Flow: Deep Reinforcement Learning for Control in SUMO

Nishant Kheterpal[1], Kanaad Parvate[1], Cathy Wu[1], Aboudy Kreidieh[2], Eugene Vinitsky[3], and Alexandre M Bayen[124]

[1] UC Berkeley, Electrical Engineering and Computer Science
[2] UC Berkeley, Department of Civil and Environmental Engineering
[3] UC Berkeley, Department of Mechanical Engineering
[4] UC Berkeley, Institute for Transportation Studies

Abstract

We detail the motivation and design decisions underpinning Flow, a computational framework integrating SUMO with the deep reinforcement learning libraries rllab and RLlib, allowing researchers to apply deep reinforcement learning (RL) methods to traffic scenarios, permitting vehicle and infrastructure control in highly varied traffic environments. Users of Flow can rapidly design a wide variety of traffic scenarios in SUMO, enabling the development of controllers for autonomous vehicles and intelligent infrastructure across a broad range of settings.

Flow facilitates the use of policy optimization algorithms to train controllers that can optimize for highly customizable traffic metrics, such as traffic flow or system-wide average velocity. Training reinforcement learning agents using such methods requires a massive amount of data, thus simulator reliability and scalability were major challenges in the development of Flow. A contribution of this work is a variety of practical techniques for overcoming such challenges with SUMO, including parallelizing policy rollouts, smart exception and collision handling, and leveraging subscriptions to reduce computational overhead.

To demonstrate the resulting performance and reliability of Flow, we introduce the canonical single-lane ring road benchmark and briefly discuss prior work regarding that task. We then pose a more complex and challenging multi-lane setting and present a trained controller for a single vehicle that stabilizes the system. Flow is an open-source tool and available online at https://github.com/cathywu/flow.

1 Introduction

During 2017 in the United States, economic loss due to traffic congestion in urban areas is estimated at \$305 billion [1] with the average commuter spending upwards of 60 hours in traffic every year [2]. In 2017, commuters in Los Angeles, a city notorious for its congestion, spent on average over 100 hours per year stuck in traffic [1]. Additionally, estimates warn that the fraction of fuel usage wasted in congestion will near 2.6% in 2020 and rise to 4.2% by 2050 [3]. Clearly, improving urban congestion has both environmental and economic impacts. As autonomous vehicles approach market availability, we see even more opportunity to develop and implement traffic-mitigating strategies, via both vehicle-level controllers and fleet-level cooperative methods.

Researchers have experimented with various techniques to reduce the congestion present in a system, such as platooning [4], intelligent vehicle spacing control to avoid infrastructure

Corresponding author: Nishant Kheterpal (nskh@berkeley.edu)
{kanaad, cathywu, aboudy, evinitsky, bayen}@berkeley.edu

bottlenecks [5], intelligent traffic lights [6], and hand-designed controllers to mitigate stop-and-go waves [7]. Analyzing hand-designed controllers can be limited by model complexity, thereby deviating from real-world considerations.

Reinforcement learning is both a natural and broadly applicable way to approach decision problems—the use of trial-and-error to determine which actions lead to better outcomes is easily fit to problems in which one or more learning agents wish to optimize an outcome in their environment. Furthermore, recent advances in algorithms and hardware have made deep reinforcement learning methods tractable for a variety of applications, especially in domains in which high-fidelity simulators are available. These methods are performant enough to apply to and perform well in scenarios for which it may be difficult to design hand-designed controllers, such as synthesizing video game controllers from raw pixel inputs [8], continuous control for motion planning [9], robotics [10], and traffic [11, 12]. Though end-to-end machine learning solutions are rarely implemented as-is due to challenges with out-of-distribution scenarios, these solutions sometimes yield unexpected solutions when compared to classical approaches. Such solutions can inspire controllers emulating desirable properties of the trained approach, such as stability, robustness, and more.

Flow [12] is an open-source framework for constructing and solving deep reinforcement problems in traffic that leverages the open-source microsimulator, SUMO [13]. Through Flow, users can use deep reinforcement learning to develop controllers for a number of intelligent systems, such as autonomous vehicles or intelligent traffic lights. In this paper, we detail the design decisions behind Flow, as motivated by the challenges of tractably using deep RL techniques with SUMO. We present the architectural decisions in terms of the steps of conducting an experiment with Flow: 1) designing a traffic control task with SUMO, 2) training the controller, and 3) evaluating the effectiveness of the controller. Finally, we demonstrate and analyze Flow's effectiveness in a setting for which determining analytically optimal controllers might be intractable: optimizing system-level speed of mixed-autonomy traffic for a multi-lane ring road. Flow is available at `https://github.com/cathywu/flow/` for development and experimentation use by the general public.

2 Related Works

Reinforcement Learning Frameworks: Virtual environments in which intelligent agents can be implemented and evaluated are for are essential to the development of artificial intelligence techniques. Current state-of-the-art research in deep RL relies heavily on being able to design and simulate virtual scenarios. A number of such platforms exist; two significant ones are the Arcade Learning Environment (ALE) [14] and MuJoCo (Multi-Joint dynamics with Contact) [15]. ALE emulates Atari 2600 game environments to support the training and evaluation of RL agents in challenging—for humans and computers alike—and diverse settings [14]. Schaul, Togelius, and Schmidhuber discussed the potential of games to act as evaluation platforms for general intelligent agents and describe the body of problems for AI made up by modern computer games in [16]. MuJoCo is a platform for testing model-based control strategies and supports many models, flexible usage by users, and multiple methods of accessing its functionality [15]. [17] and [18] use MuJoCo to evaluate agent performance. Box2D is another physics engine [19], written in C++ and used in [17] to evaluate simple agents. The 7th International Planning Competition concerned benchmarks for planning agents, some of which could be used in RL settings [20]. These frameworks are built to enable the training and evaluation of reinforcement learning models by exposing an application programming interface (API). Flow is designed to be another such platform, specifically built for applying reinforcement learning to scenarios built in traffic microsimulators.

Deep RL and Traffic: Recently, deep learning and deep reinforcement learning in particular have been applied to traffic settings. CARLA is a recently developed driving simulator supported as a training environment in RLlib [21]. However, CARLA is in an early development stage, and is a 3D simulator used mostly for autonomous vehicle testing. Lv et al. and Polson & Sokolov predicted traffic flow using deep learning [22, 23]; however, neither used any sort of simulator. Deep RL has been used for traffic control as well—the work of [11] concerned ramp metering, [24] speed limit-based control; however, both used macroscopic simulation based on PDEs. Applications of Flow to mixed-autonomy traffic are described in [5, 12].

3 Preliminaries

In this section, we introduce two theoretical concepts important to Flow: reinforcement learning and vehicle dynamics models. Additionally, we provide an overview of the framework.

