Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Systematic testing of artificial intelligence-based lanekeeping systems with genetic algorithms

Scientific Students' Association Report

Author:

Balázs Pintér
Bence Puscsizna

Advisor:

dr. András Vörös
dr. Kristóf Marussy
Attila Ficsor

2023

Contents

Kivonat

A mesterséges intelligencia fejlődésének köszönhetően egyre több fejlett vezetéstámogató funkció érhető el a gépjárművekben. A sávtartó rendszerek különösen sokat segítenek a biztonságos vezetésben, hiszen segítik a jármű sávon tartását és figyelmeztetik a vezetőt, amennyiben veszélyes sávelhagyási esemény történik. Jól látható, hogy ezen rendszerek akár kritikus beavatkozást is végezhetnek, továbbá ahogy a vezetők egyre jobban megbíznak bennük, egyre kritikusabb a helyes működésük, hiszen az esetleges hibáik balesetekhez vezethetnek.

A sávtartó funkciók a mesterséges intelligencián belül a mélytanulás (deep learning) megoldásaira épülnek, azaz a tanításukhoz nagy mennyiségű adatra van szükség, amely alapján a neurális háló megtanulja a fontos információkat és utána képes lesz működés közben a vizuális információ feldolgozására. Egy ilyen megoldásban azonban több helyen is hiba kerülhet a működésbe. A tanító adathalmaz és a tanulás során is előfordulhatnak problémák, például, ha nem elég diverz a tanító halmaz vagy ha a neurális háló rosszul általánosít. Emellett, mivel emberek írják ezeket a programokat is, magukban a neurális háló szoftverekben is lehetnek hibák. Ezeket a hibákat fontos lenne lehetőleg már a tervezési/fejlesztési időben megtalálni.

Munkánk során egy olyan megoldást fejlesztettünk, amely képes a neurális háló alapú sávtartó rendszerek hibáit és gyengeségeit szisztematikusan felderíteni. Módszerünk több különböző technikát kombinál: OpenDrive formátumú teszteket generálunk, amiket Blender segítségével feldolgozunk, és háromdimenziós modelleket hozunk létre. Ezeket utána különböző szimulátorokban, mint például a Carla, fel tudjuk használni, és akár bonyolultabb közúti szituációkat is tudunk vele modellezni. A szimulátorokkal realisztikus, és diverz tesztképeket tudunk előállítani. Ezeknek a képeknek a feldolgozása után, a sávkövető rendszer kimenetét felhasználva genetikus algoritmusok segítségével javítjuk a bemenő paramétereit a tesztgenerálási folyamatnak. A teszteredmények visszacsatolásával segítjük a kereső algoritmusokat, hogy minél nehezebb teszteseteket állítsanak elő, amiknél a sávkövető rendszer nem teljesít megfelelően.

A fejlesztés során igyekeztünk az iparban is használt nyílt forráskódú technológiákat felhasználni, továbbá a megközelítésünket egy, a közúton is használt sávtartó automatika vizsgálatára is felhasználtuk, amely segítségével kiértékeltük a megközelítésünk alkalmazhatóságát.

Abstract

Thanks to the development of artificial intelligence, more and more advanced driver assistance functions are available in vehicles. In particular, lanekeeping systems can help safe driving by helping the vehicle to keep in lane and warning the driver if a dangerous lane departure event occurs. It is clear that these systems could perform critical interventions and, as drivers become more confident in them, their correct operation becomes more critical, as any failure can lead to accidents.

Lane-keeping functions are based on artificial intelligence solution, called deep learning, i.e. they require large amounts of data to be taught, which the neural network uses to learn the relevant information and then becomes able to process the visual information during operation. However, such a solution can fail in several areas. Problems can occur in the learning data set and in the learning process, for example if the learning set is not diverse enough or if the neural network generalises incorrectly. In addition, since humans write these programs, there can be errors in the neural network software itself. It would be important to find these errors preferably at design/development time.

In our work, we have developed a solution that can systematically detect the faults and weaknesses of neural network-based lanekeeping systems. Our method combines several different techniques: We generate tests in OpenDrive format, process them using Blender, and create three-dimensional models that can then be used in various simulators such as Carla, and with them model complex driving scenarios. With the simulators, we can generate realistic and diverse test images, which, after processing, we can use the output of a lanekeeping system to improve the input parameters of the test generation process using genetic algorithms. By providing feedback on the test results, we help the search algorithms to generate test cases that are difficult, i.e. where the tracking system does not perform well.

We used open source technologies used in the industry and have also used our approach to test a lane-keeping system used on public roads to evaluate the applicability of our approach.

Chapter 1

Introduction

Advanced deep neural network (DNN) based applications have become a part of our lives in recent times. They can be found everywhere, from entertainment to critical systems, like advanced driver assistant systems (ADAS) or medical applications. Similarly to traditional software, DNNs can also have bugs and behave unexpectedly, and such failures can lead to critical errors. DNNs do not contain the working logic explicitly, making it hard to interpret the model. To ensure the correct behavior and gain people's trust, thorough testing is required.

