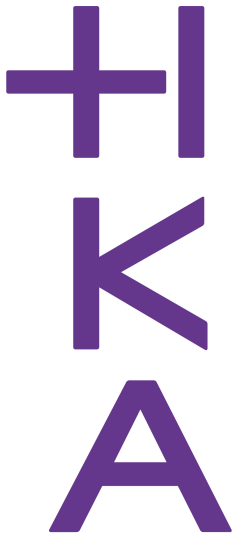
Hochschule Karlsruhe
University of
Applied Sciences

Fakultät für
Informatik und
Wirtschaftsinformatik



Optimierung einer Verkehrssimulation mit KI-basierten Agenten in SUMO

Studiengang Informatik
Sam Weiler
Matr. Nr. 73640

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[**?**], 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. Darüber hinaus verursacht ineffizienter Verkehr einen erheblichen wirtschaftlichen Schaden durch Zeitverluste und Ressourcenverschwendung[**?**].

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.[18] Auch adaptive Verfahren, wie verkehrsabhängige Steuerungen mittels Induktionsschleifen oder Kameras, sind in ihrer Reaktionsfähigkeit beschränkt und oft teuer in der Wartung. Damit bleibt ein enormes Potenzial zur Effizienzsteigerung ungenutzt.[19]

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)[10], eine quelloffene, modular aufgebaute Plattform, die es ermöglicht, Verkehrsflüsse realitätsnah zu modellieren und zu analysieren. In Kombination mit dem Framework `sumo-rl`[11], 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.

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.[10]

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.

- **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.

- **Städtebauliche Verkehrslenkung:** In der Realität regeln Städte Verkehrsflüsse z.B. durch Low-Traffic-Neighbourhoods, Zufahrtsbeschränkungen oder geregelte Zuflusssteuerung, um bestimmte Stadtbereiche zu entlasten. 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 oder eine Darstellung saisonaler sowie städtischer Ereigniszeiten 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. 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.

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. Dennoch sind viele Systeme in der Praxis noch unflexibel oder schwer skalierbar.

2.2 Simulation urbaner Mobilität mit SUMO

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

Besonders relevant für diese Arbeit sind folgende Merkmale:

- **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. 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.
- **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) 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.

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. Dabei verfolgt er das Ziel, eine kumulative Belohnung zu maximieren.

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

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

Der Agent interagiert mit der Umgebung, beobachtet den Zustand, wählt eine Aktion, erhält eine Belohnung und gelangt in einen neuen Zustand. Durch viele Wiederholungen lernt er, welche Entscheidungen langfristig die besten sind.

Wichtige Algorithmen, die in dieser Arbeit potenziell relevant sind, sind:

- **Q-Learning:** Modellfreies, off-policy Lernverfahren zur Annäherung an optimale Aktionen.
- **DQN (Deep Q-Network):** Kombination von Q-Learning mit neuronalen Netzen.
- **PPO (Proximal Policy Optimization):** Policy-basierter RL-Ansatz mit stabiler Optimierung.

2.4 SUMO-RL: Architektur und Funktionalität

`sumo-rl` ist ein Python-Framework, das SUMO mit Reinforcement Learning verbindet. Es basiert auf der `gymnasium`-Schnittstelle und abstrahiert typische Aufgaben wie die Definition von Beobachtungen, Aktionen und Belohnungen für RL-Agenten. Die Umgebung wird durch die Klasse `SumoEnvironment` bereitgestellt.

Zentrale Eigenschaften von `sumo-rl`:[11]

- **TraCI-Integration:** Ermöglicht über das Traffic Control Interface zur Laufzeit den Zugriff auf Fahrzeugdaten, Ampelphasen, Fahrzeugwarteschlangen u. v. m.
- **Ein- und Mehragentenunterstützung:** `sumo-rl` unterstützt sowohl Single-Agent-Setups als auch Multi-Agent-Steuerung über die PettingZoo-API. Jeder gesteuerte Knoten im Netz kann einem eigenen Agenten zugewiesen werden.
- **Beobachtungen:** Die Umgebung liefert Beobachtungsvektoren mit kodierter Ampelphase, Rückstaulänge, Anzahl wartender Fahrzeuge und Fahrzeugdichte je Spur.
- **Aktionen:** Die Agenten treffen diskrete Entscheidungen über Phasenwechsel, wobei `delta_time`, `yellow_time` und `min_green` die zeitliche Dynamik definieren.

- **Belohnungsfunktionen:** Der Standard-Reward basiert auf der Differenz kumulierter Wartezeiten. Eigene Funktionen können bei Initialisierung übergeben werden.
- **Kompatibilität:** Das Framework ist kompatibel mit Stable-Baselines3, PyTorch, TensorFlow, RLlib und anderen gängigen ML-Frameworks.

Beispielhafte Initialisierung:

```
env = SumoEnvironment(
    net_file='net.net.xml',
    route_file='routes.rou.xml',
    use_gui=True,
    reward_fn='diff-waiting-time',
    single_agent=True,
    delta_time=5,
    yellow_time=2,
    min_green=5
)
```

2.5 Verwandte Arbeiten

In den letzten Jahren wurden zunehmend Studien veröffentlicht, die KI-Methoden zur Optimierung der Verkehrssteuerung einsetzen. Eine Auswahl relevanter Forschungsansätze:

- **Wei et al. (2019):** Einsatz von Deep Q-Learning zur Optimierung einer einzelnen Ampel in SUMO mit signifikantem Rückgang der Wartezeiten [15].
- **Chu et al. (2020):** Untersuchung von Multi-Agent-Ansätzen mit Deep RL zur Steuerung großflächiger Ampelnetze [2].
- **Zheng et al. (2019):** Einführung eines Lernverfahrens zur Koordination konkurrierender Phasen bei Ampelsteuerungen mit Hilfe von SUMO [16].

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 mikroskopische Verkehrssimulationen mit SUMO.

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. Die anschließende Konvertierung in das SUMO-Format erfolgte mit dem Programm `netconvert` (Version 1.19.0), einem Teil der SUMO-Toolchain. Hierbei wurden relevante Parameter wie Straßentypen, Fahrspuren, Prioritäten und erlaubte Abbiegevorgänge berücksichtigt. Als Typemap kam `osmNetconvert.typ.xml` 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*). Darüber hinaus konnten 17 signalgesteuerte Kreuzungen mit Lichtsignalanlagen (*traffic lights*) identifiziert werden, die als Steuerungspunkte für das spätere Training der Reinforcement-Learning-Agenten dienen.

Optional wurden zusätzliche Informationen wie Ampeldefinitionen und Vorfahrtsregeln manuell über das Tool `netedit` 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 (eigene Darstellung).