3.1 Reinforcement Learning

The problem of reinforcement learning centers around training an agent to interact with its environment and maximize a *reward* accrued through this interaction. At any given time, the environment has a *state* (e.g. the positions and velocities of vehicles within a road network) which the RL agent may modify through certain *actions* (e.g. an acceleration or lane change). The agent must learn to recognize which states and actions yield strong rewards through *exploration*, the process of performing actions that lead to new states, and *exploitation*, performing actions that yield high reward.

An RL problem is formally characterized by a Markov Decision Process (MDP), denoted by the tuple $\mathcal{M} = (\mathcal{S}, \mathcal{A}, P, r, \rho_0, \gamma, T)$ [25, 26]. Denote $\mathcal{S}$ a set of states constituting the state space, which may be discrete or continuous, finite- or infinite-dimensional; $\mathcal{A}$ a set of possible actions constituting the *action space*; P the transition probability distribution, a function mapping $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_{[0,1]}$; r a reward function, mapping states in $\mathcal{S}$ and actions in $\mathcal{A}$ to rewards $\mathbb{R}$; ρ_0 the initial probability distribution over states ($\mathcal{S} \to \mathbb{R}_{[0,1]}$); γ the discount factor on accrued rewards in the interval $(0, 1]$; T the time horizon. Partially Observable Markov Decision Processes are MDPs characterized additionally by *observation space* Ω and observation probability distribution $\mathcal{O} : \mathcal{S} \times \Omega \to \mathcal{R}_{[0,1]}$.

Though there are a plethora of approaches to solving RL problems, *policy optimization* methods are well-suited to our problem domain as they provide an effective way to solve continuous control problems. In policy optimization methods, actions are drawn from a probability distribution generated by a *policy*: $a_t \sim \pi_\theta(a_t|s_t)$. The policy is represented by a set of parameters θ that encode a function. In the context of automotive applications, "policy" is simply the reinforcement learning term for a controller. A policy might control a vehicle's motion, the color of a traffic light, etc. The policy is optimized directly with a goal of maximizing the cumulative discounted reward:

$$\eta(\pi) = \mathbb{E}_{s_0,a_0\ldots}[\sum_{t=0}^{T} \gamma^t r(s_t)]$$

where $s_0 \sim \rho$. $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ and $a_t \sim \pi_\theta(a_t|s_t)$ are the state and action, respectively, at time t. Often, observations in the observation space Ω are passed to the agent and used by the policy in place of states s_t. Policy optimization algorithms generally allow the user to choose the type of function the policy represents, and optimize it directly with respect to the reward.

Policies in deep reinforcement learning are generally encoded by neural networks, hence the use of the term "deep" [27, 28]. *Policy gradient* algorithms are a subclass of policy optimization methods that seek to estimate the gradient of the expected discounted return with respect to the parameters, $\nabla_\theta \eta(\theta)$, and iteratively update the parameters θ via gradient descent [29, 30].

The policies used by Flow are usually multi-layered or recurrent neural networks that output diagonal Gaussian distributions. The actions are stochastic in order to both facilitate exploration of the state space and to enable simplified computation of the gradient via the "log derivative trick" [31]. We use several popular policy gradient algorithms including Trust Region Policy Optimization (TRPO) [32] and Proximal Policy Optimization (PPO) [33]. In order to estimate the gradient, these algorithms require samples consisting of (*observation, reward*) pairs. To accumulate samples, we must be able to *rollout* the policy for T timesteps. Each iteration, samples are aggregated from multiple rollouts into a *batch* and the resulting gradient is used to update the policy.

This process of performing rollouts to collect batches of samples followed by updating the policy is repeated until the average cumulative reward has stabilized, at which point we say that training has *converged*.

3.2 Vehicle Dynamics

A brief description of longitudinal and lateral driving models follows.

Longitudinal Dynamics Car-following models are the primary method of defining longitudinal dynamics [34]. Most car-following models follow the format

$$\ddot{x_i} = f(h_i, \dot{h_i}, \dot{x_i}) \tag{1}$$

where x_i is the position of vehicle i, vehicle $i-1$ is the vehicle ahead of vehicle i, and $h_i := x_i - x_{i-1}$ is the headway of vehicle i. Car-following models may also include time delays in some or all of these terms to account for lag in human perception [34].

Lateral Dynamics Lateral vehicle dynamics, unlike longitudinal dynamics, can be modeled as discrete events [35]. Such events include lane-change decisions (whether to move left, move right, or remain within one's lane) or merge decisions (whether to accelerate and merge into traffic or wait for a larger gap in traffic). Treiber's *Traffic Flow Dynamics* states that drivers choose between these discrete actions to maximize their utility subject to safety constraints [35]. Notions of utility might consider traffic rules and driver behavior.

3.3 Overview of Flow

Flow [12] is an open-source Python framework that seeks to provide an accessible way to solve vehicle and traffic control problems using deep reinforcement learning. Using Flow, users can easily create an *environment* to encapsulate an MDP that defines a certain RL problem. The environment is a Python class that provides an interface to initialize, reset and advance the simulation, as well as methods for collecting observations, applying actions and aggregating reward.

The environment provides an algorithm with all the necessary information—observations, actions, rewards—to learn a policy. In our work we have used RL algorithms implemented in two different libraries: Berkeley RLL's rllab [17] and RISELab's RLlib [36]; additionally, Flow environments are compatible with OpenAI's Gym, an open-source collection of benchmark problems [37]. rllab and RLlib are both open-source libraries supporting the implementation, training, and evaluation of reinforcement learning algorithms. RLlib in particular supports distributed computation. The libraries include a number of built-in training algorithms such as the policy gradient algorithms TRPO [32] and PPO [33], both of which are used in Flow.

We use SUMO, an open-source, microscopic traffic simulator, for its ability to handle large, complicated road networks at a microscopic (vehicle-level) scale, as well as easily query, control, or otherwise extend the simulation through TraCI [13, 38]. As a microscopic simulator, SUMO provides several car-following and lane-change models to dictate the longitudinal and lateral dynamics of individual vehicles [39, 40, 41].

Flow seeks to directly improve certain aspects about SUMO in order to make it more suitable for deep reinforcement learning tasks. SUMO's car following models are all configured with a minimal time headway, τ, for safety [42]. As a result, time delayed models are more difficult to implement natively. Secondly, SUMO's models may experience issues for a timestep shorter than 1.0 seconds, which poses issues for deep reinforcement learning algorithms, which depend on high-fidelity simulators [43]. Development of new vehicle dynamics models—longitudinal and lateral alike—is cumbersome, and must be done in C++.

To address these issues, Flow provides users with the ability to easily implement, through TraCI's Python API, hand-designed controllers for any components of the traffic environment such as calibrated models of human dynamics or smart traffic light controllers. Together with the dynamics built into SUMO, Flow allows users to design rich environments with complex dynamics.

A central focus in the design of Flow was the ease of modifying road networks, vehicle characteristics, and infrastructure within an experiment, along with an emphasis on enabling reinforcement learning control over not just vehicles, but traffic infrastructure as well. Flow enables the user to programmatically modify road networks used in experiments, which allows the training of policies across road networks of varied size, density, number of lanes, and more. Additionally, RL observation spaces and reward functions can be easily constructed from attributes of the environment. Once trained, policies can be evaluated in scenarios different from those in which they were trained, making performance evaluation straightforward.