For lane-keeping assist systems (LKASs), it's essential to evaluate their performance across many roads with different curvatures, in different visibility conditions, and different traffic situations. Conducting such tests in the real world is often problematic: it is less scalable and more expensive than doing it in a synthetic simulated environment. Constructing a synthetic environment offers huge freedom. Testers and automatized test-case exploration methods can generate roads with arbitrary curvature, weather conditions, traffic scenarios, and more. This flexibility is valuable when we want to find the possible vulnerabilities of an ADAS in a broad input space. This could help to ensure that many different road layouts or traffic scenarios are covered. In order to achieve diverse test data, we need some guarantees.

Providing code coverage-like metrics is not possible for DNNs. Although domain experts could give some test cases, for such systems, this is not a sustainable solution; they are immensely complex. If the whole behavior could be formally described by domain experts, there would be no need for the system under test, as it could be replaced by the test oracle.

Intuitively, using some coverage criteria could measure the diversity of the test data from a certain aspect, making the test result more trustworthy, but defining coverage for DNN testing is not straightforward. Numerous coverage criteria are defined: neuron coverage [21, 30], which turned out to be not a good metric, as [11] showed because neuron coverage can be easily manipulated, and it does not necessarily indicate a well-tested model. There are other coverage techniques that try to describe the diversity of the input test set using abstract representations of the input space. Using logical scenarios [19] for traffic situations (instead of concrete, fully specified scenarios) has the promise [18] to give coverage guarantees for the working logic of the system.

The suggested methods can work for describing test traffic scenarios for an ADAS in general, but lane detection primarily relies on the shape of the road. To achieve a diverse test set, we can use various distance and diversity metrics for road geometries, like Jaccard similarity index [10], Iterative Levenshtein distance [23], and various other road features (e.g., curvature, complexity, and direction coverage)[24].

Testing vision-based systems and providing graphical input for them requires a lot of computational resources. Search-based test input exploration approaches have been utilized in the past [16, 17] to efficiently discover the broad input space while finding diverse and challenging inputs significantly faster than random sampling or other methods. Some of the approaches even have built-in distance mechanisms (e.g., NSGA-II [6] with crowding distance). Such algorithms keep the other generated test inputs in their working memory, and if they find multiple potential new test inputs with similar performance, according to a pre-defined metric, the algorithm will choose to keep the new input that deviates the most from the existing test cases, providing diverse test set by its nature.

The goal of our Scientific Students' Association Report is to systematically test the lane detection system of a widely used production ADAS with a genetic algorithm. In order to achieve this

- we created a framework where LKASs and their components can be tested on a generated road. This framework is modular, and with slight modification, it can be applied to test many different LKAS or lane detection models.
- We can describe the roads and lanes parametrically and retrieve standard road description files, 3D objects, and images. These formats can be used to ingest the generated road into a simulator or just to retrieve images of the road.
- We run genetic algorithms to find the weaknesses of a lane detection system.

Chapter 2

Background

2.1 DNN-based Lane Detection Approaches

DNN-based lane detection methods can be classified [25] into four main approaches. Some approaches are considered more accurate, while others are more lightweight. Lane detection is usually part of a bigger functionality, like lane-keeping, but standalone computer vision models also exist that can identify lane lines, among other objects. Lane detection models all have visual inputs, but many lane detection components integrated into ADAS consume multiple frames: images from different angles or a sequence of images. Depending on the task, performance, or accuracy requirements and of course the expected input of the other components in the ADAS, lane detection can be realized with various approaches.

2.1.1 Segmentation Approach

The segmentation approach treats lane detection as a pixel-wise classification task, determining whether each pixel belongs to a lane line or not. This approach has been widely used in recent lane detection methods and has been adopted by industry players, such as Tesla [14]. Despite its high accuracy, the segmentation approach suffers from increased computational and memory costs and requires a postprocessing step to extract lane line curves from the pixel-wise classification results.

2.1.2 Row-wise Classification Approach

Row-wise classification marks exactly one (or zero) pixel as a lane line in each row of the image. This approach, introduced in [22], leverages domain-specific knowledge, assuming that lane lines follow the longitudinal direction of driving vehicles and not be so curved to have more than two intersections in a row of the input image. By formulating the lane detection task as multiple row-wise classification tasks, this approach can reduce the model size and computational requirements while maintaining high accuracy. Initially, such methodologies required a post-processing step to extract lane lines, similar to the segmentation approach, but recently, more end-to-end solutions [33] have come out.

2.1.3 Curve-fitting Approach

The curve-fitting approach fits lane lines into parametric curves, such as polynomials or splines. The curve-fitting approach tends to have lower accuracy compared to other methods [25], and it may be biased towards straight lines [29] due to the dominance of straight lane lines in training data. However, it is essential to note that lower lane detection accuracy does not necessarily imply worse lane keeping or impaired driving assistant performance. This approach is used in OpenPilot [5], the open-source production driver assistance system, and is known for its lightweight computation, allowing it to run on smartphone-like devices.

2.1.4 Anchor-based Approach

The anchor-based approach represents each lane line as a straight proposal line (anchor) and lateral offsets from the proposal line. By exploiting domain-specific knowledge that lane lines are generally straight, this method can achieve high performance and low latency.

2.2 Testing vision-based systems

Program testing can be used to show the presence of bugs, but never to show their absence![1]

This is no different in the case of vision-based machine learning or artificial intelligence (ML\AI) models. There are many different vision-based systems; based on the system's goal, we can talk about image classification, object detection, semantic or instance segmentation, or more. These systems often perform critical tasks, e.g., determining if a mole is malignant or detecting lanes or pedestrians for an ADAS.