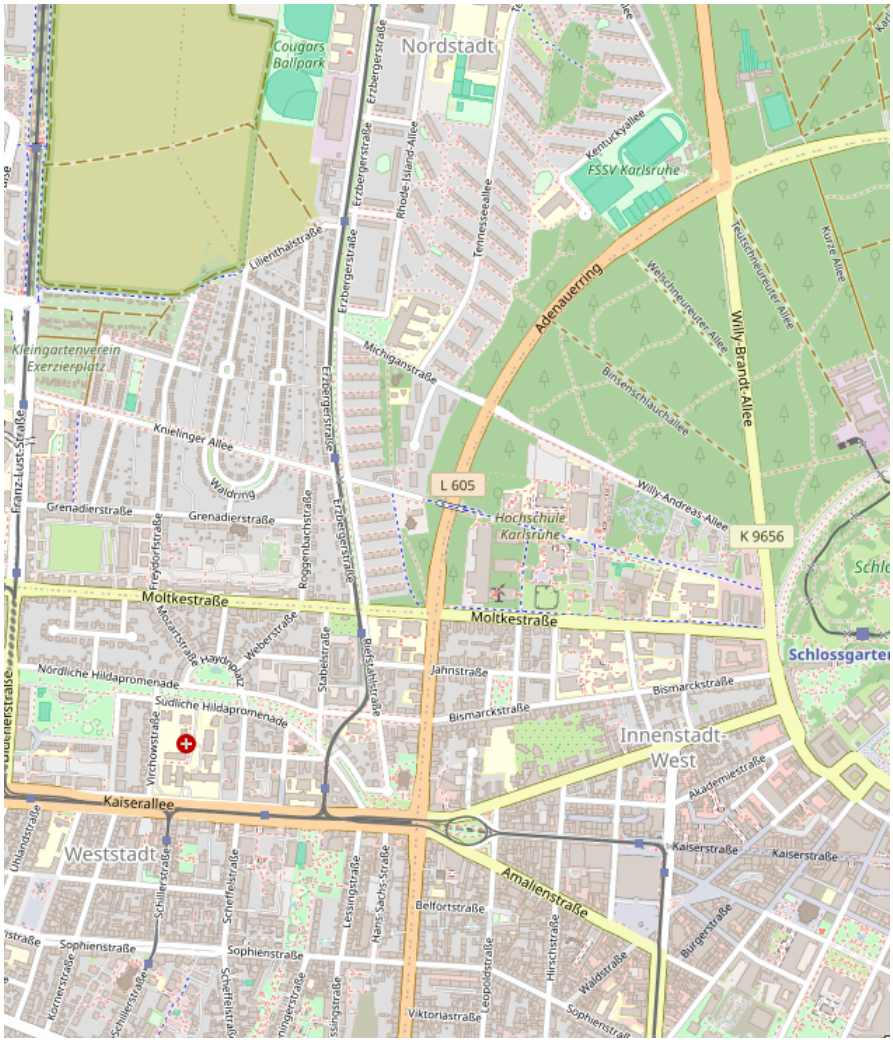


Abbildung 2: Screenshot des ursprünglichen OpenStreetMap-Ausschnitts (eigene Darstellung).

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:[9]

- **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.

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

[1] 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.

Über die Plattform MobiData BW werden offene Mobilitätsdaten gebündelt bereitgestellt, darunter auch historische Detektordaten und OpenTraffic-Feeds. Die Bundesanstalt für Straßenwesen (BASt)[1] wiederum veröffentlicht bundesweite Zähldaten, insbesondere für überörtliche Straßen.

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.

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

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:

- **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 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)

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

Abbildung 3: Lage der Dauerzählstellen im Raum Karlsruhe (Quelle: MobiData BW [6]).

Diese externen Zuflüsse liefern eine verlässliche Grundlage zur Generierung plausibler Eingangsrouten für Fahrzeuge in der Simulation. Das betrachtete Gebiet wird von mehreren dieser Hauptachsen direkt gespeist, wodurch sich ein realitätsnahes Verkehrsaufkommen im Bereich von 20.000–40.000 Fahrzeugen/Tag ergibt – im Einklang mit der Messung in der Reinhold-Frank-Straße.

Die Verwendung dieser Zähldaten ermöglicht es, Fahrzeugströme in SUMO proportional zu real beobachteten Verkehrsverhältnissen zu modellieren. Zudem erleichtert sie die spätere Kalibrierung und Validierung der simulierten Szenarien.

3.2.3 Kommerzielle APIs: TomTom, Google Maps

[3, 14]

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, ermöglicht jedoch eine deutlich feinere zeitliche und räumliche Auflösung, was für die Modellierung und spätere Optimierung des Verkehrsflusses mittels KI von Vorteil ist.

3.3 Modellierung der Ampelschaltungen

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

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.

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 auf eine synthetische Modellierung zurückgegriffen. 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 zumindest eine vereinfachte Lichtsignalsteuerung zu modellieren. Diese erlaubt die spätere Untersuchung von Optimierungsstrategien mittels Reinforcement Learning, stellt jedoch keine Repräsentation der realen Signalprogramme dar.

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:

- **Hohe Verkehrsbedeutung:** Das Gebiet stellt einen wichtigen innerstädtischen Verkehrsraum dar, der sowohl Pendlerverkehr als auch lokalen Individualverkehr aufnimmt.
- **Bekanntes Stauaufkommen:** Die Reinhold-Frank-Straße ist in der Stadtbevölkerung für regelmäßige Verkehrsstaus bekannt, insbesondere zu Stoßzeiten.
- **Verfügbarkeit realer Verkehrszähldaten:** Eine automatische Dauerzählstelle erhebt dort täglich Verkehrsdaten. Für den Zeitraum vom 1.1. bis 20.6.2025 wurden durchschnittlich 21.300 Kfz/Tag erfasst.
- **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.
- **Gute Abgrenzbarkeit:** Das Gebiet ist topologisch geschlossen und in SUMO sauber simulierbar.
- **Verfügbarkeit von Geodaten:** Die Region ist in OpenStreetMap detailliert kartiert.

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 (MobiData BW)[7]. 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 (PKW, lNfz, sNfz). Für den Zeitraum 01.01.–20.06.2025 lag der durchschnittliche Tagesverkehr (DTV[2]) bei ca. 21.300 Kfz/Tag.

[2]DTV = durchschnittlicher Tagesverkehr: Durchschnittliche Anzahl an Fahrzeugen pro Tag über einen bestimmten Zeitraum hinweg.

- **Historische Jahresmittelwerte:** Langzeitdatenreihen von 2008–2024 aus MobiData BW ermöglichen eine Kontextualisierung der aktuellen Verkehrsbelastung.
- **Zählstellen an äußeren Zufahrtsachsen:** Ergänzende Zähldaten aus dem Jahr 2023 an acht stark befahrenen Einfallstraßen (u. a. B10, B36, L605)[6] 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 und Verkehrsflussmodellierung

Zur Modellierung des realen Straßennetzes wurde ein Kartenausschnitt des Untersuchungsgebiets über die Exportfunktion von OpenStreetMap[9] heruntergeladen und anschließend mit JOSM[4] bearbeitet. Die Auswahl des Ausschnitts orientierte sich an der geografischen Abgrenzung rund um die Reinhold-Frank-Straße sowie die angrenzenden Hauptverkehrsachsen im Bereich des Mühlburger Tors. Der bereinigte Kartenausschnitt wurde anschließend mit dem SUMO-Werkzeug `netconvert`[12] in ein netzwerkkompatibles XML-Format überführt. Dabei kamen zusätzliche Optionen zur Verbesserung der Ampelmodellierung und Fahrstreifenzuordnung zum Einsatz (z. B. `-tls.guess-signals` und `-junctions.join`).

Die Erzeugung der Fahrzeugbewegungen erfolgte auf zwei Wegen: Zum einen wurden mit dem SUMO-Skript `randomTrips.py` initiale Testflüsse erzeugt, um die Simulation zu validieren. Zum anderen wurden auf Basis der analysierten Verkehrszähldaten realitätsnahe Flussprofile definiert, welche die beobachteten DTV-Werte auf die Randkanten des simulierten Netzes verteilen. Dabei wurde darauf geachtet, dass die Hauptverkehrsachsen wie die B10 oder B36 als primäre Zufahrtsrouten fungieren und mit einer höheren Fahrzeugdichte gewichtet werden.

Die resultierenden Routendateien wurden anschließend mit `duarouter` verarbeitet, um konfliktfreie Fahrten über das simulierte Netz zu erzeugen. Um unterschiedliche Verkehrssituationen abzubilden, wurden Szenarien mit variierender Verkehrsdichte simuliert – beispielsweise für Morgen- und Abendspitzen sowie für gleichmäßige Durchfahrtsverteilung.

Die erzeugten Flüsse orientieren sich dabei an der realen Kapazität der Knoten und Straßen und wurden mit Hilfe von Detektor-Ausgaben (u. a. `laneAreaDetector`) überprüft und bei Bedarf nachjustiert.

4.2.2 Identifikation relevanter Zufahrtskanten

Die Generierung realistischer Verkehrseinträge in das simulierte Netz basiert auf der systematischen Ermittlung geeigneter Zufahrtskanten. Diese stellen die äußeren Einfallstraßen dar, über die der Verkehr gemäß den Zähldaten in das Untersuchungsgebiet einfließt.

