3DCV - Critical Driving Scenarios

Generate critical driving scenarios in CARLA simulator and apply imitation learning to train a neural network

Christopher Klammt Tobias Richstein Julian Seibel Karl Thyssen

***Abstract*—In this project report, we present an approach for generating new scenarios for supporting autonomous driving tasks in critical situations using the CARLA simulator along with it's Scenario-Runner. Our approach consists of generating new XML-based OPENScenario files which can be interpreted by the Scenario-Runner. Based on already present basic Scenarios, we apply a variety of changes to attributes which are crucial for different sensors of an autonomous car. We considered external factors like weather, road conditions, time of day, color and models of cars, cyclists and pedestrians. In a second step, we transform the scenarios to different places in the same town and to different towns. To ensure the novel generated scenarios are valid, we consider different sanity checks as well as a specific verification to avoid duplicating scenarios. Also, we use the generated scenarios for training a model using imitation learning. TODO..**

***Index Terms*—CARLA simulator, Scenario generation, Critical driving scenarios, Imitation learning, Autonomous driving**

I. Introduction

Autonomous driving vehicles are not just an idea for the more distant future, but rather a very current topic. Not only Tesla, the company that dominates the news in this regard, is showing how far autonomous driving has come in the last years. One crucial issue still lies in providing a stable and safe behavior in critical situations, especially in which traffic participants behave unexpectedly. The problem is that to robustly learn the appropriate behavior for these specific critical scenarios and to generalize using supervised learning a large amount of training data is needed.

In order to tackle this problem, our paper describes the development of a generator for such critical driving scenarios in the CARLA simulator. These critical driving scenarios are mainly inspired by the report **NHTSA:PreCrashScenarios** [**NHTSA:PreCrashScenarios**] for the National Highway Traffic Safety Administration in the United States. These contain scenarios such as avoiding an obstacle, lane changing with oncoming traffic or crossing traffic running a red light at an intersection (as shown in Figure 1).

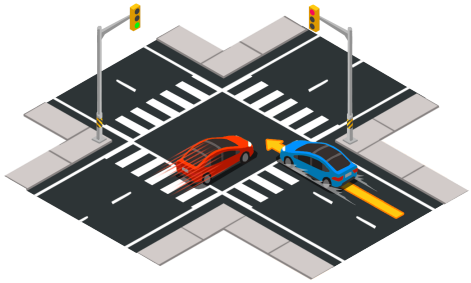

Fig. 1: NHTSA scenario: Crossing traffic running a red light at an intersection [**CARLAChallenge:Scenarios**]

Furthermore, after generating these different critical driving scenarios we utilize them to learn a robust model. To do so imitation learning is used, in which an expert demonstrates the desired behavior in each critical situation and is then adopted by the model.

II. Related Work

A. CARLA

As presented by **Dosovitskiy17:CARLA**, CARLA is an open-source urban driving simulator for autonomous driving research [**Dosovitskiy17:CARLA**]. It enables handling different use cases within the general problem of driving, such as learning driving policies or training perception algorithms. To control the simulation, e.g. changing weather, adding cars or pedestrians, an API is available in Python and C++. CARLA consists of the simulator responsible for rendering etc. as well as multiple components, for example a traffic manager controlling the vehicles or a component handling the sensors.

B. Critical driving scenarios

Another component that works in unison with the CARLA simulator is the ScenarioRunner

[**CARLA:ScenarioRunner**]. This is a module that makes it possible to define various traffic scenarios and execute them in the CARLA simulator. These scenarios can be defined using Python or the OpenSCENARIO [**OpenScenario**] standard. The ScenarioRunner already contains some predefined Scenarios which are based on the critical driving scenarios as described in **NHTSA:PreCrashScenarios** [**NHTSA:PreCrashScenarios**].

C. Generating data in simulators

In order to generate data it first is necessary to identify parameters that are to be adjusted and that should vary across the different entities. The data generation can then be approached in different ways. One possibility is to use a random statistical distribution of these parameters.