Users of Flow can algorithmically generate roads of a number of types. At the moment, circular ring roads, figure-eight roads, merge networks, and intersection grids are supported. Ring roads (shown in Figure 1 at top middle) are defined by their length and the number of lanes, and have been used to study congestion and stabilization methods [7, 44]. Figure-eight roads (shown in Figure 1 at bottom left) are defined by their loop radius. Merge networks (shown in Figure 1 at top left) consist of vehicles traveling in an inner loop and an outer loop, which are connected in two points to create merge dynamics. They are defined by their loop radii. Intersection grids (shown in Figure 1 at top right) have numerous intersecting roads to test traffic-light patterns and other schemes and can be grids of arbitrary dimension.

Flow also supports the import of OpenStreetMap networks, an example of which is shown in Figure 1 at bottom middle. This enables autonomous vehicles and traffic-management policies to be tested on complex road networks quickly without needing to specify scenario parameters or design a road network like the merge network or figure-eight. High-performing policies can also be trained on challenging features from real-world road networks, like complex interchanges, merges, curves, and more. This flexibility is essential in enabling broad use cases for Flow; the framework makes it simple to evaluate traffic dynamics—given different car-following models, vehicle types, lane-change controllers, speed limits, etc.—on any network specified by OSM data.

Vehicle traffic in a Flow experiment can be specified arbitrarily, either by an initial set

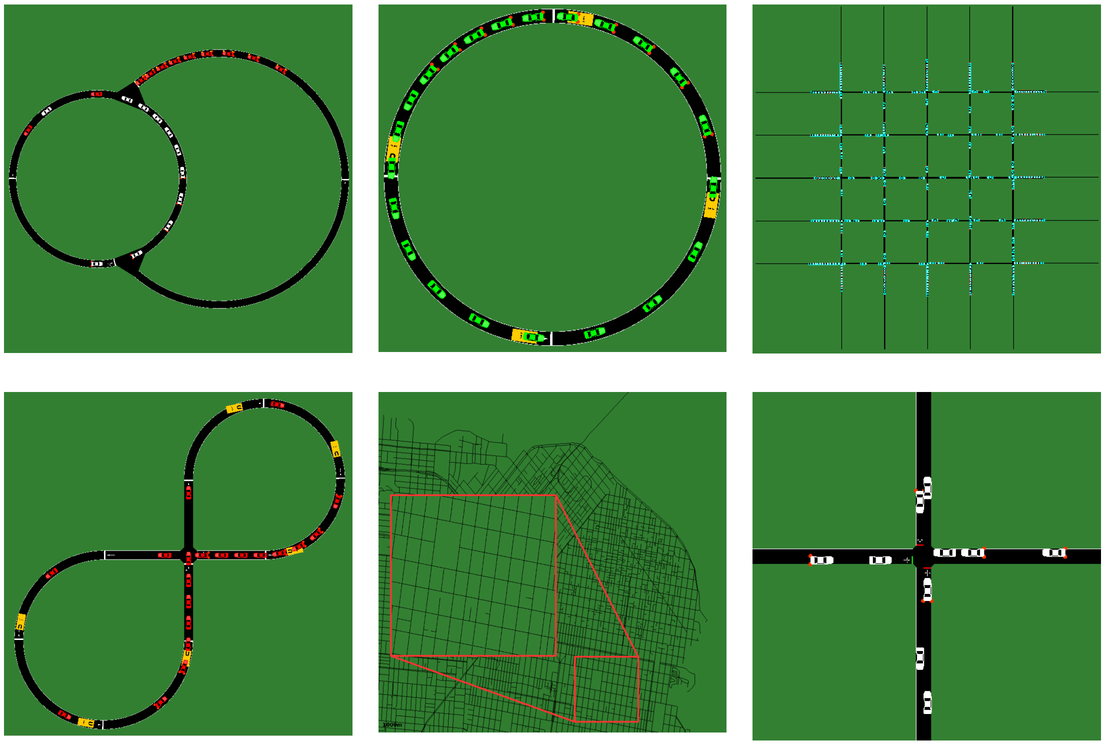


Figure 1: Networks supported in Flow. **Clockwise from top left**: merge network with two loops merging; single-lane ring; intersection grid network; close-up of intersections in grid; roads in downtown San Francisco, imported from OpenStreetMap; figure-eight scenario

of vehicles in the network or using SUMO vehicle inflows. In our work using Flow, we often reference existing experiments such as [44] and [45] to dictate road network demand. Flow's extensibility also enables the measurement of network characteristics like inflow and outflow, allowing the number of vehicles in the network to be set to, say, just above the critical density for a road network. This network density can come from the initial vehicles on the road, for closed networks, or in the form of inflows for open networks in which vehicles enter and leave the network.

Flow supports SUMO's built-in longitudinal and lateral controllers and includes a number of configurable car-following models. Arbitrary acceleration-based custom car-following models are supported as well. The implementation details of Flow's longitudinal and lateral controllers are described further in subsection 5.1. In an experiment, multiple types of vehicles—defined by the dynamics models they obey—can be added. Arbitrary numbers of vehicle of each type are supported. In this way, Flow enables the straightforward use of diverse vehicle behavior and configurations in SUMO and provides fully-functional environments for reinforcement learning.

4 Architecture of Flow

In Figure 2, we map out the interactions between the various components of SUMO. The **scenario** holds static information about the network, including attributes about the shape methods

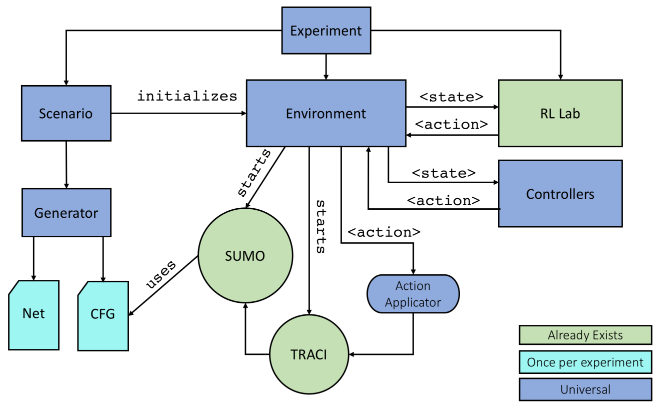


Figure 2: Flow provides components to programatically generate networks and configuration, as well as an interface with reinforcement learning libraries. Flow is designed to be both modular and extensible.

defining absolute positions, methods for generation of starting positions, and a handle to the **generator**. Generators are predefined classes meant to create the `.net.xml` and `.sumo.cfg` files that define the system.

The **environment** holds methods to initialize, step through, and reset a simulation; definitions for the observation and action spaces; and methods that aggregate information to calculate observations and rewards.