Zur Auswahl wurden zunächst bekannte Hauptverkehrsachsen wie die B10, B36, L605 oder die Durlacher Allee herangezogen. Anschließend erfolgte eine semiautomatische Zuordnung der SUMO-Kanten (`<edge>`) auf Basis der in OpenStreetMap vergebenen Straßennamen. Hierzu wurde ein Python-Skript eingesetzt, das alle Kanten mit einem `name`-Attribut durchsuchte und auf relevante Teilstrings prüfte (z. B. `"B10"`, `"Reinhold-Frank-Straße"`). Die Ergebnisse wurden manuell geprüft und bei Bedarf durch visuelle Kontrolle in `netedit` ergänzt.

Die so extrahierten Kanten wurden je Verkehrsachse gruppiert und bilden die Grundlage für die segmentierte Trip-Erzeugung.

4.2.3 Automatisierte Generierung von Trips auf Basis realer Zählerdaten

Zur Simulation realitätsnaher Verkehrsströme wurden die durchschnittlichen Tagesverkehre (DTV) aus Abschnitt 3.2.2 auf die jeweiligen Zufahrtsgruppen skaliert und anschließend auf die Simulationsdauer von 3600 s verteilt. Ein eigens entwickeltes Python-Skript erzeugte aus diesen Daten Fahrzeugeinträge (`<trip>`), die mithilfe von `randomTrips.py` über die identifizierten Randkanten eingespielt wurden.

Die erzeugten Trips wurden im XML-Format gespeichert und anschließend mit `duarouter` zu vollständigen, konfliktfreien Routen (`<route>`) konvertiert. Die Gesamtanzahl und Verteilung der Fahrzeuge orientierte sich dabei an den stündlich extrapolierten DTV-Werten. Es wurde darauf geachtet, dass insbesondere stark belastete Zufahrten (z. B. B10, L605) mit höherem Gewicht berücksichtigt wurden.

4.2.4 Visuelle und technische Validierung des Netzmodells

Vor dem eigentlichen Einsatz des Modells wurde das gesamte Netz sowohl strukturell als auch funktional validiert. Die Prüfung erfolgte in mehreren Stufen:

- **Netzprüfung:** Einsatz von `netconvert -check-lane-geometry` und `netcheck` zur Überprüfung der topologischen Konsistenz.
- **Visuelle Kontrolle:** Mit der SUMO-GUI sowie `netedit` wurden kritische Knoten visuell inspiziert, um Fehler wie unverbundene Spuren, widersprüchliche Geometrien oder falsche Richtungen zu identifizieren.
- **Manuelle Überprüfung aller TLS:** Jede einzelne Lichtsignalanlage wurde manuell in `netedit` geöffnet. Die Phasenpläne, gesteuerten Verbindungen und Zustandslängen (`state`) wurden dabei geprüft und bei Bedarf korrigiert.

Diese manuelle Validierung war äußerst zeitaufwändig, da fehlerhafte TLS nicht automatisch von SUMO erkannt werden. Die Identifikation von Problemen wie unpassenden `linkIndex`-Werten oder inkonsistenten Zustandslängen erforderte intensives Testen und systematisches Debugging. In mehreren Fällen mussten TLS-Definitionen komplett neu erstellt oder aufgelöst werden, was ein hohes Maß an Modellierungsverständnis und Geduld erforderte.

4.2.5 Szenarien und Referenzsimulationen

Zur Validierung der Verkehrsflüsse wurden unterschiedliche Simulationsszenarien implementiert:

- **Morgenspitze (Rush Hour):** Verstärkte Einträge an den Süd- und Westzufahrten mit hoher Verkehrsdichte.
- **Abendliche Rückstaus:** Stärkere Belastung der Ausfallstraßen und des Innenstadtrings.
- **Gleichmäßiger Tagesverlauf:** Homogene Einträge mit ca. 1000 Fahrzeugen/h pro Richtung.
- **Zentrumsfokus:** Fokus auf Verkehrsströme über die Reinhold-Frank-Straße und das Mühlburger Tor.

Die resultierenden Simulationen dienten der Überprüfung der Netzdurchlässigkeit und der Validierung der physischen Kapazität der Kreuzungen. Dazu wurden Heatmaps, Fahrzeugzählungen sowie Detektor-Ausgaben (z. B. `laneAreaDetector`) ausgewertet. Die gewonnenen Erkenntnisse flossen in die finale Konfiguration der Flüsse und Phasenpläne ein.

4.2.6 Signalsteuerung und Simulationsparameter

Das untersuchte Verkehrsnetz umfasst insgesamt 17 signalgesteuerte Kreuzungen. Diese wurden aus dem OpenStreetMap-Datensatz automatisch erkannt und im Rahmen der Netzkonvertierung mit `netconvert` anhand der Option `-tls.guess-signals` initial als Lichtsignalanlagen (Traffic Light Systems, TLS) modelliert. Die generierten Ampelphasen wurden anschließend manuell überprüft und bei Bedarf über die mit SUMO mitgelieferte GUI `netedit` angepasst.

Für die initiale Simulation wurde eine feste Phasenlogik (fixed-time control) verwendet, um ein Grundverhalten zu etablieren. Dabei erhielten alle Kreuzungen definierte Signalphasen mit festen Umlaufzeiten zwischen 30 und 60 Sekunden, abhängig von der Knotentopologie. Zur Vorbereitung des Reinforcement-Learning-Trainings wurden alle relevanten Kreuzungen so konfiguriert, dass sie in SUMO als „aktuiert“ geschaltet werden konnten. Dies ist erforderlich, damit die Steuerung durch externe Agenten via `TraCI` möglich ist.

Die Simulation wurde mit folgenden Parametern durchgeführt:

- **Simulationszeitraum:** 3600 Sekunden (entspricht 1 Stunde Echtzeit)
- **Zeitschritt (step-length):** 1,0 s
- **Routengenerierung:** deterministisch mit fixer seed zur Reproduzierbarkeit
- **Simulationstyp:** `meso`-Modus zur Beschleunigung des Trainings (später auch `default`-Modus für Evaluation)
- **Verkehrsverteilung:** über `flows.xml` definiert und über Randkanten eingeleitet

Die Definition und Verwaltung der TLS-Systeme erfolgt in separaten Dateien (`*.add.xml`), welche in der SUMO-Konfigurationsdatei eingebunden werden. Für die spätere Anbindung an das Reinforcement-Learning-Modul wurden alle zu steuernden Ampelanlagen mit eindeutigen IDs versehen und überprüft.

4.3 Reinforcement-Learning-Konzept

4.3.1 Formulierung des RL-Problems

Das Problem der Verkehrssteuerung wird als sequentielles Entscheidungsproblem modelliert und mit Hilfe von Reinforcement Learning (RL) gelöst. Ziel ist es, einen Agenten zu trainieren, der durch geeignete Steuerung der Ampelphasen den Verkehrsfluss optimiert. Die Umgebung besteht aus dem simulierten Straßennetz, wie es in SUMO definiert ist. Die Interaktion erfolgt über das TraCI-Interface, das eine Laufzeitsteuerung der Ampelanlagen erlaubt.

Zustände Der Zustand s_t eines Reinforcement-Learning-Agenten beschreibt die Verkehrssituation an einer einzelnen Kreuzung zum Zeitpunkt t. Ziel ist es, dem Agenten ausreichend Informationen über den lokalen Verkehrsfluss zur Verfügung zu

stellen, damit er fundierte Entscheidungen über die Steuerung der Lichtsignalanlage treffen kann.