Another way to go about data generation is to use learning-based methods to adjust the parameters of the simulator. Such an approach was chosen by **DBLP:LearningToSimulate** and published in **DBLP:LearningToSimulate** [**DBLP:LearningToSimulate**]. They proposed a reinforcement learning-based method to adjust the parameters of the synthesized data to maximize the accuracy of a model trained on that data. A quite similar approach was chosen by **DBLP:Meta-Sim** [**DBLP:Meta-Sim**] called Meta-Sim, which learns a generative model of synthetic scenes and modifies the attributes using a neural network.

D. Imitation learning

To learn a model based on labeled data some form of supervised learning is generally applied. **Chen:LearningByCheating** propose a method called **Chen:LearningByCheating** [**Chen:LearningByCheating**]. This is a two-stage method, in which first an agent is trained using privileged information. In the second stage, the privileged agent acts as a teacher that trains a purely vision-based agent.

Another approach is **Toromanoff˙2020˙CVPR** [**Toromanoff˙2020˙CVPR**]. Here reinforcement learning is utilized to learn an optimal behavior policy based on a reward-function, effectively punishing wrong behavior such as leaving the track or running over pedestrians.

To train and manage the trainings of imitation learning networks jointly with evaluations on the CARLA simulator **felipecode:coiltraine** [**felipecode:coiltraine**] can be used.

III. Method

A. Generating new scenarios through ScenarioRunner

In general, we identified the two possible ways to create new scenarios that are supported by the ScenarioRunner. The first option consists of using the Python based API of the ScenarioRunner library to define the dynamic behavior of every actor in combination with a `xml`-based configuration file, which can be used to initialize different parameters for a predefined python based scenario.

The second option is to utilize OpenSCENARIO-based configuration files. OpenScenario is a `xml`-based file standard for describing dynamic contents in driving simulation applications [**OpenScenario**]. These configuration files differ from the former approach mainly in one aspect. For defining OpenSCENARIO based scenarios, there is no need to create the python-based scenario description. Therefore OpenSCENARIO can be used to generate all configurations for executing a scenario through the ScenarioRunner library combined at once.

Both the introduced options come with advantages and disadvantages. For our decision process, we identified different criteria which we would expect fulfilled for the approach we want to work with. The first criteria is *generalization*. We would expect that the approach can be well generalized in terms of extracting an abstract process which can be applied to all the different formats a new scenarios could possibly have. The second criteria is *the number of directly accessible and adjustable parameters*. Since we aim to provide a scenario generation process which is capable of generating as many scenarios as possible, we want to achieve a high number of adjustable parameters for providing a large combinatoric basis. The third and last criteria we considered is *simplicity* of the overall concept. Reducing the complexity and following the KISS (Keep it simple, stupid)-Principle for achieving different advantages in further developments such as extensions and maintenance.

Finally after evaluating both the options for generating new scenarios with respect to the defined criteria, we concluded to use the OpenSCENARIO-based approach. Overall we only see advantages by using the openSCENARIO standard as our final way for generating new scenarios. In comparison to the Python-based scenario generation, openSCENARIO can be better generalized because of its approved standard and fixed schema. Generating Python-based scenario definitions would bring a much larger effort. In addition, the `xml` configuration needs to be generated as well for every Python-file,

where multiple xml configuration could be used with one Python implementation. OpenSCENARIO-based files are including all the necessary data combined in one file. Therefore the selected approach provides an easy file-generation process while maintaining all the information necessary for describing one scenario. Considering the second criteria of direct accessible and adjustable parameters, with OpenSCENARIO-files all possible parameters which are crucial for a scenario are directly accessible and values can be changed easily. All the described differences can be transferred to the last and third criteria of following the KISS-principle. It might be that a OpenSCENARIO file alone has a more complex structure, but nevertheless, it would be much more complicated to generate generic Python-files for defining new Scenarios. Since it was declared as a standard, we are much more open for further developments even by using defined critical driving scenarios from different simulators.

