2023 IEEE 26th International Conference on
Intelligent Transportation Systems (ITSC)
24-28 September 2023. Bilbao, Bizkaia, Spain

MontiSim: Agent-Based Simulation for Reinforcement Learning of Autonomous Driving*

Tristan Höfer[1], Mattis Hoppe[1], Evgeny Kusmenko[1], Bernhard Rumpe[1]

Abstract— **Reinforcement learning is a machine learning method particularly interesting for the autonomous driving domain, as it enables autopilot training without the need for large and expensive amounts of manually labeled training data. Instead, agents are trained by evaluating the effects of their actions and punishing or rewarding them accordingly. In autonomous and particularly cooperative driving a core problem is however that multiple vehicles need to be trained in parallel while having an impact on each other's behavior. In this paper, we present a simulation solution providing cooperative training capabilities out-of-the-box and compare the quality of the resulting autopilots in an intersection scenario.**

I. Introduction

Reinforcement learning is a machine learning method which can be used to train agents such as autonomously driving autopilot software to make decisions in their environment. The agent, i.e. the autonomous vehicle, produces actions to alter its own physical state and its environment, e.g. by accelerating, braking, steering, or sending V2V messages. Based on the resulting environmental change, the agent is punished in the case of a deviation of the desired behavior, e.g. a crash or a divergence from the planned trajectory, or rewarded, if the actions led to the desired state. The reward or punishment is a numerical score, which the training algorithm uses to adapt the behavior of the agent. Mostly, a pre-training is carried out in a *simulator*, since conducting the necessary experiments using a real vehicle would be not only costly, but also slow and dangerous (in our example, we needed several thousand training episodes). Once the network is sufficiently pre-trained, a fine-tuning can be carried out in a real vehicle.

We are particularly interested in simulation-based reinforcement learning for automated cooperative driving. Hence, a suitable training environment is required. And while it is straight-forward to train a single vehicle in a given environment, the situation becomes more intricate, if we have several vehicles in a scenario, all using the same autopilot software to be trained. Hence, the vehicles need to learn simultaneously, each one probably starting from zero and depending on the equally untrained behavior of its peers. Consequently, the agents need to synchronize what they have learned, making the training more complicated and requiring a lot of manual implementation work by the developer. In this paper we present a simulation toolchain for cooperative reinforcement learning based on the MontiSim ITS simulator offering functionality for basic reinforcement learning as well as several cooperative reinforcement learning approaches out-of-the-box. In an evaluation we demonstrate the general capability of our training environment as well as the two competing learning approaches for cooperative driving in an urban setting and compare the resulting autopilots.

The remainder of this paper is structured as follows. In section II we discuss other works concerned with multi-agent reinforcement learning. Section III introduces the EMADL deep learning framework which we use for reinforcement learning and MontiSim, the ITS simulator used as environment for the vehicles to be trained. In section IV we discuss the extensions we developed for MontiSim as well as the two cooperative learning approaches which we implemented in our toolchain. Section V summarizes the training results of the discussed training methods before the paper is concluded in section VI.

II. Related Work

A. TORCS

The open racing car simulator (TORCS) [1] is an open source multi-agent car simulator. TORCS focuses on low-level simulation of racing cars in racing scenarios. Hence, several physical phenomenons such as realistic aerodynamics or varying friction across different surfaces on the track.

TORCS provides a low level API that allows detailed information from sensors on the cars that grant insight into current state of the simulation. This data made it possible to use the simulator as a training environment for a reinforcement learning agent [2]. The state is composed of a 29-dimensional vector containing the sensor values, and the action is a 3-dimensional vector containing the basic inputs throttle, braking and steering that are required to handle a vehicle.

Since TORCS contains racing scenarios, the reward function intended to maximize performance within a race. Hence, on the one hand it had high-level goals like minimizing the lap time of the agent by finding optimal trajectories across the track. On the other hand, however, the basic driving tasks such as staying on the track in corners, while still maximizing the velocity needed to be considered.