Die Zustandsrepräsentation basiert auf den folgenden Komponenten:

- **Fahrzeuganzahl pro Zufahrtsspur:** Für jede dem Knoten zuführende Fahrspur wird die aktuelle Anzahl an Fahrzeugen ermittelt. Dies geschieht über sogenannte `laneAreaDetector`, die für jede Spur individuell in SUMO definiert werden. Die Werte werden periodisch über `TraCI` abgefragt.
- **Warteschlangenlänge (queue length):** Gibt die Anzahl der Fahrzeuge an, die sich auf einer Spur mit Geschwindigkeit `< 0.1 m/s` befinden. Dies ist ein wichtiges Maß für Rückstaus an Kreuzungen.
- **Durchschnittliche Geschwindigkeit pro Spur:** Diese Kenngröße erlaubt Rückschlüsse auf den Verkehrsfluss pro Richtung und ergänzt die reine Anzahlinformation.
- **Ampelphase (TLS state):** Die aktuell geschaltete Ampelphase wird als diskrete Phase kodiert (z. B. 0, 1, 2, ...). In SUMO entspricht dies der Index der aktiven Phase im Phasenplan der TLS.
- **Dauer der aktuellen Phase:** Die Anzahl der Zeitschritte seit Beginn der aktuellen Phase. Diese Information ist notwendig, um Phasenlängen sinnvoll zu steuern (z. B. Mindestgrünzeit).
- **Binärmasken zur Phasenwechselbarkeit:** Kodierung, ob ein Wechsel zur nächsten Phase gemäß Übergangsbedingungen (z. B. Mindestgrünzeit) aktuell möglich ist. Diese Information ist erforderlich, falls das Aktionsmodell auch direkte Sprünge zwischen nicht direkt benachbarten Phasen erlaubt.
- **Optional – Nachbarschaftszustand:** In Multi-Agent-Settings kann es sinnvoll sein, zusätzlich aggregierte Zustandsinformationen benachbarter Knoten einzubeziehen (z. B. Gesamtwarteschlange auf ausgehenden Spuren, die zu benachbarten TLS führen).

Die Zustände werden zu einem normierten Merkmalsvektor kombiniert und bilden damit die Eingabe für das neuronale Entscheidungsmodell des Agenten.

Aktionen Die Aktionsmenge A eines Agenten beschreibt die Eingriffsmöglichkeiten in den Steuerungsablauf der jeweiligen Ampelkreuzung. Dabei wird zwischen zwei gängigen Aktionsmodellen unterschieden:

1. **Phasenwechsel-Modell:** Der Agent entscheidet, ob die aktuelle Phase fortgesetzt oder zur nächsten gewechselt werden soll. Es handelt sich um ein binäres Aktionsmodell:
$$A = \{\texttt{keep},\ \texttt{switch}\}$$
Diese Variante wird häufig in klassischen SUMO-RL-Implementierungen verwendet (z. B. 'sumo-rl'). Die Reihenfolge der Phasen ist dabei festgelegt (z. B. zyklischer Übergang).
2. **Direktwahl-Modell:** Der Agent wählt direkt aus allen möglichen Phasen die nächste aus:
$$A = \{\texttt{phase}_0,\ \texttt{phase}_1,\ \ldots,\ \texttt{phase}_n\}$$

Diese Variante erfordert eine eigene Definition der Übergangslogik in SUMO (z. B. über permissive TLS-Ketten), erlaubt aber größere Flexibilität und exploratives Verhalten.

Unabhängig vom Modell gelten folgende Einschränkungen:

- **Mindestgrünzeiten:** Ein Wechsel der Phase darf erst nach einer definierten Mindestgrünzeit erfolgen (z. B. 5 s), um realistische Signalisierung und Verkehrssicherheit zu gewährleisten.
- **Sicherheitsbedingte Zwischenphasen:** SUMO erzwingt automatisch Zwischenphasen wie Gelb- oder Räumzeiten. Der Agent gibt nur den Phasenwunsch an, die exakte Ablaufsteuerung erfolgt durch das TLS-Modell in SUMO.
- **Simultane Agentenentscheidung:** Bei mehreren Knoten wird jeder TLS-Agent unabhängig gesteuert, es sei denn, ein zentrales Multi-Agent-Training wird implementiert.

Zur Reduktion der Aktionsfrequenz wird häufig ein sogenanntes **Action Interval** festgelegt (z. B. alle 5 s), sodass Entscheidungen nur in bestimmten Zeitschritten getroffen werden können. Dies verhindert zu hektisches Umschalten der Ampeln.

Belohnungsfunktion Die Belohnungsfunktion ist zentrales Element des Lernprozesses und bestimmt das Optimierungsziel. Sie wurde so gestaltet, dass sie folgende Aspekte negativ gewichtet:

- **Gesamte Wartezeit aller Fahrzeuge** (minimieren)
- **Länge der Fahrzeugschlangen** (minimieren)
- **Anzahl der Stopps** (minimieren)

Die konkrete Belohnung r_t zum Zeitpunkt t berechnet sich nach:

$$r_t = -\alpha \cdot \sum_{\text{alle Spuren}} \text{queueLength}_i(t) - \beta \cdot \sum_{\text{alle Fahrzeuge}} \text{waitingTime}_j(t)$$

wobei α und β Gewichtungsfaktoren darstellen, die im Rahmen der Hyperparameteroptimierung bestimmt werden. In späteren Varianten kann die Belohnung durch zusätzliche Komponenten wie Emissionen oder Energieverbrauch ergänzt werden, um umweltfreundliche Steuerungsstrategien zu fördern.

4.4 Analyse und Herausforderungen bei der OSM-Netznutzung

4.4.1 Grundstruktur von Lichtsignalanlagen (TLS) in SUMO

Bevor die Probleme beim OSM-Import analysiert werden, ist es hilfreich, den Aufbau und die Abhängigkeiten der relevanten XML-Elemente in SUMO zu verstehen, insbesondere im Zusammenhang mit der Steuerung von Lichtsignalanlagen (*Traffic Light Systems, TLS*).

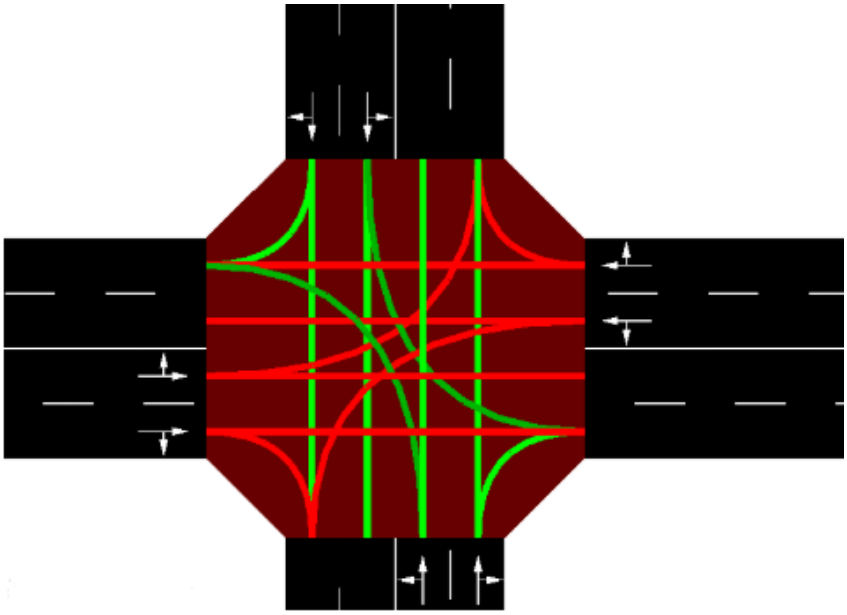


Abbildung 4: Visualisierung einer TLS-Kreuzung (eigene Darstellung).

- **<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 = Grün, 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.