Flow comes with `base_env` and `base_scenario` modules that serve as parent classes defining methods for initialization, stepping, and resetting the simulation. Taking advantage of inheritance makes it especially easy to extend the existing scenarios and environments to modify an experiment.

When an environment is initialized, it starts a SUMO process on an open port loaded with the appropriate configuration files. When the simulation loads, Flow stores each vehicles initial position so that they can be reset at the end of a rollout. Figure 3 presents a process diagram for Flow, demonstrating the various interactions between the RL library, Flow and SUMO over the course of an experiment.

The reinforcement learning library calls the `step` method repeatedly, passing in a set of RL actions. The actions are applied to the system through the `apply_rl_actions` method. This method is deliberately separated from the rest of the `step` method so it can be overwritten in different subclasses. Once actions have been applied, the simulation is progressed by calling the TraCI command `simulationStep`. Finally the resulting reward and a new observation are calculated and returned to the reinforcement learning library.

At the end of the rollout, the simulation is reset back to its initial state. This is done by removing each vehicle from the simulation, and adding back a new vehicle with the same id to the original location. We specifically chose to reset this way because it is more computationally efficient than restarting a SUMO instance.

5 Experiment Structure

Experiments in Flow can be thought of as consisting of three portions:

- Task Design

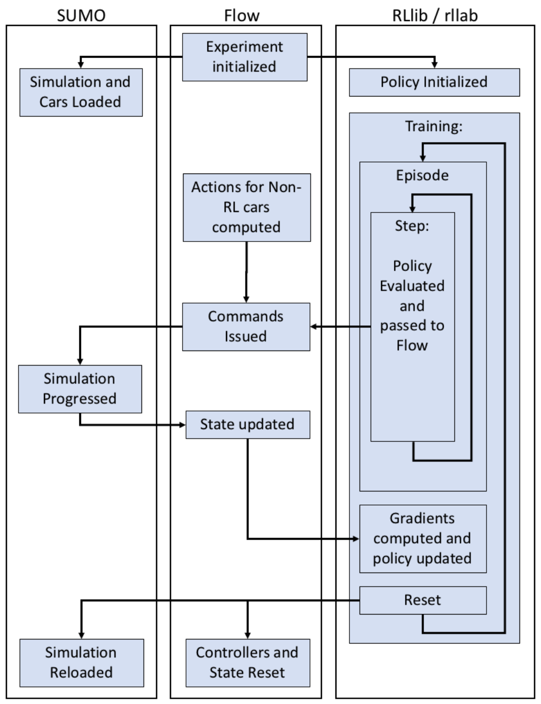

SUMO
Flow
RLlib / rllab
Experiment initialized
Simulation and Cars Loaded
Policy Initialized
Training:
Episode
Step:
Policy Evaluated and passed to Flow
Actions for Non-RL cars computed
Commands Issued
Simulation Progressed
State updated
Gradients computed and policy updated
Reset
Simulation Reloaded
Controllers and State Reset

Figure 3: Flow Process Diagram. Flow manages coordinating SUMO with the observations, actions, and rewards needed and produced by the reinforcement learning libraries by a reinforcement learning library. Through TraCI, Flow is able to load, progress, and reset the simulation. The reinforcement learning library internally handles maintaining and updating the policy, providing actions for the RL agents.

- Policy Training
- Policy Evaluation

In this section, we explain how architecture decisions centering on deep RL impacts the implementation of each.

5.1 Task Design

Scenario Generation: For a given scenario, `net.xml` files are generated automatically based on the network parameters (such as ring length or merge network loop sizes). Generator methods are defined per scenario—for example, Flow includes merge network and figure-eight generators, among others. Given network parameters, the generator creates separate XML files specifying the network's nodes, edges, types, connections, and relevant configuration information. It then calls `netconvert` to create a `net.xml` file which is opened using SUMO. Vehicles are placed on the network as specified by an initial configuration (described shortly), at which point the experiment can be run. The flexibility inherent in programmatically generated road networks enables rapid changes to scenarios.

Converting OpenStreetMap networks into `net.xml` files is supported in Flow, and leverages customizable features of SUMO's `netconvert` method to generate networks which are computationally less burdensome and relevant to vehicle control. This includes removing edges from the network which do not support vehicle classes using `--remove-edge.by-vclass` and edges that are isolated from the rest of the network with `--remove-edges.isolated`.

Just as road network sizes can be changed quickly between experiments, so too can the number and characteristics of vehicles. Vehicles and their parameters are specified within an experiment run script; such parameters include the name of the vehicle type, the car-following and lane-changing models used, the route(s) to take, and SUMO configuration settings such as speed & lane-change modes, car-following parameters, and lane-changing parameters [13]. A vehicle type is defined by its parameters inside an experiment run script, which also includes the number of each vehicle type. Flow supports SUMO's repeated vehicle emissions, or "flows", through an `InFlows` class requiring all the parameters of a SUMO flow defined in the `params` module.

Flow defines a Vehicles class which acts as an abstraction barrier between custom environments and TraCI. This barrier simplifies the creation of Flow environments and controllers. Furthermore, it removes the need for users to make TraCI calls and speeds up simulation. Information regarding vehicles in the experiment is fetched using a SUMO subscription at each timestep and stored. The Vehicles class includes utility functions used by car-following models, lane-change controllers, etc. This collection of utility functions includes functions to return vehicle headways, leaders, followers, routes, and properties. Setter methods use TraCI to modify vehicle states and properties when necessary and are included also.

Flow supports uniformly spaced vehicles as well as vehicles spaced randomly within a lane. Custom initial configurations of vehicles are supported. A set of starting edges can be specified, so that vehicles are not initially spread throughout the entire network but instead occupy a smaller section at greater density. Heterogeneous distributions of vehicles across lanes are supported as well. The order of vehicles can be shuffled to train policies capable of identifying and tracking vehicles across time. This shuffling can be set to occur once at the start of an experiment, or before each rollout to randomize the conditions in which the agent trains. In order to prevent instances of the simulation from terminating due to numbers of initial vehicles that will lead to overlapping vehicles, a minimum gap parameter is implemented. This variable ensures that the minimum bumper-to-bumper distance between two vehicles never drops before a certain threshold. Flow raises an error before an experiment begins if the density is too high to support this gap.

Vehicle Controller Design: Custom vehicle behavior is supported in Flow. Users can create car-following models of their choosing by instantiating an object corresponding to the model with a `get_accel` method that returns accelerations for a vehicle. At each timestep, Flow fetches accelerations for each controlled vehicle using `get_accel`; these accelerations are then Euler integrated to find vehicle velocity at the next timestep, which is commanded using TraCI. The implementation of lane-changing controllers is supported also, using objects corresponding to a lateral controller that define a method `get_action` that returns a valid lane number. Target lanes are passed to a lane-change applicator function within the `base_env` module, which uses TraCI to send a `changeLane` command. Desired lane-change behavior can be set on a per-vehicle basis by specifying the relevant 12-bit SUMO lane change mode value.

