree import ElementTree as ET

tree = ET.parse("karlsruhe.net.xml")
root = tree.getroot()

for logic in root.findall("tlLogic"):
    tl_id = logic.attrib["id"]
    for i, phase in enumerate(logic.findall("phase")):
        state = phase.attrib["state"]
        if len(state) != 57:
            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.

```
from sumo_rl import SumoEnvironment
import traci
import os

def test_tls(tls_id):
    try:
        env = SumoEnvironment(
            net_file="karlsruhe.net.xml",
            route_file="karlsruhe.rou.xml",
            use_gui=False,
            single_agent=True
        )
        env.ts_ids = [tls_id]
        env.reset()
        env.close()
        return True
    except Exception as e:
        print(f" TLS {tls_id} nicht gültig: {e}")
        return False

# Alle TLS holen
try:
    env = SumoEnvironment(
        net_file="karlsruhe.net.xml",
        route_file="karlsruhe.rou.xml",
        use_gui=False,
        single_agent=True
    )
```

```
    all_tls = env.ts_ids
    env.close()
except Exception as e:
    print(" Konnte TLS nicht auslesen:", e)
    all_tls = []

print(f" Teste {len(all_tls)} TLS auf Gültigkeit...\n")
valid_tls = []

for tls_id in all_tls:
    if test_tls(tls_id):
        valid_tls.append(tls_id)

print("\n Gültige TLS:")
print(valid_tls)
```

E.7 `find_relevant_edges.py` – Suche alle relevanten edges

```
import xml.etree.ElementTree as ET

# Konfiguration: Pfad zur .net.xml-Datei und Suchbegriffe
NET_FILE = "network.net.xml"
SUCHBEGRIFFE = ["B10", "B36", "L605", "Durlacher Allee",
↪  "Reinhold-Frank-Straße"]

# Ausgabe-Datei für gefundene Kanten
OUTPUT_FILE = "edges.txt"

def finde_relevante_kanten(net_file, suchbegriffe):
    tree = ET.parse(net_file)
    root = tree.getroot()

    relevante_kanten = []

    for edge in root.findall("edge"):
        name = edge.get("name")
        if name:
            for begriff in suchbegriffe:
                if begriff.lower() in name.lower():
                    relevante_kanten.append((edge.get("id"), name))
                    break  # nicht doppelt eintragen, falls mehrere Begriffe
                    ↪  passen

    return relevante_kanten

if __name__ == "__main__":
    kanten = finde_relevante_kanten(NET_FILE, SUCHBEGRIFFE)

    # Ergebnisse speichern
    with open(OUTPUT_FILE, "w", encoding="utf-8") as f:
        for edge_id, name in kanten:
            f.write(f"{edge_id}\t{name}\n")

    print(f"{len(kanten)} relevante Kanten gefunden.")
    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.

```
<configuration>
  <input>
    <net-file value="network.net.xml"/>
    <route-files value="routes.xml"/>
  </input>
  <time>
    <begin value="0"/>
    <end value="5000"/>
  </time>
</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] *Bundesanstallt 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.ntl.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.

Glossar

SUMO Simulation of Urban MObility, eine quelloffene mikroskopische Verkehrssimulationssoftware, die das Verhalten einzelner Fahrzeuge in Straßennetzen abbildet.. 5