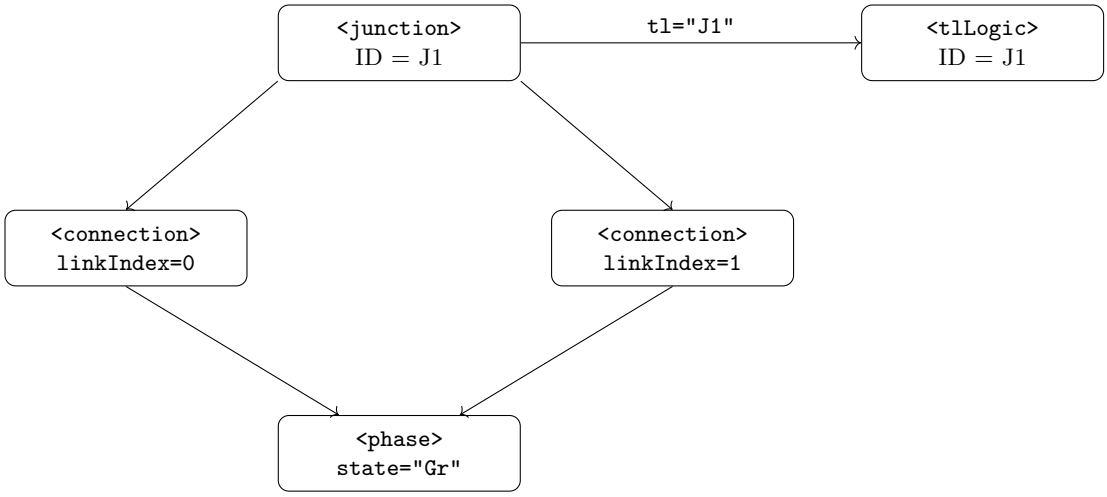


Abbildung 5: Zusammenspiel von Kreuzung, Verbindungen und Ampellogik in SUMO

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`) – inklusive Phasen und Zustände – fehlt vollständig. SUMO generiert daher beim Netzimport mit `-tls.guess-signals` heuristische Ampeldefinitionen, die jedoch oft lückenhaft oder unbrauchbar sind.
- **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.
- **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 – etwa wenn der `state` zu kurz oder zu lang ist. Dies führt in `sumo-rl` zu Indexfehlern oder undefiniertem Verhalten.
- **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 – `sumo-rl` hingegen bricht mit Ausnahmen ab.
- **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`.

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

- 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 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:

- **Separate Fahrspuren für Radverkehr:** Zusätzliche Radstreifen erzeugen neue Kanten mit eigenen Abbiegebeziehungen, die von SUMO automatisch als TLS-relevant eingestuft werden – oft mit übermäßig vielen Signalgruppen.
- **Fußgängerüberwege mit Konfliktzonen:** `highway=crossing`-Elemente erzeugen automatisch Übergänge mit Konfliktzonen, die eine Ampelregelung erfordern – auch 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:

- `-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` 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.

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 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` 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

function CHECKTLSLENGTHS(net.xml)
    Lade XML-Baum und extrahiere `<connection>`-Elemente
    Erstelle Dictionary `tls_controlled_links` mit Anzahl gesteuerter Links pro TLS
    for all `tlLogic`-Elemente im Netz **do**
        `expectedLen` ← Anzahl `controlledLinks` aus Dictionary
        if `expectedLen` = 0 **then**
            Gib Warnung: TLS hat keine gesteuerten Verbindungen
            continue
        end if
        for all Phasen i in `tlLogic` **do**
            `actualLen` ← Länge des `state`-Strings
            if `actualLen` ≠ `expectedLen` **then**
                Gib Warnung mit TLS-ID, Phase und `state`-Inhalt aus
            end if
        end for
    end for
    if keine Abweichungen gefunden **then**
        Gib Erfolgsmeldung aus
    end if
end function

- `check_tls_requests.py` 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

function CHECKTLSREQUESTS(net.xml)
    Lade XML-Datei und parse Wurzelknoten
    Erzeuge Dictionary `tls_signal_indices` mit Signalindizes je TLS aus `<connection>`-Elementen
    for all `junction`-Elemente im Netz **do**
        `tls_id` ← ID der Junction
        if `tls_id` in `tls_signal_indices` **then**
            `expected_max` ← Länge der Signalindizes für dieses TLS
            for all `request`-Elemente in Junction **do**
                `index` ← Wert des `index`-Attributs
                if `index` ≥ `expected_max` **then**
                    Gib Warnung mit `tls_id` und `index` aus
                end if
            end for
        end if
    end for
    if keine Warnungen ausgegeben **then**
        Gib Erfolgsmeldung aus
    end if
end function

Automatische Reparaturwerkzeuge Die folgenden Programme wurden zur strukturellen Korrektur entwickelt:

- `fix_requests.py` 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` 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` 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` 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: `karlsruhe.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`,
- 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 von SUMO, das eine Echtzeitkommunikation zwischen Simulator und RL-Agenten ermöglicht. Die zentrale Steuerung und das Training der Agenten basiert auf der RL-Bibliothek `Stable-Baselines3`, konkret dem Algorithmus `Proximal Policy Optimization (PPO)`.

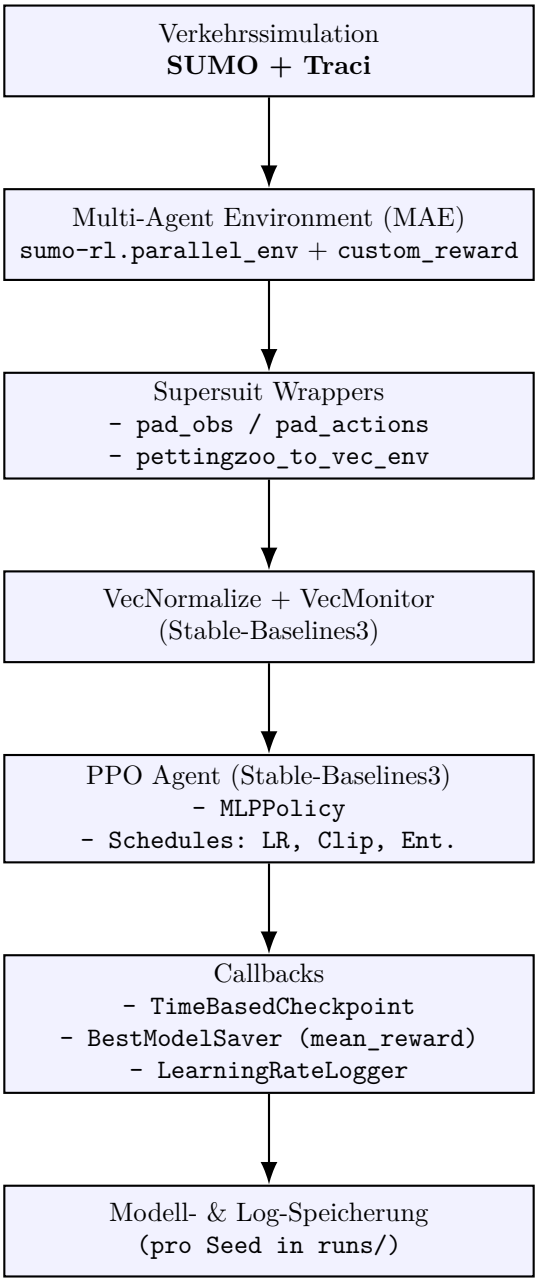


Abbildung 6: Architektur der RL-Trainingspipeline mit SUMO, MAE und Stable-Baselines3

Zur Vereinheitlichung der Multiagentenumgebung kamen die Bibliotheken `PettingZoo` und `SuperSuit` zum Einsatz. `PettingZoo` stellt ein standardisiertes API für Multiagenten-Umgebungen bereit – analog zu `Gymnasium`, 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, wie sie für paralleles Training mit `Stable-Baselines3` erforderlich sind.

4.6.2 Konfiguration der Umgebung

Die Reinforcement-Learning-Umgebung wurde auf Basis der Klasse `SumoEnvironment` bzw. `parallel_env` aus `sumo-rl` konfiguriert. Wichtige Parameter umfassen:

- `net_file`, `route_file`: Pfade zum bereinigten Verkehrsnetz und zugehöriger Routendatei.
- `use_gui`: Aktiviert die grafische Visualisierung von SUMO (zur Laufzeit abschaltbar für Trainingsgeschwindigkeit).
- `num_seconds`: Dauer einer Simulationsepisode in Sekunden.
- `reward_fn`: Referenz auf die benutzerdefinierte Belohnungsfunktion.
- `min_green`: Minimale Grünphasenlänge in Sekunden zur Sicherstellung realistischer Signalzyklen.
- `max_depart_delay`: Maximale Verzögerung bei der Einfahrt eines Fahrzeugs (zur Kontrolle der Spawn-Zeit).
- `sumo_seed`: Zufalls-Seed zur Reproduzierbarkeit von Verkehrsflüssen und Routenentscheidungen.
- `add_system_info`: Wenn aktiviert, werden systemweite Kennzahlen (z. B. durchschnittliche Wartezeit) in die Beobachtung eingebettet.
- `add_per_agent_info`: Ergänzt die lokale Beobachtung jedes Agenten um zusätzliche Kontextdaten (z. B. Verkehrsdichte im Knoten).
- `single_agent = False`: Aktiviert den Multiagentenmodus, bei dem jede steuerbare Ampel einen separaten Agenten darstellt.