Even though TORCS works well as a training environment within its intended domain, for the training of agents used in everyday-life driving especially with a large number of agents, it is not suited, due to the racing orientated nature of its scenarios and the low-level nature of the simulation.

*This work was supported by the grant SPP1835 of the German Research Foundation (DFG).

[1]All authors are with the Software Engineering department of the RWTH Aachen University, Germany.

B. MATSim

The Multi-Agent Transport Simulation (MATSim) [3] is an open-source, multi agent simulation framework implemented in Java. Since it's purpose is to simulate the daily traffic of whole cities, the simulation is high-level. The main goal of the simulator is to find optimal traffic distribution with respect to time and routing. This enables investigating possible weaknesses in the street layout, and exploring new solutions without incurring high cost in case of failure.

The people using the transportation system of the city are modeled as agents. Every agent has an own agenda consisting of a collection of tasks to fulfill during the simulated day. Tasks are usually destinations to which an agent has to travel using the transportation infrastructure. The tasks are usually obtained from studies of that specific area, but can be artificial as well.

Each agent chooses a plan out of its agenda at the beginning of a simulation. The plan defines in which order the tasks are completed, the departure time for each task, the route taken to reach the location of each task and the vehicle used for reaching each location. The maximal number of plans in an agenda is fixed and agent dependent.

After every simulated day, a score for the plan taken by each agent is calculated. The score of a plan is obtained by applying a function to the scores of the different tasks of the respective agent which is then used to change existing, and create new plans for future days. The criteria which are taken into account for each task is customizable and task dependent. The goal of an agent is to obtain a plan with a maximized score.

Since MATSim focuses on high-level simulation of traffic, low level simulation is only possible to a limited extend, due to the extensive computation power that would be required otherwise.

III. Foundations

A. EMADL Learning Framework

EmbeddedMontiArc Deep Learning (EMADL) is a framework that allows modeling and training of deep reinforcement learning pipelines. It is composed out of three different languages which are introduced in the following.

The heart of EMADL is the textual modeling language EmbeddedMontiArc (EMA) [4], [5] that enables modeling architectures as a system of hierarchically organized interacting components. It was developed with the MontiCore language workbench, that allows the development of domain-specific languages. EMA is based on the component and connector (C&C) paradigm, which enables the division of a system into different components. Thus, components can communicate and exchange data between each user using ports. The data type system includes mathematical primitive types which can be extended to multi-dimensional vectors and matrices. One advantage of the C&C paradigm is the separation of tasks into smaller tasks that can be accomplished independently by a component. Because the components are connected, they can provide a solution to the main task by combining the partial results.

To define atomic components EMADL offers two behavior modeling languages called **MontiAnna** and **MontiMath**.

The deep learning framework MontiAnna [6] enables easy declarative modeling of deep neural networks. MontiAnna allows detailed constructing of network architecture via the language CNNArch and a definition of training parameters using the language ConfLang. The developer can choose between common neural network layers such as convolutional or fully-connected. Furthermore the input and output ports of the EMADL component can be seamingless integrated as input and output layers of the network architecture. In an additional file, the training configuration can be defined which include typical deep learning parameters but also hyperparameters for different reinforcement learning algorithms [**?**], e.g. DDPG [7]. MontiAnna is especially important for this work to configure and train the autopilots using reinforcement learning.

The benefit of splitting up the architecture and the training definition is that both offer options for modification without affecting the other one. For using datasets, a declarative dataset model exists. The architecture, dataset, and training definition are merged before generating the C++ or Python code for the model. After the compatibility between the different components has been validated, code for popular deep learning frameworks such as MxNet, Caffe2, or Tensorflow can be generated [8].

The second language for atomic components, MontiMath [4], is a matrix based programming language similar to MATLAB that enables the implementation of mathematical expressions.

B. MontiSim

MontiSim [9] is an agent based, open-source Intelligent Transportation System (ITS) simulator [10]. Its main goal is to provide an environment for developing and evaluating software for autonomous vehicles in isolated as well as cooperative driving scenarios.