Currently, Flow supports all of SUMO's default longitudinal and lateral dynamics models, as well as other longitudinal models. The bilateral car-following model described in [46, 47], the Optimal Velocity Model in [48], and the Intelligent Driver Model (IDM), with and without noise, of [35] and [49] are implemented in Flow as car-following models as described in Section 3.2. Human drivers in Flow experiments are modeled using a noisy IDM model with Gaussian noise $\mathcal{N}(0, 0.2)$ as in [49]. Autonomous vehicles in Flow are defined by their use of the `RLController` object, which sends commands from the policy at each timestep; as a result, RL vehicles are completely controlled by the RL agent. These commands vary depending on the MDP's action space. Each environment includes a method `apply_rl_actions` which commands the RL vehicles as needed, using Flow's built-in methods to interact with TraCI.

Reinforcement Learning Properties To appropriately define an experiment for RL control, it must have methods to generate observations and rewards; furthermore, it must be able to convert the actions supplied by the RL library into concrete actions to be applied via TraCI.

Flow gives users a great deal of flexibility in defining observation spaces, as the user has access to every part of the environment. This makes it easy to construct large observation spaces that detail the positions and velocities of every car in the simulation. The `Vehicles` class is specifically designed to make generating observation spaces as efficient as possible as it stores state variables in memory and provides many useful utility functions.

Each environment, when defined, includes a method to interpret the action returned by the RL library. For example, if the agents are autonomous vehicles the policy may provide an acceleration, a lane-change value between 0 and 1, and a direction. If the lane-change value is greater than some threshold, the car will lane-change in the direction specified. If it is less than the threshold, then no action is taken.

However, Flow agents are not limited to vehicles; agents may be traffic lights, speed limits over road sections, and more. Scenarios with variable speed limits, for example, act by iterating through controlled vehicles, identifying the speed limit for the section of road the vehicle is in, and setting the vehicle's maximum speed to that speed limit through the TraCI command, `setMaxSpeed`. For agents that are traffic lights, a single action is again compared to a threshold, and if it exceeds the threshold value, the light's color is changed.

Rewards must be defined for an environment in order to provide a training signal for an agent. Flow includes built-in reward functions, rewarding system velocity reaching a target, minimization of total delay, low variance in headway, and more. Such rewards can be weighted and combined in order to incentivize or penalize certain behaviors. Custom rewards can be easily defined as well, by implementing a scalar function of the environment, its vehicles, or other inputs.

5.2 Policy Training

One of the downsides of using RL approaches to develop effective policies is the sample-inefficiency of such methods. Policy gradient methods in particular, are reliant on simulation because of their need for a large number of samples [50, 51]. In order to ensure that policies are trained quickly and reliably, we have implemented several techniques to improve training time and robustness.

Amazon Web Services As a consequence of the need to simulate the evolution of traffic dynamics over large time scales, we integrated Flow with Amazon Web Services (AWS). AWS offers powerful compute hardware that can be accessed via SSH and other methods, so a properly-configured AWS instance running Flow and SUMO could run any experiment supported in Flow. To streamline the process of running experiments and training agents, we used Docker to containerize a SUMO and rllab instance, which is publicly available to enable outside use of Flow. That container can be easily deployed to AWS, which allows Flow experiments to be run in the cloud quickly. For use of RLlib on AWS, an Amazon Machine Image preinstalled with RLlib and SUMO was built. This allows us to quickly spin up images that can run Flow and take advantage of Amazon's autoscaling groups, parallelizing across machines.

Parallelization Typically, a single iteration contains a batch of samples collected from several rollouts of the same policy. The batch is used to compute a Monte Carlo estimate of the gradient used by policy optimization algorithms. Larger batch sizes allow for a diverse set of states to influence the gradient and reduce the stochasticity of the gradient estimate; this makes the gradient step more likely to advance towards towards a global minimum.

Because the policy is held constant throughout each iteration, we can parallelize rollout exection and sampling. We do this by spawning several running instances of SUMO, each on a different port, allowing TraCI to connect to and distinguish between simulations. This allows for us to take advantage of multi-cored machines, and drastically improves training time.

Furthermore, Ray and RLlib support parallelization across multiple machines, correctly distributing and aggregating reward across nodes in a cluster. Reinforcement learning experiments in Flow using RLlib can then take advantage not only of more than one core per node, but can distribute computation across nodes. The benefits of this approach to parallelization are significant; using, say, four 16-core computers is more cost-effective than using a single 64-core computer. Furthermore, the potential maximum number of compute threads is simply higher when distributed cluster systems are used.

The following table describes values associated with running one iteration of a reinforcement learning task in which the agent sought to stabilize a single-lane ring. The data shown demonstrates that distributing rollouts across multiple nodes with fewer CPUs per node is more cost-effective and faster than using a single, more powerful node. Each batch was made up of 144 rollouts. The columns are, in order, the number of nodes used, total number of CPUs used, time per *rllab rollout*, time per *RLlib rollout*, RLlib rollout speedup relative to a one-node 16-CPU instance, and cost per hour on Amazon EC2.

Nodes	**CPUs**	**rllab rollout (s)**	**RLlib rollout(s)**	**Speedup**	**Cost ($/hr)**
1	72	79s	74s	2.58x	1.08
1	16	205s	191s	1.00x	0.24
2	32	*N/A*	98s	1.95x	0.48
4	64	*N/A*	62s	3.08x	0.96
8	128	*N/A*	42s	4.55x	1.93

Note that the rollout using 64 CPUs total across four nodes ran faster in RLlib (62s) than the rollout using 72 CPUs on one node. This result is not intuitive, as one might expect a single, powerful machine to outperform a less powerful cluster. We hypothesize this result occurred due to memory constraints on the 72-CPU node; the four-node cluster with 64 CPUs includes more total memory.

Subscription Speedups As a course of our work, we found that limiting the number of TraCI calls directly improves the speed of the simulation. As a result, we've leveraged TraCI's subscriptions to update a global record of various state variables. This allows us to avoid directly querying SUMO and incurring computational overhead through TraCI unnecessarily. Flow is designed to make as few calls to TraCI as possible, and instantiates subscriptions to all relevant variables at the beginning of each experiment. At each timestep, information returned by the subscription is stored in a Python dictionary, which has constant-time lookup—significantly faster than fetching required values through TraCI calls. In the future, we look forward to using `libsumo` to directly interface Flow's Python codebase with SUMO and reduce communication overhead through the socket.

Failsafes and Early Termination As an RL policy is being learned, an unrestricted RL vehicle may crash into cars around it. Thus, we explored strategies to prevent rollouts from terminating and our training from ending. Attaching a large negative penalty to crashes would indeed discourage the car from crashing; however, this greatly increases the variance of rollout rewards. These sparse negative rewards make it difficult for RL to learn anything other than crash avoidance.