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

4.6.3 Beobachtungen und Aktionsraum

Jeder Agent erhält eine lokale Beobachtung, die sich aus folgenden Informationen zusammensetzt:

- aktuelle Ampelphase (diskreter Index oder One-Hot-Encoding),
- Dauer der aktuellen Phase (zur Einhaltung von Mindestzeiten),
- für jede anliegende Spur: Anzahl wartender Fahrzeuge, durchschnittliche Geschwindigkeit, Dichte.

Der Aktionsraum ist diskret und erlaubt die Auswahl der nächsten Phase. Der Phasenwechsel wird durch SUMO automatisch mit einer Zwischenphase (Gelbphase) ergänzt. Die Entscheidung erfolgt synchron für alle Agenten alle `delta_time` Sekunden.

4.6.4 Belohnungsfunktionen

`sumo-rl` unterstützt verschiedene standardisierte Reward-Funktionen:

- `"diff-waiting-time"`: Reduktion der Differenz kumulierter Wartezeiten,
- `"average-speed"`: Maximierung der mittleren Geschwindigkeit im Netz,
- `"queue"`: Minimierung der Gesamtlänge aller Warteschlangen.

Im Rahmen dieser Arbeit wurde zusätzlich eine eigene Reward-Funktion definiert, welche folgende Größen kombiniert:

- aktuelle Warteschlangenlänge (negativ),
- akkumulierte Wartezeiten (negativ),
- Anzahl an Teleportationen und Kollisionen (stark negativ),
- Anzahl neu eingetroffener Fahrzeuge (positiv),
- Veränderung der Stauhöhe zur Vorperiode (positiv).

Die Belohnung wird nach jedem Simulationsschritt einzeln für jeden TLS-Agenten berechnet. Extreme Ereignisse (z. B. viele Teleports) führen zu stark negativen Strafwerten, um stabile Lernverläufe zu fördern.

4.6.5 Trainingsalgorithmus und Hyperparameter

Das Training der Agenten erfolgte mittels `PPO`, wobei folgende Hyperparameter eingesetzt wurden:

- **Policy-Architektur:** Zwei Hidden-Layer mit jeweils 128 Neuronen,
- **Batchgröße:** 2048,
- **Lernrate:** linear abnehmend von $3 \cdot 10^{-4}$,
- **Clip-Range:** dynamisch, linear von 0.2 auf 0.1,
- **Entropiekoeffizient:** 0.005 zur Förderung explorativen Verhaltens,
- **Discount-Faktor:** $\gamma = 0.99$,
- **GAE-Lambda:** 0.95 für stabilisierte Vorteilsschätzung.

Die Umgebung wurde über `VecNormalize` normalisiert und mit `VecMonitor` überwacht. Zusätzlich kamen Wrapper zur Aktion- und Beobachtungsstandardisierung (`pad_observations_v0`, `pad_action_space_v0`) zum Einsatz, um variable TLS-Strukturen zu harmonisieren.

4.6.6 Checkpoints, Monitoring und Logging

Zur Sicherstellung eines robusten Trainingsprozesses wurde eine Reihe von Callback-Mechanismen implementiert:

- **Checkpointing:** Zeitbasierte Sicherung des Modells alle 60 Minuten,
- **Bestmodell-Erkennung:** Automatische Speicherung des jeweils besten Modells (höchste mittlere Reward),
- **Adaptive Schedules:** Dynamische Anpassung von Lernrate und Clip-Range an den Trainingsfortschritt,
- **Logging via TensorBoard:** Visualisierung von Reward-Kurven, Lernraten, Clip-Werten und Modellmetriken.

Alle Modellartefakte (`.zip`, `vecnormalize.pkl`) sowie die TensorBoard-Logs wurden pro Seed-Version strukturiert gespeichert. Dadurch konnten sowohl Reproduzierbarkeit als auch vergleichende Auswertungen zwischen Trainingsläufen gewährleistet werden.

4.6.7 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 Reward-Funktion, adaptiven Trainingsparametern und umfassendem Monitoring wurde eine solide Grundlage für die experimentelle Evaluation lernbasierter Verkehrssteuerung geschaffen.