Testing vision-based systems is fundamentally the same as testing traditional software components: providing inputs for the system and evaluating its result with the help of a test oracle. However, complex and large neural networks or other image-processing models came with special challenges:

Robustness A common property of a DNN-based system is the lack of robustness, meaning that small changes in the input can drastically change the model's prediction. As [28] showed, input-output mappings for deep neural networks are fairly discontinuous. There are many other testing approaches and adversarial attacks that can reveal such weaknesses in the system. In [28], they added an adversarial filter. For the human eye, the picture remained essentially the same, but the model failed to predict the object in the altered picture. Another famous example is the one-pixel attack[27]; here, changing a single pixel or a smaller area in the input image made the model fail.

In our former scientific students' association report [8], we addressed this robustness on a higher, semantic level in the autonomous driving domain: we generated test traffic scenarios in a simulator and spawned different roadside objects. We found that introducing such objects in the images can distract the object detection model: generally, the accuracy of the model was lower on the images with the extra roadside objects compared to the baseline without them. To sum it up, there are plenty of examples of the fragility of DNN models: from adding seemingly irrelevant noise to mutating high-level features (e.g., object replacement), breaking the model's prediction is possible.

[1] Dijkstra, E. W. (1970). Notes on structured programming

Feature space Vision-based systems have large feature space since images are represented as an array of pixels, covering all possible combinations of the input is neither possible nor meaningful. Rather than attempting to test every pixel combination, some approaches recommend domain-specific abstractions. For instance, in the domain of autonomous driving, rather than focusing on every pixel in a traffic scenario, one might abstract the scene into cars and pedestrians and their abstract relations. This abstraction could be used to generate diverse test cases and can offer[18] some coverage guarantee in the level of the abstraction.

Lack of interpretability Vision-based models, due to their complexity and depth, can be notoriously hard to interpret. They consist of many layers, each with potentially millions of parameters, essentially making it a black box to humans. White box testing is barely possible, although internal neurons can be observed, and metrics like neuron coverage [21] can be used, these are not helping to describe the working mechanism of the model. There are models that have some mechanism that can be used for explanation, for instance, autoencoders [15], where a narrow layer in the middle of the model represents features in the image.

Scalability and efficiency Handling DNNs is resource-intensive. While training these models can be time-consuming, testing and evaluation come with their own sets of challenges. This is especially true when evaluating the model's performance across diverse scenarios. Running each test case can require significant computational resources, but the real barrier is running so many tests that could potentially cover the input space extensively. To utilize the finite resources effectively, diverse test data is needed, but as shown in the previous paragraphs, this is also a great challenge.

Data challenges Generally, testing in the development phase often serves the purpose of detecting failures and immaturities of the system under test. The detected failures should be fixed, which in the case of DNN-s means retraining, applying some pre- or postprocessing step, or just changing the architecture of the model. Many studies [28, 3] showed that different architecture models tend to fail in a similar manner when trained with the same dataset. This shared pattern of failure suggests that the root of the problem may not necessarily lie in the architecture itself but rather in the training data. This would also suggest that adding more data to the training set, similar to the failed tests, could fix the retrained models' performance.

2.3 Genetic algorithms

Genetic algorithms (GAs) are useful tools for finding near-optimal solutions for multiple tasks with decent performance compared to naive or brute-force methods. These evolutionary algorithms were inspired by the process of natural selection and genetics: the process starts with the creation of an initial population for the potential solutions, then comes the iterated steps of mutation, selection, and crossover to generate solutions to optimization and search problems, once the solution set is good enough according to a fitness function or a termination condition is met, the algorithm stops. These are applicable to a wide range of problems. The concept of GAs is often introduced as a great and fast shortcut for the backpack problem (more formally known as the "Knapsack problem"), but beyond this, it has been successfully applied in various other ML tasks.

For lane detection tests, it can be used to generate the shape and curvature of the road on which the ADAS will be evaluated. Parameters for parametric cubic curves, or Bézier curves, can serve as parameters to optimize for the algorithm. GAs may not always provide the global optimum solution but rather a good enough or near-optimal solution that is adequate for many purposes, including ours, and can handle even larger feature spaces. There are many different algorithms for different purposes. In our research, we used a generic and widespread genetic algorithm: Non-dominated Sorting Genetic Algorithm II (NSGA-II) [6] and its python implementation, `pymoo`[4]