Instead, to encourage the RL car to learn traffic-mitigating behavior while preventing crashes, we instead terminate the rollout immediately upon a crash. In scenarios with non-negative rewards, this truncation of rollouts stops the accrual of reward, and so policies that lead to crashes result in low reward. In this manner, the agent learns to optimize for longer-running simulations.

However, this is still a difficult problem to solve—as such, we leverage both the failsafes provided in SUMO, and allow for custom failsafe models that can completely eliminate crashes. This can be accomplished by setting an appropriate speedmode, usually '31' to enforce all checks. Similarly, with lane changing modes, we enforce surrounding gap constraints with a lane change mode of '512'.

Flow also has the capability for a user to implement custom fail-safes. A potential user-defined fail-safe might be a rule that brings vehicles to a halt if their headway is below some threshold and risks a collision. Flow includes a built-in fail-safe, called the *final position rule*, that considers the maximum deceleration a of vehicles in the scenario and the delay τ of a subject vehicle [12]. The delay τ is the same τ as would delay the vehicle's car-following model. If the lead vehicle (ahead of the subject) is at position $x_{\text{lead}}(t_0)$ and decelerates at rate a, then its final position is $x_{\text{lead}}(t_f) = x_{\text{lead}}(t_0) + \frac{v_{\text{lead}}^2(t_0)}{2a}$. A delay of τ for the subject vehicle means that the subject travels $v_{\text{safe}}\tau$ before decelerating at rate a. Thus the final position of the subject $x_{\text{subj}}(t_f) = x_{\text{subj}}(t_0) + v_{\text{safe}}(\tau + \frac{v_{\text{safe}}}{2a})$. We set v_{safe} such that $x_{\text{subj}}(t_f) < x_{\text{lead}}(t_f)$. All commanded velocities to the subject vehicle must be below v_{safe} so that if the lead vehicle decelerates at its maximum rate, the subject vehicle will still come to rest at the rear bumper of the lead vehicle. This fail-safe is active during both training and testing.

5.3 Policy Evaluation

Policy evaluation is key to understanding agents trained by a reinforcement learning algorithm. Many methods of evaluation exist, the most naive of which is analyzing the cumulative discounted reward accrued by the agent over time horizon T. Flow includes utilities to facilitate the evaluation of policy performance via other, more informative means.

Policy Visualizer Experiment visualizers for both rllab and RLlib are included with Flow. Both libraries store the results of training, rllab in pickle files and RLlib in binary and hexadecimal checkpoint format. Each library includes utilities for reloading an agent at a saved stage of training. Flow's built-in visualizers load or initialize an agent from the saved results as well as an environment similar to those described previously. A number of rollouts are then visualized using the trained agent as the environment steps through time. The visualizer for RLlib-trained agents uses parameters stored by an experiment during training to restore the policy configuration and environment parameters.

During visualization, the observations and rewards passed to the agent, normally used for training, are instead stored in a file. These stored values can be used to evaluate agent performance; for example, velocity observations averaged over time could be used to see if an agent improved average speeds over random behavior or some baseline.

Space-Time Plotting Flow also includes a utility to load agent training results, replay a policy rollout, and plot the results. The resulting plot shows one trace per vehicle graphing position against time, color-coded by the vehicle's velocity at that time. Figures 5a and 5b are examples of such space-time plots.

6 Experiment Results

Here, we describe the results of experiments in which traffic-mitigating agents were trained using reinforcement learning in Flow. An in-depth example of traffic stabilization on a two-lane ring road serves as a case study. Further results can be found in our other work [5, 12].

(a) The 2008 Sugiyama ring road experiment [44]

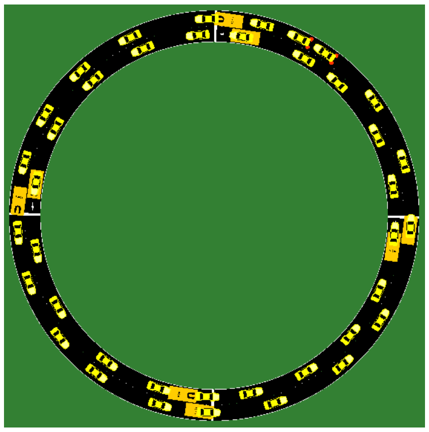

(b) Two-lane ring road

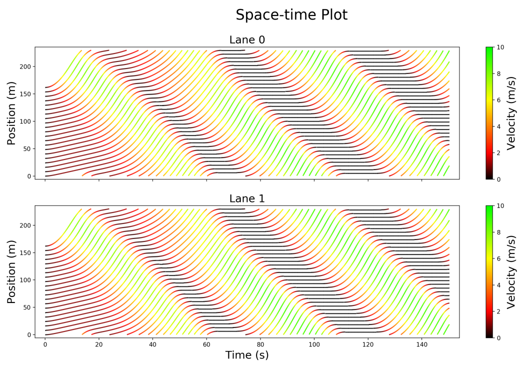
Figure 4

6.1 Case Study: Ring Roads

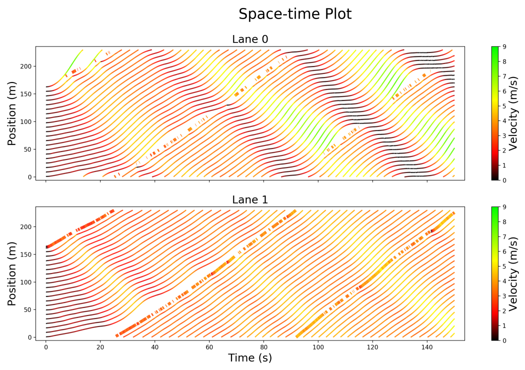
The work of Sugiyama et al. (shown in Figure 4a) showed that human-driven traffic traveling in a ring road was unstable and eventually decayed into stop-and-go waves [44]. The Sugiyama experiment ran 22 vehicles on a 230m ring road and identified that traffic, in some parts of the ring, came to a complete halt despite the lack of external perturbation. Stern et al. studied single-lane ring road stability as well, demonstrating that a longitudinal controller deployed on an autonomous vehicle reduced congestion [7]. In an earlier article, we reproduced the Stern setup in Flow and trained a policy to maximize the two-norm of the system velocity [12]. The longitudinal behavior of the 21 human drivers was modeled using the Intelligent Driver Model as our car-following model. The 22$^{\text{nd}}$ car was controlled by the reinforcement learning algorithm and acted as the agent in this scenario. We found that the RL-trained policy for traffic stabilization outperformed the controllers in the Stern experiment [7, 12].

However, in the multi-lane case, a car-following model alone is not sufficient to stabilize the ring, as the additional lanes will continue to exhibit waves. To stabilize both lanes, it is necessary to introduce a trained lane-changing controller. In the scenario we present, 41 IDM vehicles using SUMO's default lane-changing model and one RL-controlled agent vehicle were placed on a 230m, two-lane ring. Using the reward function in [12], we train the RL vehicle to accelerate and lane change so as to maximize the two-norm of the velocity of all vehicles in the system. The RL agent is passed observations consisting of the speed & position of the RL-controlled vehicle and the headway & speed of the first trailing vehicle in each lane.