5 Evaluation und Ergebnisse

5.1 Vergleichsszenarien

5.2 Leistungsmetriken

5.3 Simulationsergebnisse

5.4 Interpretation und Diskussion der Ergebnisse

6 Herausforderungen und Limitationen

6.1 Technische und methodische Hürden

6.2 Repräsentativität und Qualität der Daten

6.3 Generalisierbarkeit der Ergebnisse

7 Fazit und Ausblick

7.1 Zusammenfassung der wichtigsten Erkenntnisse

7.2 Mögliche Weiterentwicklungen

7.3 Relevanz für reale Verkehrsplanung

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

A Trainings-Skripte

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

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

```python
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

# ==== Seeds definieren ====
SEEDS = [1234, 3456, 5678, 7890]  # beliebig erweiterbar

# ==== Custom Reward Function ====
def custom_reward(traffic_signal):
    if not hasattr(traffic_signal, "prev_queue"):
        traffic_signal.prev_queue = traffic_signal.get_total_queued()

    queue = traffic_signal.get_total_queued()
    waiting = np.sum(traffic_signal.get_accumulated_waiting_time_per_lane())

    sim = traci.simulation
    arrived = sim.getArrivedNumber()
    teleport = sim.getStartingTeleportNumber()
    collisions = sim.getCollidingVehiclesNumber()

    delta_queue = traffic_signal.prev_queue - queue
    traffic_signal.prev_queue = queue

    reward = (
        -0.1 * queue
        - 0.05 * waiting
        - 2.0 * teleport
```

```
        - 10.0 * collisions
        + 1.0 * arrived
        + 0.3 * delta_queue
    )

    if teleport > 10 or collisions > 5:
        reward -= 20

    return np.clip(reward, -100, 100)

# ==== 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):
            continue

        final_model = os.path.join(dir_path, "model.zip")
        if os.path.exists(final_model):
            return dir_path, final_model, norm_path

        checkpoint_models = [
            f for f in os.listdir(dir_path)
            if re.match(r"ppo_sumo_model_(\d+)_steps\.zip", f)
        ]
        if checkpoint_models:
            checkpoint_models.sort(key=lambda x: int(re.findall(r"\d+",
            ↪  x)[0]), reverse=True)
            best_checkpoint = checkpoint_models[0]
            return dir_path, os.path.join(dir_path, best_checkpoint),
            ↪  norm_path

    return None

# ==== Adaptive Parameter-Schedules ====
def adaptive_entropy_schedule(start=0.01):
    return lambda progress: max(0.001, start * (1 - progress))

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)
```

```
99          norm_path = os.path.join(dir_path, "vecnormalize.pkl")
100         if not os.path.exists(norm_path):
101             continue
102
103         final_model = os.path.join(dir_path, "model.zip")
104         if os.path.exists(final_model):
105             return dir_path, final_model, norm_path
106
107         checkpoint_models = [
108             f for f in os.listdir(dir_path)
109             if re.match(r"ppo_sumo_model_(\d+)_steps\.zip", f)
110         ]
111         if checkpoint_models:
112             checkpoint_models.sort(key=lambda x: int(re.findall(r"\d+",
                ↪  x)[0]), reverse=True)
113             best_checkpoint = checkpoint_models[0]
114             return dir_path, os.path.join(dir_path, best_checkpoint),
                ↪  norm_path
115
116     return None
117
118 # ==== Checkpoint Callback ====
119 class TimeBasedCheckpointCallback(BaseCallback):
120     def __init__(self, save_interval_sec, save_path,
        ↪  name_prefix="ppo_sumo_model", verbose=0):
121         super().__init__(verbose)
122         self.save_interval_sec = save_interval_sec
123         self.save_path = save_path
124         self.name_prefix = name_prefix
125         self.last_save_time = time.time()
126
127     def _on_step(self) -> bool:
128         return True
129
130     def _on_rollout_end(self) -> bool:
131         current_time = time.time()
132         if current_time - self.last_save_time >= self.save_interval_sec:
133             timestep = self.num_timesteps
134             filename = f"{self.name_prefix}_{timestep}_steps"
135             self.model.save(os.path.join(self.save_path, filename + ".zip"))
136             if hasattr(self.training_env, "save"):
137                 self.training_env.save(os.path.join(self.save_path,
                    ↪  f"{filename}_vecnormalize.pkl"))
138             print(f"[Checkpoint] Modell gespeichert bei {timestep} Schritten
                ↪  ({filename})")
139             self.last_save_time = current_time
140         return True
141
142 # ==== Learning Rate Logger ====
143 class LearningRateLoggerCallback(BaseCallback):
144     def __init__(self, verbose=0):
145         super().__init__(verbose)
146
147     def _on_step(self) -> bool:
148         progress = self.num_timesteps / self.model._total_timesteps
149         lr = self.model.lr_schedule(progress)
150         self.logger.record("train/learning_rate", lr)
```

```
151
152         if hasattr(self.model, 'clip_range'):
153             clip = self.model.clip_range(progress)
154             self.logger.record("train/clip_range", clip)
155
156         return True
157
158 # ==== Best Model Saver Callback ====
159 class BestModelSaverCallback(BaseCallback):
160     def __init__(self, save_path, verbose=0):
161         super().__init__(verbose)
162         self.best_mean_reward = -float('inf')
163         self.save_path = save_path
164
165     def _on_step(self) -> bool:
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             if mean_rew > self.best_mean_reward:
173                 self.best_mean_reward = mean_rew
174                 model_path = os.path.join(self.save_path, "best_model.zip")
175                 self.model.save(model_path)
176                 if hasattr(self.model.env, "save"):
177                     norm_path = os.path.join(self.save_path,
                        ↪  "best_model_vecnormalize.pkl")
178                     self.model.env.save(norm_path)
179                 print(f"[AUTOLOG] Neuer Bestwert {mean_rew:.2f} → best_model
                    ↪  gespeichert!", flush=True)
180
181 # ==== Hauptschleife über Seeds ====
182 for SEED in SEEDS:
183     np.random.seed(SEED)
184     torch.manual_seed(SEED)
185
186     now = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
187     log_dir = os.path.join("runs", f"ppo_sumo_{SEED}_{now}")
188     os.makedirs(log_dir, exist_ok=True)
189
190     print(f"\n[INFO] Starte Training mit Seed: {SEED}")
191
192     env = parallel_env(
193         net_file="map.net.xml",
194         route_file="map.rou.xml",
195         use_gui=False,
196         num_seconds=1000,
197         reward_fn=custom_reward,
198         min_green=5,
199         max_depart_delay=100,
200         sumo_seed=SEED,
201         add_system_info=True,
202         add_per_agent_info=True,
203     )
204
205     if hasattr(env, "seed"):
```

```
206             env.seed(SEED)
207
208         env = pad_observations_v0(env)
209         env = pad_action_space_v0(env)
210         env = pettingzoo_env_to_vec_env_v1(env)
211         env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=8,
            ↪  base_class="stable_baselines3")
212         env = VecMonitor(env)
213         env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.0)
214
215         model = PPO(
216             policy="MlpPolicy",
217             env=env,
218             verbose=1,
219             tensorboard_log=log_dir,
220             batch_size=2048,
221             n_steps=2048,
222             learning_rate=linear_schedule(3e-4),
223             clip_range=dynamic_clip_range(0.2),
224             ent_coef=0.005,
225             gamma=0.99,
226             gae_lambda=0.95,
227             device="cpu",
228             policy_kwargs=dict(net_arch=dict(pi=[128, 128], vf=[128, 128])),
229         )
230
231         callbacks = CallbackList([
232             TimeBasedCheckpointCallback(
233                 save_interval_sec=3600,
234                 save_path=log_dir,
235                 name_prefix="ppo_sumo_model",
236                 verbose=1,
237             ),
238             LearningRateLoggerCallback(),
239             BestModelSaverCallback(save_path=log_dir),
240         ])
241
242         try:
243             model.learn(
244                 total_timesteps=1_500_000,
245                 callback=callbacks,
246             )
247             model.save(os.path.join(log_dir, "model.zip"))
248             env.save(os.path.join(log_dir, "vecnormalize.pkl"))
249             print(f"\n[INFO] Training abgeschlossen für Seed {SEED}. Modell
                ↪  gespeichert unter: {log_dir}")
250
251         except KeyboardInterrupt:
252             print("[ABBRUCH] Manuelles Beenden erkannt. Speichere aktuellen
                ↪  Stand...")
253             model.save(os.path.join(log_dir, "model_interrupt.zip"))
254             env.save(os.path.join(log_dir, "vecnormalize_interrupt.pkl"))
255
256         except Exception as e:
257             print(f"\n[FEHLER] Während des Trainings bei Seed {SEED} aufgetreten:
                ↪  {e}")
258
```

```
    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)

# ==== Custom Reward Function ====
def custom_reward(traffic_signal):
    if not hasattr(traffic_signal, "prev_queue"):
        traffic_signal.prev_queue = traffic_signal.get_total_queued()

    queue = traffic_signal.get_total_queued()
    waiting = np.sum(traffic_signal.get_accumulated_waiting_time_per_lane())

    sim = traci.simulation
    arrived = sim.getArrivedNumber()
    teleport = sim.getStartingTeleportNumber()
    collisions = sim.getCollidingVehiclesNumber()

    delta_queue = traffic_signal.prev_queue - queue
    traffic_signal.prev_queue = queue

    reward = (
```

```
        -0.1 * queue
        - 0.05 * waiting
        - 2.0 * teleport
        - 10.0 * collisions
        + 1.0 * arrived
        + 0.3 * delta_queue
    )

    if teleport > 10 or collisions > 5:
        reward -= 20

    return np.clip(reward, -100, 100)

# ==== 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):
            continue

        final_model = os.path.join(dir_path, "model.zip")
        if os.path.exists(final_model):
            return dir_path, final_model, norm_path

        checkpoint_models = [
            f for f in os.listdir(dir_path)
            if re.match(r"ppo_sumo_model_(\d+)_steps\.zip", f)
        ]
        if checkpoint_models:
            checkpoint_models.sort(key=lambda x: int(re.findall(r"\d+",
            ↪  x)[0]), reverse=True)
            best_checkpoint = checkpoint_models[0]
            return dir_path, os.path.join(dir_path, best_checkpoint),
            ↪  norm_path

    return None

# ==== Zeitbasierter Checkpoint Callback ====
class TimeBasedCheckpointCallback(BaseCallback):
    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
```

```
97          self.last_save_time = time.time()
98
99      def _on_step(self) -> bool:
100         return True
101
102     def _on_rollout_end(self) -> bool:
103         current_time = time.time()
104         if current_time - self.last_save_time >= self.save_interval_sec:
105             timestep = self.num_timesteps
106             filename = f"{self.name_prefix}_{timestep}_steps"
107             self.model.save(os.path.join(self.save_path, filename + ".zip"))
108             if hasattr(self.training_env, "save"):
109                 self.training_env.save(os.path.join(self.save_path,
                    ↪  f"{filename}_vecnormalize.pkl"))
110             print(f"[Checkpoint] Modell gespeichert bei {timestep} Schritten
                ↪  ({filename})")
111             self.last_save_time = current_time
112         return True
113
114 # ==== Learning Rate Logger Callback ====
115 class LearningRateLoggerCallback(BaseCallback):
116     def __init__(self, verbose=0):
117         super().__init__(verbose)
118
119     def _on_step(self) -> bool:
120         lr = self.model.lr_schedule(self.num_timesteps /
            ↪  self.model._total_timesteps)
121         self.logger.record("train/learning_rate", lr)
122         return True
123
124 # ==== Logging ====
125 now = datetime.datetime.now().strftime("%Y-%m-%d_%H-%M-%S")
126 log_dir = os.path.join("runs", f"ppo_sumo_{now}")
127 os.makedirs(log_dir, exist_ok=True)
128
129 # ==== SUMO-RL Umgebung ====
130 env = parallel_env(
131     net_file="map.net.xml",
132     route_file="map.rou.xml",
133     use_gui=False,
134     num_seconds=1000,
135     reward_fn=custom_reward,
136     min_green=5,
137     max_depart_delay=100,
138     sumo_seed=SEED,
139     add_system_info=True,
140     add_per_agent_info=True,
141 )
142
143 if hasattr(env, "seed"):
144     env.seed(SEED)
145
146 # ==== Wrapping ====
147 env = pad_observations_v0(env)
148 env = pad_action_space_v0(env)
149 env = pettingzoo_env_to_vec_env_v1(env)
```

```
150 env = concat_vec_envs_v1(env, num_vec_envs=1, num_cpus=8,
    ↪ base_class="stable_baselines3")
151 env = VecMonitor(env)
152
153 # ==== Modell laden oder neu starten ====
154 result = find_latest_complete_run()
155 if result:
156     latest_run_dir, model_path, normalize_path = result
157     print("Fortsetzung wird gestartet mit:")
158     print(f"Verzeichnis : {latest_run_dir}")
159     print(f"Modell      : {model_path}")
160     print(f"Normalize   : {normalize_path}\n")
161
162     env = VecNormalize.load(normalize_path, env)
163     env.training = True
164     env.norm_reward = True
165
166     model = PPO.load(model_path, env=env, tensorboard_log=log_dir, verbose=1,
        ↪ device="cpu")
167     print(f"[INFO] Modell startet bei {model.num_timesteps} Timesteps.")
168 else:
169     print("[INFO] Kein vorheriges Modell gefunden. Starte frisches
        ↪ Training.\n")
170     env = VecNormalize(env, norm_obs=True, norm_reward=True, clip_obs=10.0)
171     model = PPO(
172         policy="MlpPolicy",
173         env=env,
174         verbose=1,
175         tensorboard_log=log_dir,
176         batch_size=2048,
177         n_steps=2048,
178         learning_rate=linear_schedule(3e-4),
179         clip_range=dynamic_clip_range(0.2),
180         ent_coef=0.005,
181         gamma=0.99,
182         gae_lambda=0.95,
183         device="cpu",
184         policy_kwargs=dict(net_arch=dict(pi=[128, 128], vf=[128, 128])),
185     )
186
187 # ==== Automatisches Speichern bei verbessertem ep_rew_mean ====
188 class BestModelSaverCallback(BaseCallback):
189     def __init__(self, save_path, verbose=0):
190         super().__init__(verbose)
191         self.best_mean_reward = -float('inf')
192         self.save_path = save_path
193
194     def _on_step(self) -> bool:
195         # Muss vorhanden sein, selbst wenn sie nichts tut
196         return True
197
198     def _on_rollout_end(self):
199         ep_info_buffer = self.model.ep_info_buffer
200         if len(ep_info_buffer) > 0:
201             mean_rew = np.mean([ep_info['r'] for ep_info in ep_info_buffer])
202
203             if mean_rew > self.best_mean_reward:
```

```
                self.best_mean_reward = mean_rew

                model_path = os.path.join(self.save_path, "best_model.zip")
                self.model.save(model_path)

                if hasattr(self.model.env, "save"):
                    norm_path = os.path.join(self.save_path,
                    ↪  "best_model_vecnormalize.pkl")
                    self.model.env.save(norm_path)

                print(f"[AUTOLOG] Neuer Bestwert {mean_rew:.2f} → best_model
                ↪  gespeichert!", flush=True)

# ==== Callbacks kombinieren ====
callbacks = CallbackList([
    TimeBasedCheckpointCallback(
        save_interval_sec=300,
        save_path=log_dir,
        name_prefix="ppo_sumo_model",
        verbose=1,
    ),
    LearningRateLoggerCallback(),
    BestModelSaverCallback(save_path=log_dir),
])

# ==== Training starten ====
try:
    model.learn(
        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 Netzwerk-Skripte

B.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.