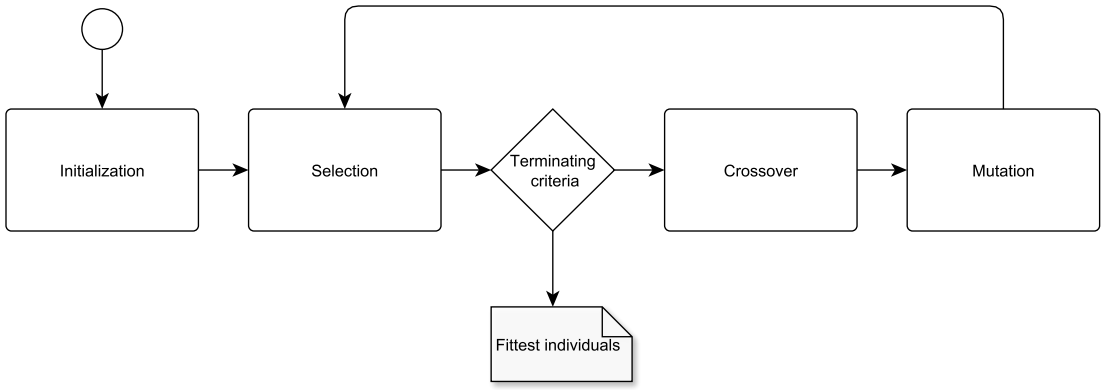


Figure 2.1: Genetic algorithm workflow

Initialization Initially, the algorithm starts with a set of (usually) random solutions called an initial population. It is a set of potential solutions to the problem, each encoded as a `chromosome` or `individual`. The random generation of this set should consider the domain-specific knowledge, which usually means that the parameters should be in a range that can be considered normal. This will result in a faster divergence and also more accurate output.

Fitness function The fitness function (or objective function) is what the genetic algorithm optimizes for. In the context of optimization problems, they are either minimization or maximization. The fitness function evaluates each individual in the population. It assigns a fitness score based on how well the solution addresses the problem at hand. The exact composition of the fitness function often varies, reflecting the specificities of the domain and the problem. For example, the Knapsack problem's fitness function is the

total value of the items in the backpack: this has to be maximized. For lane parameter generation, the fitness function can measure the average deviation of the prediction from the ground truth lane, weighted with distance-based decay: this is basically an error function. Certain algorithms aim to find optimal minimum values. In such scenarios, the formulation of the problem may need to be adjusted: if the error function is bounded, one approach is to subtract the original error function from this limit, effectively converting the problem into a minimization one. Alternatively, by multiplying the error function by -1, the problem can be just turned upside down if the algorithm can handle negative fitness values.

Population size The population size is critical for the GA's effectiveness. A small population may converge quickly but risks settling for a local optimum. Conversely, a larger population offers a broader search but demands more computational resources. Balancing this trade-off is crucial.

Problem representation: parameter count & type For a given problem, the input is the representation of potential solutions, called genotype or chromosome representation. The representation and handling of parameters may vary based on the nature of the problem's solution space. Parameters of a possible solution can be a binary array: for the backpack problem, assigning to each item if it is present (1) or not (0). Problems sometimes need more complex inputs, like finding control points for a curve can involve continuous values (floats) or integers: they all can be represented as bitstrings. Most of the time, this is what happens under the hood, but GA libraries offer high-level interfaces that can handle floats for better usability. It is also possible to combine both discrete and continuous value representations within a single problem definition.

Number of generations The number of generations determines the depth of the search. Too few might lead to a suboptimal solution, while too many might be computationally expensive without much improvement. To avoid the latter, early stopping or termination mechanisms are often employed.

Selection Selection acts as a survival mechanism, where individuals with better fitness scores have a higher chance of being chosen for the next generation. Techniques like simply choosing the best-performing individuals, roulette wheel selection, tournament selection, and rank-based selection can be employed. The chosen method can impact the diversity and quality of the selected individuals. An advanced technique that aims to maintain diversity among selected individuals is the crowding distance, introduced in NSGA-II. The crowding distance computes the distance between individuals in the objective space, and individuals with larger crowding distances are given preference since they are less crowded by their neighbors. This encourages the selection of solutions that are both high-quality and diverse.

Crossover Crossover, also known as recombination, is a method to combine the genetic information of two parents to produce one or more offspring for the next generation. In genetic algorithms, there are various ways to mix the information of two parents to make a new child. The easiest way is the single-point crossover: a crossover point is chosen in the parent data. It is like cutting two strings of the same size at a particular spot and then swapping the ends of these strings to make two new ones. Another similar method is

multi-point crossover; with this method, multiple cuts and swaps are performed. The most nature-like approach is uniform crossover. Instead of cuts, parent data is mixed based on a set pattern or ratio to gather the next offspring(s).

Mutation Mutation introduces random changes to an individual's traits, ensuring diversity and preventing premature convergence. The mutation rate, or the frequency at which mutation occurs, is a critical parameter that needs careful tuning. For binary array input, it can be simply flipping a random bit(s) with a certain probability. This analogy can be applied to numeric inputs [13]: a floating point number can be represented as a binary string (an array of bits), and doing the same, random bit flipping mutation on the floating point represented number will result in a good mutation: this has the possibility to change the value of a number significantly, meanwhile e.g. adding or removing number in an ϵ range is not able to change the magnitude of the number.

Termination Once mutation is complete, a new population is formed, which will undergo the same cycle of fitness evaluation, selection, crossover, and mutation. This iterative process is carried out until a termination condition is met, which could be reaching a maximum number of generations, achieving a satisfactory fitness level, or if there is no significant improvement in fitness.

For most of the problems, mature Python libraries provide good general steps: there is no need to implement mutation, crossover, or selection manually. The vital parts to define for a problem are the fitness function, the number of population, and the number of generations. The fitness function can consider many domain-specific information, and in our case, this also includes running the system under test with the generated input and then evaluating its prediction, but it could even include running simulation snippets if we would optimize for some dynamic property. For other parameters, there are rules of thumb for the optimal number of population and generation, but this can be influenced by the concrete problem. Hence, this should also be evaluated by running the algorithm with different configurations to find the most adequate parameters for a specific problem.