MontiSim balances between a high and low level simulation. Accordingly, it allows simulating more complex driving scenarios with a large number of participating vehicles, and also simulates event-based intra vehicle communication as well as low level physics [11]. This is due to the fact, that MontiSim is focused on everyday-life driving scenarios and therefore is required to have good scalability in terms of the number of participants as well as low enough simulation to simulate the effects of bad driving behavior.

In a simulation scenario, there are a number of vehicles at different starting positions. Each of the vehicles has a task, namely a list of coordinates, it has to reach in the correct order during the course of the simulation. Scenario specific settings as for instance the number of vehicles and their properties such as starting positions, tasks and more low level settings are set in the scenario file, using the JSON format. MontiSim has an integrated high-level navigation and is therefore able to provide the vehicles with trajectory points to reach their destination. In addition, the vehicles have several sensors for, among more, their current velocity, position and orientation. The only way to drive the vehicle is by using the throttle, braking and steering values.

Since the target domain of MontiSim are everyday-life driving scenarios, it must be possible to obtain versatile and realistic scenarios in an easy way. To accomplish this, the world in which the scenarios take place, is imported from OpenStreetMaps (OSM) [12] resulting in street constellations occurring in real life. To further improve realism when testing automotive driving software, the simulator uses a Hardware Emulator [13] that computes worst case execution times of control software in each time step for hardware parameters set in advance. The simulator uses the calculated time to simulate the time required for computation of the control signals [14].

IV. MontiSim for Reinforcement Learning

A. General

So far, MontiSim was only able to execute conventionally programmed control software. However, since the domain of automated driving often makes use of machine learning techniques, we extended MontiSim to provide the possibility for it to be used as a virtual training, execution and evaluation environment for reinforcement learning agents. To achieve a loose coupling between the simulator, the training is realized by ROS [15], which ensures a language independent communication between the processes. While MontiSim is mostly independent of the implementation of the agent and its training, the interface has to match. We intended an easy use of EMADL for this implementation. Hence, the design was chosen such that EMADL models can be generated against the ROS interface of MontiSim directly. The only precondition is that the EMADL autopilot has a state input port to read the environment state (in particular the sensor data) and an action output port to produce actuator commands (steering, breaking, accelerating).

The training result relies heavily on the design choices of the reward function, and it therefore can take time before obtaining any useful results. Hence, integrated a default reward function into MontiSim for autonomous driving that can be used out-of-the-box. However, since the task that the agent is supposed to learn dictates the choice of the reward function, custom reward functions can be designed in EMADL or the simulator itself.

To guarantee synchronization between EMADL agent and MontiSim, the training process is made to be event-based. Meaning that the simulation only proceeds after receiving a signal from the agent. The reason for this is to make the simulation completely reproducible. Furthermore, the calculations done in the training process takeslonger than during the execution of the trained model, which would result in different, unrealistic training results without such synchronization.

So far, MontiSim only had deterministic scenarios, meaning that per scenario the task was to drive a single and constant trajectory. However, only training with a single scenario would lead to strong overfitting to the chosen scenario, and hence bad generalization to unseen tasks. To prevent this, we introduced a randomization process, that chooses new starting and target positions at the start of each episode. The integrated high-level navigation of the simulator ensures that these positions still result in valid trajectories that use the present streets in the map.

Aside from the training of isolated vehicles, another goal is to train agents for cooperative driving vehicles where the training usually contains multiple agents. This, however, results in challenges for the reinforcement learning toolchain, as all vehicles have to be trained in parallel. This means that multiple agents need to process their perceived world state and produce an individual action. All of these agents start off with an untrained neural network and influence each other's training. Furthermore, the learned actions need to be shared, i.e. some synchronization of the network weights is required. Solutions to these challenges are presented in the following.