```python
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()
```

B.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.

```python
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!")
```

B.3 `fix_requests.py` – Automatische Korrektur von Requests und Phasen

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

```python
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}")
```

B.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.")

```

B.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)}")
```

B.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)
```

Literatur

[1] Bundesanstalt für Straßenwesen (BASt). https://www.bast.de, Zugriff am 21.05.2025.

[2] Chu, T.; Wang, J.; He, R.; Xia, Y. (2020): *Multi-Agent Deep Reinforcement Learning for Large-Scale Traffic Signal Control*. In: IEEE Transactions on Intelligent Transportation Systems. DOI: https://doi.org/10.1109/TITS.2020.3014863

[3] Google Maps Traffic API. , Zugriff am 21.05.2025.

[4] JOSM – Java OpenStreetMap Editor. https://josm.openstreetmap.de, Zugriff am 21.05.2025.

[5] MobiData BW – Mobilitätsdatenplattform Baden-Württemberg. https://www.mobidata-bw.de, Zugriff am 21.05.2025.

[6] MobiData BW: Karte der Dauerzählstellen im Straßenverkehr, https://mobidata-bw.de/dataset/karte_strassenverkehrszaehlung, Zugriff am 21.05.2025.

[7] MobiData BW: Stundenwerte an Dauerzählstellen (Straßenverkehr), https://mobidata-bw.de/dataset/stundenwerte_dauerzaehlstellen, Zugriff am 21.05.2025.

[8] OpenStreetMap. https://www.openstreetmap.org, Zugriff am 21.05.2025.

[9] OpenStreetMap Wiki: Export. https://wiki.openstreetmap.org/w/index.php?title=Export&oldid=2822860, Zugriff am 21.06.2025.

[10] SUMO Dokumentation. https://sumo.dlr.de, Zugriff am 21.05.2025.

[11] sumo-rl: Reinforcement Learning Environments for Traffic Signal Control in SUMO. GitHub Repository. https://github.com/LucasAlegre/sumo-rl, Zugriff am 21.05.2025.

[12] SUMO Toolchain: Netzwerk- und Routengeneratoren. https://sumo.dlr.de/docs/Tools.html, Zugriff am 21.05.2025.

[13] Straßenverkehrszentrale Baden-Württemberg. https://www.svz-bw.de, Zugriff am 21.05.2025.

[14] TomTom Traffic API. https://developer.tomtom.com/traffic-api, Zugriff am 21.05.2025.

[15] Wei, H.; Zheng, G.; Yao, H.; Li, Z. (2019): *A Deep Reinforcement Learning Approach for Traffic Signal Control in Vehicular Networks*. In: Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining, S. 2496–2505. DOI: https://doi.org/10.1145/3292500.3330662

[16] Zheng, G.; Ye, H.; Zhang, H.; Wu, K.; Li, Z. (2019): *Learning Phase Competition for Traffic Signal Control*. arXiv preprint. https://arxiv.org/abs/1907.09076

[17] *Umweltbundesamt* https://www.umweltbundesamt.de/daten/private-haushalte-konsum/mobilitaet-privater-haushalte#-hoher-motorisierungsgrad https://www.umweltbundesamt.de/daten/verkehr/emissionen-des-verkehrs#pkw-fahren-heute-klima-und-umweltvertraglicher https://www.umweltbundesamt.de/themen/verkehr/klimaschutz-im-verkehr#bepreisung Zugriff am 21.05.2025.

[18] *Baden-wuerttemberg* https://www.baden-wuerttemberg.de/de/service/presse/pressemitteilung/pid/land-startet-testfeld-mit-ki-gesteuerten-ampeln Zugriff am 21.05.2025.

[19] *Bundesanstallt für Straßenwesen* https://edocs.tib.eu/files/e01fn19/166939879X.pdf