2.4 Description of roads and curves

Our research is based on the generation of appropriate tests. The tests should be easily and highly customizable in order to test the system under investigation with as many potential inputs as possible. The generated tests are in OpenDrive 1.6 format, which is a common format in the industry for describing different road networks. OpenDrive provides the possibility to create roads, intersections, and traffic rules so that complex traffic situations can be defined. At the moment, we only use the features needed to define a simple two-lane road, but in the future, this can be extended to include, for example, intersections. The geometries that can be used to define a path are as follows:

- Line
- Arc
- Cubic curve
- Parametric cubic curve

For roads, the geometry is interpreted in the plane (x-y), and the position (height) in the z direction is given by a slope profile, whose formula is:

$$elev(ds) = a + b * ds + c * (ds)^2 + d * (ds)^3 \tag{2.1}$$

In addition, the format also allows you to set the superelevation of the road, but this feature has not yet been used by us. The tests are generated by a Python script where the user can specify the desired geometry of the path.

The parametrization of the roads was done using two different curve types. For the generation itself, we used parametric cubic curves, but for the parametrization, we decided to use Bézier curves. The former was used because it is supported by the OpenDrive format, and the latter was used because it made the parametrization easier.

The parametric cubic curve that is used during generation can be described as follows:

$$u(p) = aU + bU * p + cU * p^2 + dU * p^3 \tag{2.2}$$

$$v(p) = aV + bV * p + cV * p^2 + dV * p^3 \tag{2.3}$$

As it can be seen in Figure 2.2, the two equations give us the u-v (x-y) coordinates separately. In order to make the curve's starting direction parallel to previous sections or

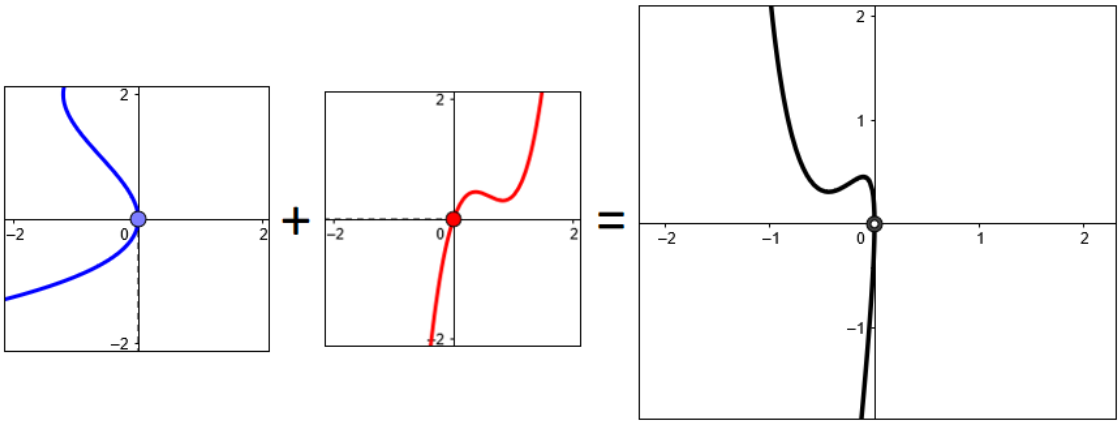


Figure 2.2: Parametric cubic curve

the camera, we had to determine the direction of the parametric curve. It can be achieved by examining the derivatives. The derivative of a cubic polynomial at a given point is the quotient of the derivatives of its component polynomials at the same point.

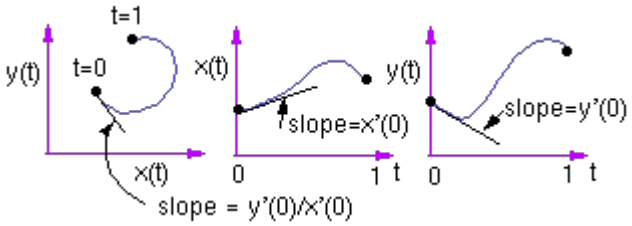


Figure 2.3: Derivative of the cubic curve[20]

Because from the parametric cubic curve's parametrization, it is harder to determine the shape of the curve before actually visualizing the geometry, we decided to use a cubic Bézier

curve instead. We chose this because both can be written down with a cubic polynomial equation, which means that they can be converted into each other using simple equations. The equation of the third-degree Bézier curve:

$$\mathbf{B}(p) = (1-p)^3 * \mathbf{P0} + 3 * (1-p)^2 p * \mathbf{P1} + 3 * (1-p) p^2 * \mathbf{P2} + p^3 * \mathbf{P3} \tag{2.4}$$

In order to use this new parametrization, we had to convert the Bézier parameters to parametric cubic polynomial parameters. To do this, the Bézier curve needed to be decomposed and sorted. It meant that Equation 2.4 had to be split into x-y(u-v) coordinates and then sorted so that the parameters of Equation 2.2 and Equation 2.3 can be determined. With this, we achieved that the curves were parametrized with Bézier control points but then generated with parametric cubic polynomial parameters.

Chapter 3

Related work

This chapter presents methodologies that either evaluate lane-keeping or ADAS components similarly to our approach or employ similar techniques, such as search-based methods, to generate valuable test cases or to find adversarial examples.