B. Cooperative Learning

We discuss two alternative approaches for cooperative learning, which we implemented and evaluated as part of the MontiSim simulation framework for reinforcement learning. Our first approach creates separate agents and trains them in a round robin manner, letting the training algorithm think that there is only one agent, with the only difference being the extended state vector. Secondly, the self-play [16] mode uses a learning agent and a self-play agent which is fixed and gets updates from the learning agent in predefined intervals.

1) Mini-Step: In this approach, it is assumed that every vehicle is controlled separately. However, the vehicles are controlled by the same network instance. Considering that EMADL requires the reward for an action to be returned before calculating the next action, MontiSim has to provide rewards for every vehicle instantly after receiving its actions. The problem arises, that the reward r_{ti} for action a_{ti} for vehicle i at simulation step t cannot be calculated, because the reward can only be calculated after the simulation of time step t, that is at time step $t+1$. However, before simulating time step t, one has to obtain the actions for the remaining vehicles $j \in \{i+1, \dots, n\}$, since all vehicles should receive new actions in each time step. Before obtaining these, EMADL requires the reward r_{ti} for a_{ti} to be present which is not possible, as stated before.

This problem can be solved by dividing every time step t of size Δt into n time steps of size $\frac{\Delta t}{n}$. Then, for an action a_{ti} the reward after simulating the reduced time step is returned. After every vehicle has received its action, time steps of accumulated size $n \cdot \frac{\Delta t}{n} = \Delta t$ have been simulated, resulting in time step $t+1$ being reached. Therefore, new actions are obtained after Δt for each vehicle, and the actions for all vehicles are applied in a pseudo-simultaneous matter.

In order to still support cooperative driving to a further extend than simple lane centering, explicit inter-vehicle communication is required that allows the vehicles to send and receive messages from other vehicles. This is modeled by adding $(n-1) \cdot x$ additional values to the state containing the messages sent of all the other vehicles, with x being the number of values communicated from one vehicle to another.

2) Self-Play: Self-play is a well-known reinforcement learning mechanism [16] to train multiple agents by using a separate instance of the same network for each agent. So far, it was mainly used in e.g. competitive games, but was not much explored and tested in the cooperative domain.

Therefore, during training, two agents are executed in parallel in the same environment. An agent trained with EMADL controls one vehicle and learns based on the resulting reward. Meanwhile, the self-play agent simulates the actions for all other vehicles in the scenario. The self-play agent is generated based on the parameters using the EMADL code generator at the beginning of the training process. After a number of episodes a snapshot of the EMADL agent's current policy is created. The self-play agent then gets updated according to the network parameters and weights of the latest snapshot. This update is managed by a script which is called automatically when a snapshot is created. It extracts the newest parameters and restarts the self-play agent. At the beginning of the training, an initial strategy is created that the agent relies on until the first snapshot is taken. The available training parameters of the ConfLang are extended by a new parameter for self-play that activates the network weight update of the self-play agent after each snapshot.

For each vehicle except the one currently being trained by EMADL, the simulator sends the current state of the environment to the self-play agent, which then computes an action. Since the agent executable does not demand a reward for its action, this procedure can be repeated for all not trained vehicles. At last, the currently trained vehicle receives the state and returns its action. Since the simulator now has the action of every vehicle, it can simulate the next step and return the reward to the trained vehicle immediately. For further illustration consider figure 1.

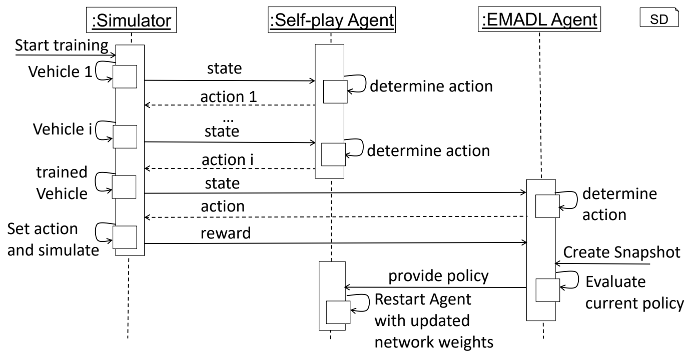

Fig. 1. Sequence diagram representing the training process using a self-play agent.