B. Design and Implementation

With the decision of using OpenSCENARIO-files, we designed a generator tool which is capable of changing a variety of parameters that are decisive for the underlying problem of critical driving scenarios. Our approach utilizes the already present basic scenarios for critical driving situations of the ScenarioRunner. Since it is of tremendous effort to develop an abstract process to generate scenarios which state a completely new critical situation by changing the overall behavior of all actors, we first decided to use the five predefined scenarios, which will be called *basic scenarios* in the further course of this report. Those include situations of a crossing bicycle, changing weather conditions, following a leading vehicle, pedestrian crossing as well as a lane changing scenario.

We consider a scenario as new, if its sufficiently distinguishable for example by if single parameter value differs from all of the already generated scenarios. With this definition we can generate thousands of new scenarios by providing one base scenario. With the number of changeable parameters in combination with all possible values, it is of negligible probability that the exact same scenario will be generated twice.

To generate new values for selected parameters, we use the schema file for openSCENARIO-files which comes along with the ScenarioRunner tool. We extract mainly attribute names of tags we considered changeable in the context of this project as well as their data-types and valid values if the data-type is categorical.

Therefore we extract available information for example of the `<weather>`-tag, which has an attribute `cloudState` that only accepts values of a predefined enum type. Our generator would extract all possible values which is in the mentioned case a set of [”`free`”, ”`cloudy`”, ”`overcast`”, ”`rainy`”, ”`skyOff`”]. With the information of all attributes and their data-types and the basic scenario as template, new scenarios are generated. We also extract the information about the map, the scenario plays in. This is a prerequisite for extracting values of pedestrian and vehicle types from the CARLA Python API, since the valid and possible values of those attributes depend on the actual map. For all the tags which include attributes of categorical values we apply a random choice on all possible values. In the case of numeric attributes, we change the values based on the set values in a range of [-100%, +100%]. For every base scenario which is accessible, the user can specify how many additional scenarios should be created. After every new scenario, we use a hash value based comparison on the document to execute a duplication check such that we avoid the generation of the exact the same scenario twice. The overall generator-flow is shown in Figure 2.

For the internal representation we use a dictionary based attribute tree structure which represents the tree structure of the OPENScenario schema file for selected attributes defined as changeable, where every leaf represents a possible value if its a categorical type.

C. Adjustable parameters

CARLA and the ScenarioRunner enable us to adjust quite a few parameters, so that the generator can create a lot of diverse settings. In theory we have access to all possible parameters by the ScenarioRunner tool which are contributing to a single scenario. Since most of them are interventions into the semantics of a scenario, we propose to first change parameters which are not changing the overall actor behavior. This includes the type and model of the car, which is steered by the learned agent, but more importantly of each and every other car. This is important so that the model generalizes well, even if cars of different size, shape and color are in it’s field of vision resulting in different RGB patterns perceived by the cars sensors. Another parameter, that can be changed is the speed of the cars, which also has an impact on the behavior in critical situations. This setting is not limited to cars, also available bicycle and pedestrian types are used in similar manner.

Moreover the weather plays an important role, as it drastically changes what is perceived by sensors and