3.1 Identifying the hazard boundary of ML systems

A crucial aspect of testing deep neural networks (DNNs) is to identify the hazard boundary [26], the boundary in the input space beyond which the network's outputs become unreliable or incorrect. There are several approaches to this problem, including sampling- and search-based methods. DNNs describe incredibly complex mathematical functions, that can be fairly discontinuous [28], meaning that slight changes can drastically affect the model's prediction. This property of DNNs enables flexibility, but this could also result in failures or flaws in the prediction, for example discovering isolated counterexamples [27] or fault clusters [3], similar inputs, for which the failure of the model is significantly more than for other inputs. Even for well-trained, and thoroughly tested models. Images are complex inputs, and altering them pixel-wise is possible, but not always the most effective method. Plenty of approaches [30, 32, 9] utilize domain-specific, abstract descriptions of the input. Abstraction can be used for both the search-based and the sampling-based methods and has the promise, that the described data points are meaningful, and from the aspect of the test oracle. Determining the ground truth is much easier with such data, sometimes the abstract representation can even directly contain the ground truth itself.

3.1.1 Sampling-based Methods

Sampling-based methods aim to systematically sample the input space of a DNNs to understand its behavior and identify regions where the network's output becomes unreliable. The primary goal of these techniques is to cover as much of the network's input space as possible with a limited number of samples. Many different strategies exist to ensure diverse and representative sampling.

Random sampling The most straightforward technique is to randomly sample input data from the given input space. Though simple, it may not be effective in exploring rare or adversarial scenarios.

Adaptive Sampling Adaptive sampling techniques refine the sampling process based on previous outputs. If a region of the input space is identified as problematic, more samples are taken from that region.

3.1.2 Search-based Methods

Search-based methods seek to find the hazard boundary more guided, by iteratively exploring the input space and evaluating the network's performance. Techniques such as gradient descent or genetic algorithms are commonly used to search. This approach can be computationally intensive but is often effective at finding regions of the input space that lead to incorrect behavior.

3.2 Testing lane-keeping assist systems

Testing lane-keeping assist systems and their components with simulated or synthetic environments has gained significance in recent times. Recent studies [16, 2, 17] presented many different search-based testing approaches. In the study [16], researchers suggested GA-driven construction for roads, to test an automated lane-keeping system, similar to our work. Their road creation technique consists of generating seven control points for a Bézier curve, from which a road is subsequently synthesized, and finally running the system under test, in a simulator. Their method uses more control points, than ours (4), but their generating process can produce invalid combinations, like roads that intersect with themselves, or roads with too sharp curves. They have to verify and remove the malformed instances later. In opposition, our approach always generates valid roads, because of the pre-defined parameter constraints.

In [17] researchers compared a GA-driven test case exploration with random sampling. They tested ADAS in simulation, and they optimized to explore scenarios, with multiple actors, where the time to collision (TTC) was below a minimal threshold. In their study, the GA-based approach outperformed others, by producing critical scenarios with fewer test executions than both random testing and simulated annealing.

Another study [2] generated test cases, by combining GA with classification. They showed that an evolutionary search algorithm combined with decision tree classification creates more distinct (diversity) and critical test scenarios for vision-based control systems, compared to a baseline sampling, or a standalone genetic algorithm approach, without the classification step. First, classification models guide the search-based generation of tests faster toward critical test scenarios. Second, search algorithms refine classification models so that the models can accurately characterize critical regions, eventually discovering fault clusters in the input space.

Chapter 4

Overview of the approach

We created a modular, search-based testing workflow that can be used to evaluate lane detection systems, or by introducing an extra simulation step, even lane-keeping systems or other ADAS's dynamic behavior.

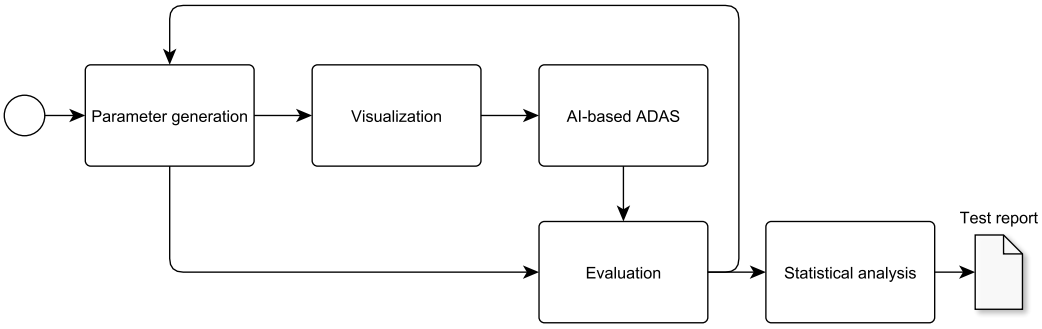


Figure 4.1: Workflow overview

The workflow is genetic algorithm (GA) based, describing the problem with parameters is a must so that we can generate roads to test the system with.

The visualization step is also crucial for vision-based systems. The input image is synthetic but should be close to real-world appearance. The challenges here were generating a 3D model of the road with correct curvature and texture. We also added basic surroundings, e.g., curbs, grass texture, and sky. For the extendibility and the simulation opportunity in the future, we were working with standard road description format and triangle mesh format, this way, importing the generated environment is possible into simulators like CARLA[7].