To make the policy robust to the route of every vehicle the trained vehicle is switched after every episode. Thus the vehicle trained with EMADL is now controlled by the self-play agent. If the trained vehicle remains the same throughout the training, the strategy might learn only one vehicle's route and perform inadequately on other routes.

V. Evaluation

An isolated vehicle as well as the two cooperative driving approaches presented in this paper are trained and tested using MontiSim. The isolated vehicle is trained using a map of Aachen, where start and target positions are randomly assigned at the start of each episode. For the cooperative driving approaches, an urban intersection was selected as the cooperative driving scenario where two vehicles are simulated.This scenario was selected, since it occurs in real life scenarios on a regular basis and a cooperation between both vehicles is necessary to prevent delays or collisions at the intersection center. Furthermore, the intersection was chosen as a simple learning scenario to demonstrate the functioning of the learning process.

All three agents were trained with the TD3-Algorithm, which is a deep reinforcement learning algorithm that uses an actor-critic model. The actor learns the policy and interacts with the environment while the critic tries to estimate the returned reward for the chosen action. Both, the actor and the critic, are modeled as deep neural networks using the EMADL framework. Hyperparameters are tuned using the ConfLang included in EMADL. The state space of each vehicle is 25-dimensional consisting of the trajectory's X- and Y-coordinates, the vehicle's position and it's velocity and angle. The action space is three-dimensional and includes the chosen values for gas, brake and steering on a scale of -1 to 1.

For the cooperative driving approaches, the state space of the vehicle gets extended by four states for each vehicle to support inter-vehicle information exchange. The additional states are the X and Y position coordinates, the current velocity and the angle of the other vehicle. Therefore, when using two cars, the state space becomes 29-dimensional. The action space during the decentralized approach stays the same.

In figure 2, the actor network for the cooperative driving approach implemented in EMADL is displayed. The simulator state is taken as input and the network produces an action as output (l.9 and l.16). The network consists of three fully connected layers with 500 units for the first, 1000 for the second and three for the third layer (l.10-15). In particular, the last layer consists of three units to produce the three dimensional output. The ReLU function, which is widely used in deep learning, is used as the activation function between the fully connected layers. In addition, the Tanh function is used for the estimation of the final output.

One important part of training successful autopilots using reinforcement learning is the reward function $r^{(i)} = r_v^{(i)} + r_{st}^{(i)} + r_d^{(i)} + r_c^{(i)}$ for vehicle i. Here, r_v denotes the current velocity, r_{st} the steering, r_d distance from the lane and r_c collisions with objects and vehicles. Therefore the vehicle receives a positive reward if it is close to one of the desired values and is penalized otherwise.

First the isolated vehicle was trained on random paths in the map for 2500 episodes and 1200 training steps per episode. The average reward of the last 100 episodes over

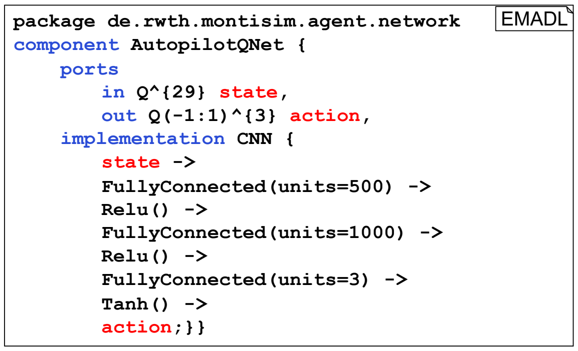
```
package de.rwth.montisim.agent.network
component AutopilotQNet {
    ports
        in Q^{29} state,
        out Q(-1:1)^{3} action,
    implementation CNN {
        state ->
        FullyConnected(units=500) ->
        Relu() ->
        FullyConnected(units=1000) ->
        Relu() ->
        FullyConnected(units=3) ->
        Tanh() ->
        action;}}
```

Fig. 2. The used actor network for the cooperative agents implemented in EMADL.

the course of the training is displayed in figure 3. The reward progression starts at about -900000 and gradually increases up to a reward of 200000. The rewards at the start and end of the training differ from the rewards in the cooperative trainings due to a slightly different reward function being used that e.g. does naturally not have the components that regard to cooperative behavior.