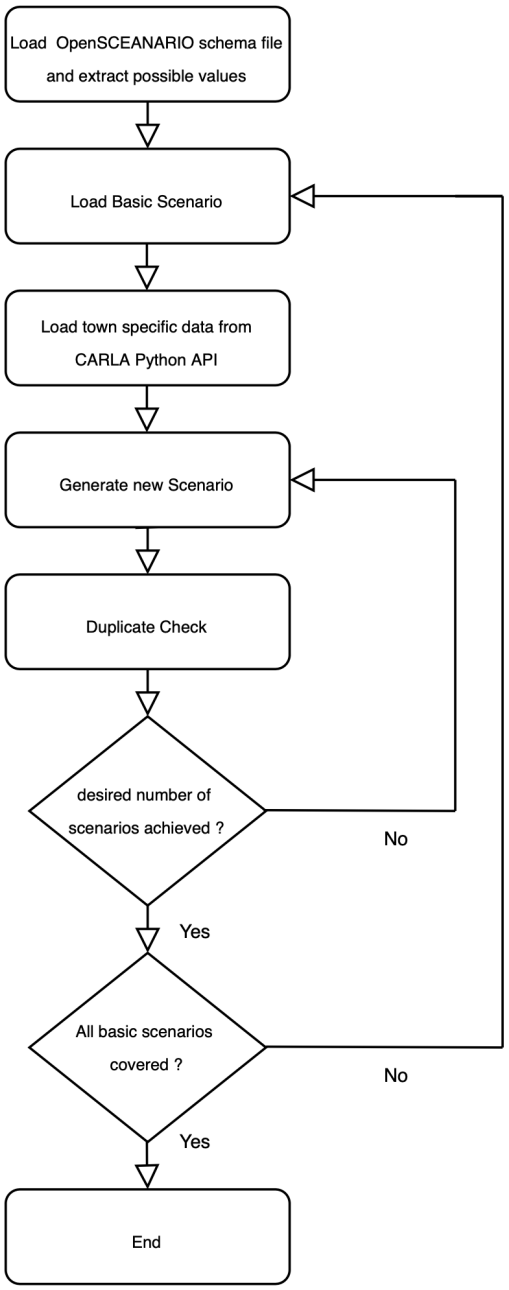

Fig. 2: Flow-Chart of the scenario-generator.

also alters the effects of a certain behavior. For example precipitation does not only impair the video of the camera, but additionally lengthens the breaking distance. These variables can all be adjusted rather easily, by just randomly picking a value from the set of possible values. Similar, we are feasible of adjusting all available attributes for sun and fog tags.

In addition, we generate a random time of day for every new scenario to cover also an increasing danger in situations for example in night, where pedestrians and bicycles are harder to detect.

To also simulate a variety of different road conditions, we include a random generated friction scale factor for every new scenario. Those factors can be influenced by many external effects like oil spilled on the road.

However, these attributes are independent from the original spawn-position of all actors. As an additional requirement, the generation of new scenarios should include a relocation of the scene to another suitable place in the same map or even in a different map. For this, we first analyze the initial situation by fetching the closest town specific waypoints based on the initial actor position. To get information about the scenario circumstances we use a simple approach to detect whether the scenario takes place on a intersection by calculating and comparing the distances between the actors and the hero (ego-vehicle) actor to the next intersection in driving direction. If the latter distance is less, we conclude that the scenario must take place at an intersection. This information is valuable for shifting the scenario to another position in the same or different town. We do this with respect to all the conditions given by the initial scenario for instance retaining distances between all actors.

If we conclude that the scenario takes place at an intersection, we fetch all possible junctions using the Python API of CARLA for a specific map. The closest junction in driving direction is used as the scenario anchor.

For some scenarios, it is typical that actor positions like for pedestrians or cyclists are not considered as waypoints since they spawn off-road. If we detect such a deviation, we calculate an offset between the actual actor position and the nearest waypoint. To apply the correct orientation to a new position p^{new}, we rotate the initial position of the actor $p = (p_x, p_y, p_z)$ subtracted the nearest waypoint $w = (w_x, w_y, w_z)$ with respect to the new orientation of the road $\Delta yaw = w_{yaw} - p^{new}_{yaw}$ using a rotation matrix with the difference in x direction $p_x - w_x$ and y direction $p_y - w_y$ indicated with Δx and Δy respectively:

$$p_{new} = \begin{pmatrix} cos(\Delta yaw) & sin(\Delta yaw) \\ -sin(\Delta yaw) & cos(\Delta yaw) \end{pmatrix} * \begin{pmatrix} \Delta x \\ \Delta y \end{pmatrix} \quad (1)$$

The considered distances and positions are illustrated in Figure 3 exemplary in the cyclist crossing scenario.

Not all positions calculated will work on different maps. Therefore we include two sanity checks when calculating new actor positions.

Fig. 3: Example of used distances and positions for transferring a critical cyclist crossing scenario to another intersection.

The part of the configuration, which is more difficult to vary is with regards to further actors, such as pedestrians and other vehicles. These have to be spawned somewhere and a behavior needs to be defined. CARLA provides a list of possible spawn points across a specific map, but if we just choose these randomly, we cannot be sure, that the spawned car has an impact on our scenario. Most importantly, the behavior of the car, which characterizes the critical scenario needs to well defined. But apart from that, we also want vehicles in the background, that drive around somewhat automated.