We created the whole workflow in order to test AI-based ADAS or their components. We chose to test openpilot's lane detection module, as testing a production ADAS deployed in thousands of cars seemed an interesting challenge.

The subsequent evaluation step uses the parameters as ground truth and compares the prediction of the AI-based ADAS to evaluate how accurate its detection was. Here, we defined the error metric based on suggestions from the literature [31, 25] in order to fit the lane detection result, but this could also be modified when we test a new type of ADAS. Based on the result of the evaluation step, the algorithm generates new parameters, optimizing to find challenging but diverse inputs for the system under test.

Finally, we analyzed the results for the outputs to understand what are the weaknesses of such systems. We analyzed the concrete parameters of the road, curvatures, the model's error, and the relationship between these values.

4.1 Parameter generation

Roads — and objects that are easily describable with a (set of) geometric objects in general — can be described with a few parameters: parametric cubic polynomials, Bézier curves, and other geometries, and their combinations can describe the shape of the central line of a road. Computer games, simulation software, or maps in general use such methods to describe complex shaped roads in a compact way. A simple road can have other parameters, like the number of lanes, but they could possibly represent other visual or semantic properties of the road as well.

This parametrization is a powerful tool, altering a few parameters can introduce a wide variety of different roads. For optimization problems, handling fewer parameters is an advantage, algorithms have to discover a smaller feature space.

For genetic algorithms or other search-based methods, developers must give the feature space beforehand. The number of parameters to explore, their type, and possible values should be specified.

On the other hand, complex polynomials, like parametric cubic curves, have a drawback: slightly changing a parameter can have an enormous effect on the actual shape of the road, as they are not necessarily independent of other parameters. Bounding the possible values of these parameters does not help in many cases: they can depend on each other, and determining if they create a meaningful parameter set can only be determined at runtime, with enormous computational overhead.

4.2 Visualization

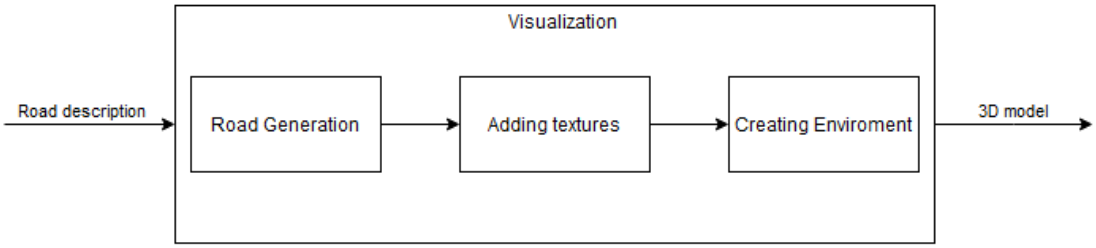


Figure 4.2: The process of visualization

In order to analyze the roads that we generated, we need a visualized version of them. To achieve this, we need to generate three-dimensional models of the roads and then take pictures of them. For this we used Blender because it is open source, has Python support, and is highly customizable. In Blender, we wrote a script that first processes the input files, which are basically road descriptions. After this, the script generates the road curves. With these curves and a road profile, Blender can use a sweep-like operation that will create the 3D shape of the roads. Next, the script applies textures to the different parts of the road to make it more realistic. The script also creates an environment which

also helps us achieve more realistic pictures in the end. The environment consists of a skybox and a ground so far, but in the future, we plan on extending it with different obstacles or even weather conditions.

4.3 Running AI-Based ADAS

Various driver assistance systems exist, with different features and use cases. Some of them make driving comfortable (e.g., adaptive cruise control), and some of them protect the user and other traffic participants (e.g., precrash systems). There are ADAS with or without actuators: emergency brakes can physically intervene when danger is detected, but alert systems only help with the driver's decision-making, but generally, they run a sense-decide-act loop:

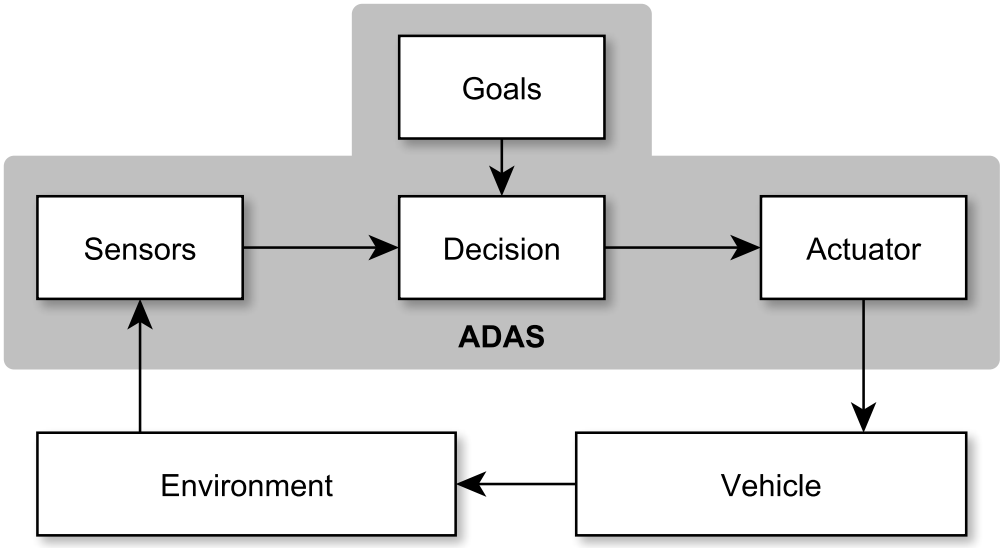


Figure 4.3: Functional overview of a generic ADAS

Advanced driver assistance systems today are up to Level 2 on the scale of driving automation levels[1], meaning that they partially automate the driving. The Level 1 assistance systems can control the vehicle either laterally or longitudinally, Level 2 assistance systems can do both: accelerate, brake, and steer at specific scenarios, but the driver must supervise the system at all times.