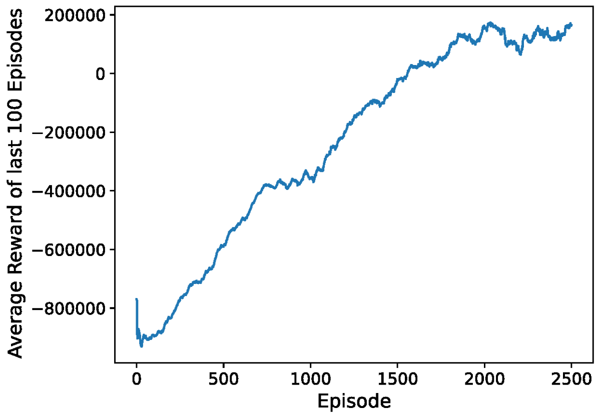

Fig. 3. The training results for the isolated vehicle.

The cooperative training with mini-steps is trained on 1500 episodes with 1200 training steps per episode. In fact one vehicle is trained for 600 steps but since two vehicles are simulated the steps are doubled because of the mini-steps.

The training results, in figure 4, show training success with a reward increase from under -800000 up to approximately +400000. Similar to the centralized approach the reward strongly increases in the first half until it converges in the second half of the training. It is important to note that the reward per episode combines the reward for both vehicles since both vehicles are simulated in every episode. The best policy has an average reward of +414550 for both vehicles, indicating results with satisfying driving capabilities.

The decentralized approach with the self-play agent was trained on 3000 episodes for 600 training steps per episode. This results in 1500 training episodes per agent for two vehicles since the trained vehicle and self-play vehicle are

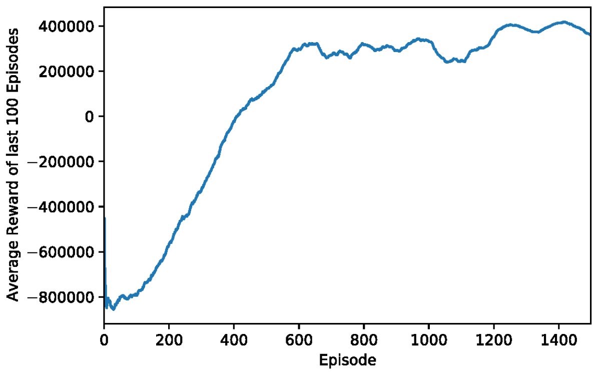

Fig. 4. The training results for the mini-step approach.

swapped after each episode. A snapshot was created after 50 episodes whereupon the network weights of the self play agent are updated.

In figure 5, the average reward for the self-play approach is shown with an almost linear reward increase over the training. The best policy has an average reward of +297250 for each vehicle. This is clearly higher compared to approximately +207275 per vehicle for the mini-step approach.

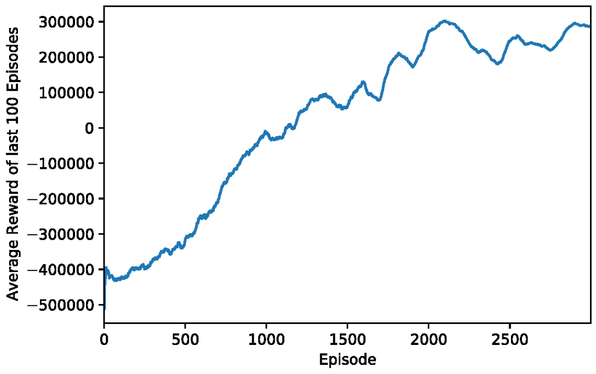

Fig. 5. The training results for the the self-play approach.