D. Limitations of scenario generation

During our research, we found some limitations of the proposed approach for generating new scenarios. First, we experienced no determinism in the driving behavior and routing of the CARLA autopilot. Each map provided by CARLA has certain probabilities assigned for each situation the driver agent has multiple routing options. Therefore it is not granted that the autopilot will take the right route the scenarios provides. Using the cyclist crossing scenario as example, the autopilot should make a right turn to trigger the scenario and therefore run into a dangerous driving scene. But with the probabilities assigned, it could be the case that the autopilot turns left for a single run such that the scenario is not triggered or the driver is not confronted with the dangerous scene. Nevertheless, an agent could be used which is capable of following a specific route given by the scenario file.

Second, at this time it is not possible to gather semantic information namely concerning actor and driving behavior. In detail, the behavior of the base scenarios will be adapted without any changes.

The last limitation of our approach is the dependency to previous defined OPENScenario basic scenarios. Since we use a predefined critical driving situation as template, we are limited to currently six available scenarios. However, we think of OPENScenario as a future-proof standard including a growing amount of available scenarios which can be used to create a variety of diverse scenarios.

E. Imitation Learning Approach

To test the generators' ability to generate diverse critical scenarios we can train a model on generated critical scenarios. We can then measure the performance of this model on a test set to see its robustness to new generated scenarios. Using the existing functionality of the ScenarioRunner to play through our generated scenarios, we can gather the 'expert' driving data from the ego vehicle of the generated scenario. Using this data we are then able to train the vision agent using the Coiltraines training capabilities.

The expert agent, which should be able to solve the critical scenario in the optimal way, uses information gleamed from the Carla Client. This includes precise weather readings, proximity measurements to other NPC actors and information pertaining to the current traffic situation as well as details of other NPC actors' intentions and actions. This allows the expert to solve the scenario safely. The model to be trained however cannot be privy to any of this privileged information. Therefore the expert gathers data using an RGBA camera sensor, mounted to the front of the ego car selected by the scenario generator. Though multiple sensors such as side cameras or LIDAR were also possible but to reduce the size of the collected data as well as the time required for training we opted into a single camera sensor. The only other data that was collected and provided to the model as part of its state vector was the steering, brake and throttle values.

The issue of the frequency of measurement is again a question of data volume and training time vs precision of the model as the frequency of data collection (dependent on resolution) could vary between 0-60fps depending on the hardware used. we opted for 3 collections per second with intent to vary this based on the effectiveness of the model trained on such sparse readings.

IV. Experiments

A. Imitation Learning Execution and Issues

Though we set out to preform the learning as outlined in the approach however ran into some issues that prevented execution. A lack of GPU hardware made the gathering of expert data an overwhelming task for our available resources, even at the low resolution of 200x300 we had seen in testing allowed for reasonable fps. This prevented any parallel data collection through a docker application or similar which caused gathering 500MB of training data to take a minimum of 5 hours with 50 hours of driving time in the Coiltraine test sets equating (allbeit at a higher resolution) to roughly 50hours of driving data.

Of course less data can be used, with 10 critical scenarios to be learned however and the span of possible changes in each one, which it is our aim to test, it is unrealistic to assume that 10 driving hours which our resources allowed us to gather would enable the model to achieve a level of competency. This assumption is based on the testing of pre-trained Coiltraine sample models at a much higher resolution with 3 RGBA sensors accompanied by LIDAR data.

B. Training a Pre-trained Coiltraine Model to Augment with Critical Scenario Data

Falls es klappt.

C. Evaluation a Pre-trained Coiltraine Model on Critical Scenarios

Falls es klappt.

V. Conclusion

To conclude this report, we try to answer three research questions based on the outcome of our experiments.

• Can we generate critical scenarios such that a model trained on these scenarios achieves robustness against critical scenarios?

• How many critical scenarios have to be represented in the training distribution in order to achieve robustness against critical scenarios?

• Does robustness against a certain type of critical scenario transfer to a different type of critical scenario?

+ future outlook

- erweiterung des generators bzgl. neue actors spawnen die einfach rumlaufen/fahren
- behvior ändern