The RL agent learns to prevent the propagation of traffic waves by rapidly changing lanes, in effect blocking traffic from passing in both lanes. This prevents human drivers from achieving the speeds where shockwaves begin to occur. Human-driven vehicles in our system do not have the opportunity to accelerate to high speeds and then decelerate when nearing a vehicle, and



(a) Human-only space time diagram of traffic flow in the two-lane ring. The lines are color-coded by speed, with green representing high speeds and red & black representing low speeds. In this scenario, traffic flow decays into stop-and-go waves, with traffic oscillating between low and high speeds.



(b) RL-controlled space time diagram of traffic flow in the two-lane ring. The RL car is the thicker line. Empty portions in the RL line represent time spent in the adjacent lane. Fewer stop-and-go waves occur in lane 1, where the RL car spends the majority of its time. The RL car has some impact on lane 0 as well, reducing the magnitude of the stop-and-go waves in that lane.

Figure 5

so the behavior that leads to traffic waves is prevented. This very closely resembles the role that the single-lane traffic stabilization policy played in the traffic scenario as well [12]. This behavior, rapidly moving between lanes to slow traffic and reduce instability caused by rapid acceleration and deceleration, occurs in real life; police cars trying to clear the road around an accident execute "traffic breaks" in which they swerve between lanes and prevent cars from passing them [52]. The trained policy reduces congestion in the two-lane system; the waves seen in the all-human case Figure 5a dissipate upon introduction of the agent as shown in Figure 5b. The trained policy improves the system average velocity to 3.11 m/s in the RL-stabilized case from a human-only average velocity of 2.83 m/s. Figure 5b shows that the autonomous vehicle spends the majority of its time in lane 1; correspondingly, that lane has less extreme backward-propagating waves. The per-lane average velocities are 2.93 m/s and 3.30 m/s for lane 0 and lane 1, respectively. Reinforcement learning-based methods of traffic control using vehicles arrive upon similar results to those theoretically derived as in [7], in both the one- and two-lane case. However, the increased complexity of the two-lane setting caused by lateral lane-changing dynamics complicates the task of developing theoretical approaches. On the other hand, RL can be easily extended to this more complex setting and yield a high-performing solution.

7 Conclusion

Modern deep reinforcement learning methods are powerful and show promise for traffic control applications but have very large data requirements. Flow is an open-source framework built upon SUMO to facilitate the simulation of complex traffic scenarios and the use of reinforcement learning to train autonomous vehicle policies within those scenarios. In order to support the data requirements, we implemented a number of features and components within Flow, including AWS support, distributed simulation, use of subscriptions, failsafe and early termination handling. These architectural and engineering decisions may be of general interest to the SUMO community, for other use cases with high data requirements or large simulation times. Finally, we demonstrated the use of Flow for the challenging control task of maximizing velocity in a mixed-autonomy two-lane ring road.

Further exploration of traffic control schemes for the multi-lane ring is planned; a topic of particular interest is the benchmarking of RL-trained stabilization methods against existing methods for control in the literature. Future goals for Flow and our work include: refining multi-agent reinforcement learning algorithms; streamlining the process of training agents on arbitrary imported road networks; evaluating traffic scenario safety; leveraging `libsumo` to speed up simulating traffic dynamics; and exploring more efficient methods of simulating system dynamics.

Acknowledgments

The authors would like to thank Jakob Erdmann and the entire SUMO team and community for supporting and providing a rich, open-source, and extensible traffic microsimulation system upon which we (and others) could build frameworks which tackle challenging traffic problems. We would also like to thank Philipp Moritz, Robert Nishihara, and Ion Stoica (RISELab, UC Berkeley) for their support in integrating RLlib and Ray with Flow. We would also like to thank the deep learning community for its deep nomenclature suggestions.

References

[1] G. Cookson and B. Pishue, “2017 INRIX Global Traffic Scorecard,” 2018.

[2] D. Schrank, B. Eisele, T. Lomax, and J. Bak, “2015 Urban Mobility Scorecard,” Texas A&M Transportation Institute and INRIX, Inc, Tech. Rep., 2015.

[3] J. Kwon and P. Varaiya, “Effectiveness of California’s High Occupancy Vehicle (HOV) system,” *Transportation Research Part C: Emerging Technologies*, vol. 16, no. 1, pp. 98 – 115, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0968090X07000435

[4] D. Swaroop, “String stability of interconnected systems: An application to platooning in automated highway systems,” 1997.

[5] C. Wu, A. Kreidieh, E. Vinitsky, and A. M. Bayen, “Emergent behaviors in mixed-autonomy traffic,” in *Proceedings of the 1st Annual Conference on Robot Learning*, ser. Proceedings of Machine Learning Research, S. Levine, V. Vanhoucke, and K. Goldberg, Eds., vol. 78. PMLR, 13–15 Nov 2017, pp. 398–407. [Online]. Available: http://proceedings.mlr.press/v78/wu17a.html

[6] T. Babicheva, “The use of queuing theory at research and optimization of traffic on the signal-controlled road intersections,” *Procedia Computer Science*, vol. 55, pp. 469 – 478, 2015, 3rd International Conference on Information Technology and Quantitative Management, ITQM 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S187705091501491X

[7] R. E. Stern, S. Cui, M. L. D. Monache, R. Bhadani, M. Bunting, M. Churchill, N. Hamilton, R. Haulcy, H. Pohlmann, F. Wu, B. Piccoli, B. Seibold, J. Sprinkle, and D. B. Work, “Dissipation of stop-and-go waves via control of autonomous vehicles: Field experiments,” *CoRR*, vol. abs/1705.01693, 2017. [Online]. Available: http://arxiv.org/abs/1705.01693

[8] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[9] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, “High-dimensional continuous control using generalized advantage estimation,” *arXiv preprint arXiv:1506.02438*, 2015.

[10] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *Journal of Machine Learning Research*, vol. 17, no. 39, pp. 1–40, 2016.

[11] F. Belletti, D. Haziza, G. Gomes, and A. M. Bayen, "Expert level control of ramp metering based on multi-task deep reinforcement learning," *CoRR*, vol. abs/1701.08832, 2017. [Online]. Available: http://arxiv.org/abs/1701.08832

[12] C. Wu, A. Kreidieh, K. Parvate, E. Vinitsky, and A. M. Bayen, "Flow: Architecture and benchmarking for reinforcement learning in traffic control," *arXiv preprint arXiv:1710.05465*, 2017.

[13] D. Krajzewicz, J. Erdmann, M. Behrisch, and L. Bieker, "Recent development and applications of SUMO - Simulation of Urban MObility," *International Journal On Advances in Systems and Measurements*, vol. 5, no. 3&4, pp. 128–138, December 2012.