Another interesting metric of training success is the number of collisions per episode over the training course displayed in figure 6. During training, collisions decrease, indicating cooperation between the vehicles. Towards the end there are even episodes without any collisions at all, which emphasizes the cooperation between the vehicles.

During execution on the trained scenario, both autopilots show a cooperative driving behavior since the vehicles do not collide in the middle of the intersection. Instead they let each other drive first. This increases the overall driving safety and efficiency. However, the trained network does not generalizes well on new untrained scenarios. To achieve an applicability to different scenarios, the agent needs to be trained significantly longer and on many different routes. Since the intersection scenario in this case had the purpose to demonstrate the functioning of the implemented cooperative approaches it was only trained on one single scenario.

MontiSim supports the generation of randomized scenarios during training to achieve an applicability to different and unknown scenarios, as was used for training of the isolated agent. Training on randomized routes focuses on driving unknown routes rather than on cooperation between vehicles, since they meet irregularly when randomized and thus learning cooperation is more tedious. To illustrate that the implemented toolchain can also achieve robustness to different scenarios two additional randomized trainings with two vehicles, on 3000 and 5500 episodes, with the self-play approach were performed. A part of the city of Aachen is chosen as the training map and a new scenario for both vehicles is generated on the map in each episode. For 3000 episodes the policy has an average reward of -39850 and for 5500 episodes of +30787.5. This indicates a training success on randomized scenarios and underlines that longer training increases the performance. The results are tested on the map of Aachen with random and therefore unknown routes, which on average lead to a similar reward as the policy during training. Moreover, the self-play approach outperforms the

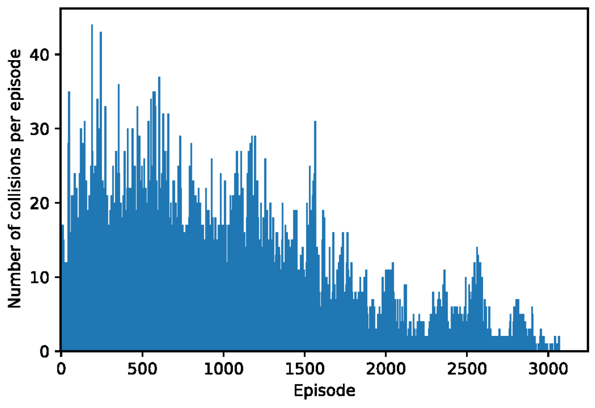

Fig. 6. The collisions per episode for the self-play approach.

min-step approach not only in the rewards received. It also decreases the computation time compared to the mini-step approach. With the mini-step approach, the more cars the more training steps per episode had to be calculated. In the self-play approach, this problem does not occur, since only one car is trained in each episode. Furthermore, the execution interval of the self-play agent can be adjusted depending on the number of trained cars, which means that there is no significant time overhead for a higher number of cars.

In addition, with the self-play approach all network architectures are compatible. This is not the case for the mini-step approach, where recurrent neural networks do not work. The reason for this is, that recurrent neural networks rely on the previous state for some computations. Here, however, the state does not belong to the same but to the previous vehicle.

VI. Conclusion

In this work we presented a holistic simulation toolchain for the multi-agent reinforcement learning of autonomously driving vehicles by extending MontiSim. We compared different modes of learning including a standard isolated, a mini-step and a self-play approach. The learning models have been implemented in the MontiSim simulator and evaluated successfully based on intersection scenarios. Thereby, we showed the success of MontiSim as a training environment on the one hand, as well as the fact that the self-play approach not only outperforms the mini-step approach in terms of rewards, but also decreases the needed computational resources.

References

[1] B. Wymann, E. Espié, C. Guionneau, C. Dimitrakakis, R. Coulom, and A. Sumner, “Torcs, the open racing car simulator,” *Software available at http://torcs. sourceforge. net*, vol. 4, no. 6, p. 2, 2000.

[2] N. Gatto, E. Kusmenko, and B. Rumpe, “Modeling Deep Reinforcement Learning Based Architectures for Cyber-Physical Systems,” in *MODELS 2019. Workshop MDE Intelligence*, September 2019, pp. 196–202.

[3] K. W Axhausen, A. Horni, and K. Nagel, *The multi-agent transport simulation MATSim.* Ubiquity Press, 2016.

[4] E. Kusmenko, A. Roth, B. Rumpe, and M. von Wenckstern, “Modeling Architectures of Cyber-Physical Systems,” in *ECMFA'17*, ser. LNCS 10376. Springer, July 2017, pp. 34–50.

[5] E. Kusmenko, B. Rumpe, S. Schneiders, and M. von Wenckstern, “Highly-Optimizing and Multi-Target Compiler for Embedded System Models: C++ Compiler Toolchain for the Component and Connector Language EmbeddedMontiArc,” in *Conference on Model Driven Engineering Languages and Systems (MODELS'18).* ACM, October 2018, pp. 447 – 457. [Online]. Available: http://www.se-rwth.de/publications/Highly-Optimizing-and-Multi-Target-Compiler-for-Embedded-System-Models.pdf

[6] E. Kusmenko, S. Nickels, S. Pavlitskaya, B. Rumpe, and T. Timmermanns, “Modeling and Training of Neural Processing Systems,” in *MODELS'19.* IEEE, September 2019, pp. 283–293.

[7] 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,” *arXiv preprint arXiv:1509.02971*, 2015.

[8] E. Kusmenko, S. Pavlitskaya, B. Rumpe, and S. Stüber, “On the Engineering of AI-Powered Systems,” in *ASE'19. Workshop SE Intelligence.* IEEE, November 2019, pp. 126–133.

[9] F. Grazioli, E. Kusmenko, A. Roth, B. Rumpe, and M. von Wenckstern, “Simulation Framework for Executing Component and Connector Models of Self-Driving Vehicles,” in *Proceedings of MODELS 2017. Workshop EXE*, ser. CEUR 2019, September 2017.

[10] J. C. Kirchhof, E. Kusmenko, B. Rumpe, and H. Zhang, “Simulation as a Service for Cooperative Vehicles,” in *MODELS 2019. Workshop MASE.* IEEE, September 2019, pp. 28–37.

[11] C. Frohn, P. Ilov, S. Kriebel, E. Kusmenko, B. Rumpe, and A. Ryndin, “Distributed Simulation of Cooperatively Interacting Vehicles,” in *ITSC'18.* IEEE, 2018, pp. 596–601.

[12] M. Haklay and P. Weber, “Openstreetmap: User-generated street maps,” *IEEE Pervasive computing*, vol. 7, no. 4, pp. 12–18, 2008.

[13] J. C. Kirchhof, E. Kusmenko, J. Meurice, and B. Rumpe, “Simulation of Model Execution for Embedded Systems,” in *MODELS 2019. Workshop MLE.* IEEE, September 2019, pp. 331–338.

[14] M. Hoppe, J. C. Kirchhof, E. Kusmenko, C. Y. Lee, and B. Rumpe, “Agent-Based Autonomous Vehicle Simulation with Hardware Emulation in the Loop,” in *2022 IEEE Intelligent Vehicles Symposium (IV'22).* IEEE, June 2022, pp. 16–21. [Online]. Available: http://www.se-rwth.de/publications/Agent-Based-Autonomous-Vehicle-Simulation-with-Hardware-Emulation-in-the-Loop.pdf

[15] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng *et al.*, “Ros: an open-source robot operating system,” in *ICRA workshop on open source software*, vol. 3, no. 3.2. Kobe, Japan, 2009, p. 5.

[16] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel *et al.*, “Mastering chess and shogi by self-play with a general reinforcement learning algorithm,” *arXiv preprint arXiv:1712.01815*, 2017.