[14] M. G. Bellemare, Y. Naddaf, J. Veness, and M. Bowling, "The arcade learning environment: An evaluation platform for general agents," *J. Artif. Intell. Res.(JAIR)*, vol. 47, pp. 253–279, 2013.

[15] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *Intelligent Robots and Systems (IROS), 2012 IEEE/RSJ International Conference on.* IEEE, 2012, pp. 5026–5033.

[16] T. Schaul, J. Togelius, and J. Schmidhuber, "Measuring intelligence through games," *CoRR*, vol. abs/1109.1314, 2011. [Online]. Available: http://arxiv.org/abs/1109.1314

[17] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," *CoRR*, vol. abs/1604.06778, 2016. [Online]. Available: http://arxiv.org/abs/1604.06778

[18] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *CoRR*, vol. abs/1509.02971, 2015. [Online]. Available: http://arxiv.org/abs/1509.02971

[19] E. Catto, "Box2d: A 2D physics engine for games," 2011.

[20] A. Coles, A. Coles, O. Sergio, S. Sanner, and S. Yoon, "A survey of the seventh international planning competition," *AI Magazine*, 2012.

[21] A. Dosovitskiy, G. Ros, F. Codevilla, A. López, and V. Koltun, "CARLA: an open urban driving simulator," *CoRR*, vol. abs/1711.03938, 2017. [Online]. Available: http://arxiv.org/abs/1711.03938

[22] Y. Lv, Y. Duan, W. Kang, Z. Li, and F.-Y. Wang, "Traffic flow prediction with big data: a deep learning approach," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 2, pp. 865–873, 2015.

[23] N. G. Polson and V. O. Sokolov, "Deep learning for short-term traffic flow prediction," *Transportation Research Part C: Emerging Technologies*, vol. 79, pp. 1–17, 2017.

[24] E. Walraven, M. T. Spaan, and B. Bakker, "Traffic flow optimization: A reinforcement learning approach," *Engineering Applications of Artificial Intelligence*, vol. 52, pp. 203 – 212, 2016. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0952197616000038

[25] R. Bellman, "A Markovian decision process," DTIC Document, Tech. Rep., 1957.

[26] R. A. Howard, "Dynamic programming and Markov processes," 1960.

[27] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "A brief survey of deep reinforcement learning," *CoRR*, vol. abs/1708.05866, 2017. [Online]. Available: http://arxiv.org/abs/1708.05866

[28] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *Nature*, vol. 521, pp. 436 EP –, 05 2015. [Online]. Available: http://dx.doi.org/10.1038/nature14539

[29] R. S. Sutton, D. A. McAllester, S. P. Singh, Y. Mansour *et al.*, "Policy gradient methods for reinforcement learning with function approximation." in *NIPS*, vol. 99, 1999, pp. 1057–1063.

[30] J. Peters, "Policy gradient methods," *Scholarpedia*, vol. 5, no. 11, p. 3698, 2010, revision #137199.

[31] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," in *Reinforcement Learning.* Springer, 1992, pp. 5–32.

[32] J. Schulman, S. Levine, P. Abbeel, M. I. Jordan, and P. Moritz, "Trust region policy optimization," in *ICML*, 2015, pp. 1889–1897.

[33] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. [Online]. Available: http://arxiv.org/abs/1707.06347

[34] G. Orosz, R. E. Wilson, and G. Stépán, "Traffic jams: dynamics and control," *Philosophical Trans. of the Royal Society of London A: Mathematical, Physical and Engineering Sciences*, vol. 368, no. 1928, pp. 4455–4479, 2010.

[35] M. Treiber and A. Kesting, "Traffic flow dynamics," *Traffic Flow Dynamics: Data, Models and Simulation, Springer-Verlag Berlin Heidelberg*, 2013.

[36] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, "Ray RLLib: A composable and scalable reinforcement learning library," *arXiv preprint arXiv:1712.09381*, 2017.

[37] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," *arXiv preprint arXiv:1606.01540*, 2016.

[38] A. Wegener, M. Piorkowski, M. Raya, H. Hellbrck, S. Fischer, and J.-P. Hubaux, "Traci: An interface for coupling road traffic and network simulators," *11th Communications and Networking Simulation Symposium (CNS)*, 2008.

[39] S. Krauß, P. Wagner, and C. Gawron, "Metastable states in a microscopic model of traffic flow," *Physical Review E*, vol. 55, no. 5, p. 5597, 1997.

[40] S. Krauß, "Microscopic modeling of traffic flow: Investigation of collision free vehicle dynamics," Ph.D. dissertation, 1998.

[41] J. Erdmann, "Lane-changing model in sumo," *Proceedings of the SUMO2014 modeling mobility with open data*, vol. 24, pp. 77–88, 2014.

[42] ——. (2016) Simulation of urban mobility - wiki: Car-following-models. [Online]. Available: http://sumo.dlr.de/wiki/Car-Following-Models#tau

[43] (2016) Simulation/basic definition. [Online]. Available: http://sumo.dlr.de/wiki/Simulation/Basic_Definition#Defining_the_Time_Step_Length

[44] Y. Sugiyama, M. Fukui, M. Kikuchi, K. Hasebe, A. Nakayama, K. Nishinari, S.-i. Tadaki, and S. Yukawa, "Traffic jams without bottlenecks–experimental evidence for the physical mechanism of the formation of a jam," *New Journal of Physics*, vol. 10, no. 3, p. 033001, 2008.

[45] A. D. Spiliopoulou, I. Papamichail, and M. Papageorgiou, "Toll plaza merging traffic control for throughput maximization," *Journal of Transportation Engineering*, vol. 136, no. 1, pp. 67–76, 2009.

[46] B. K. Horn, "Suppressing traffic flow instabilities," in *Intelligent Transportation Systems-(ITSC), 2013 16th International IEEE Conference on.* IEEE, 2013, pp. 13–20.

[47] L. Wang, B. K. Horn, and G. Strang, "Eigenvalue and eigenvector analysis of stability for a line of traffic," *Studies in Applied Mathematics*, 2016.

[48] I. G. Jin and G. Orosz, "Dynamics of connected vehicle systems with delayed acceleration feedback," *Transportation Research Part C: Emerging Technologies*, vol. 46, pp. 46–64, 2014.

[49] M. Treiber and A. Kesting, "The Intelligent Driver Model with stochasticity-new insights into traffic flow oscillations," *Transportation Research Procedia*, vol. 23, pp. 174–187, 2017.

[50] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, "Q-prop: Sample-efficient policy gradient with an off-policy critic," *arXiv preprint arXiv:1611.02247*, 2016.

[51] K. Asadi and J. D. Williams, "Sample-efficient deep reinforcement learning for dialog control," *CoRR*, vol. abs/1612.06000, 2016. [Online]. Available: http://arxiv.org/abs/1612.06000

[52] "California driver handbook - special driving situations." [Online]. Available: https://www.dmv.ca.gov/portal/dmv/detail/pubs/hdbk/driving_in