In our work, we primarily focused on the lane detection capabilities of decision-making modules, hence, we ran the decision-making module in a standalone way, without the rest of the feedback loop. To do this, we provided the sensory inputs and goals, like images and other model-specific required inputs. After the execution, we capture the signals of the decision-making module, and then we evaluate them, without actually executing the signals given by the ADAS.

4.3.1 Openpilot

Openpilot[1] is an advanced open-source driver assistance system developed by comma.ai. The system is designed to perform various driver assistance functions such as adaptive cruise control (ACC), automated lane centering (ALC), forward collision warning (FCW), and lane departure warning (LDW). To run openpilot, the developer company offers a dedicated hardware, which can be connected to most modern cars. This device has a touch screen, where the driver can see the detected lanes and lead car overlayed on the real-time camera footage. Besides this, they can also interact with openpilot: engage or disengage functions, see maps, and more. The hardware is equipped with two front cameras, and a camera for driver monitoring.

As openpilot is public, everyone can access its inner end-to-end pre-trained model, which provides ADAS functionalities. We tested the lane-detection capabilities of this model.

4.4 Evaluation of lane detection results

As written in Section 2.1, multiple lane detection methods exist, with different performance and accuracy. To evaluate such models, we need the ground truth and the predicted values. We also have to define an error metric or loss function that can evaluate how well the model's prediction aligns with the actual data.

The most straightforward error metric for each lane detection method is what they were trained with, although during the training, the objective function usually includes regularization and other components besides the cumulative loss function:

- For the semantic segmentation approach it can be cumulative pixel-wise losses [12].
- For row-wise classification, any loss function for classification can be used, e.g. cross-entropy or KL-divergence [33].
- Error metric for curve-fitting and anchor-based methods can be similar, for instance, geometric loss functions as mean squared error [31] for discrete (sum) or continuous (integral calculation) cases. Figure 4.4 makes it easier to understand the notion of geometric loss.

It is worth mentioning that all methods describe lane- and roadlines, hence they can be transformed into each other's representation, making it possible to evaluate them using other methods. Also, the lane detector models are usually not standalone, but integrated into an ADAS and are used in a dynamic environment. In this case, the nearer parts of the lanes are more important, this should be weighted in some way. With methods like mean squared error, a distance-based decay is easy and convenient to implement, we chose a similar approach.

In general, any ADAS (just like Openpilot) operates by continuously running a sense-decide-act loop (Figure 4.3), which iterates multiple times per second to adapt to the changing environment and new goals. For the dynamic use case, measuring the component's overall performance during each iteration and observing contingent inaccuracies and their effect on the ADAS's behavior is also important [25], but measuring, and evaluating dynamic behavior goes beyond the scope of our work.

[1] `https://github.com/commaai/openpilot`

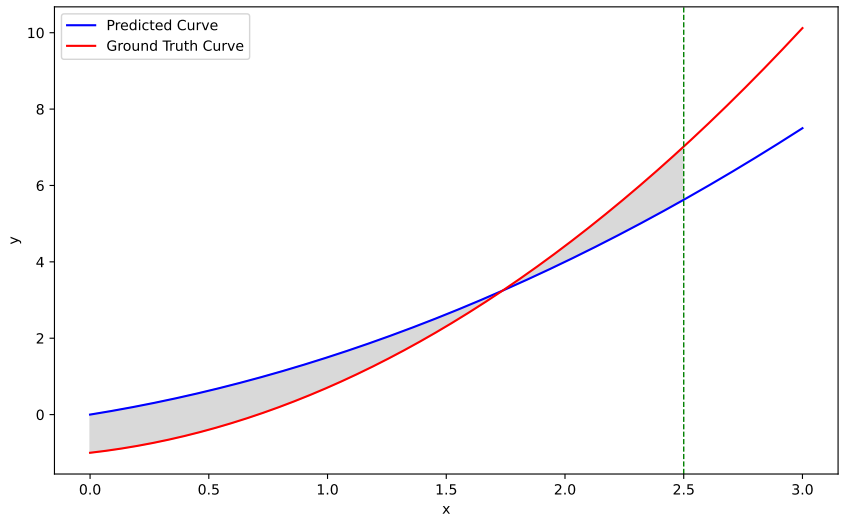


Figure 4.4: The geometric loss: the (squared) area between the predicted curve and ground truth curve up to a certain distance

4.5 Statistical analysis

The purpose of statistical analysis is to find out important details about the system's performance and the testing process during and after running the genetic algorithm. By comparing what the system predicts, what the actual correct answers are (ground truth), and what are the errors for the corresponding test cases, we can systematically identify the lane detection model's weaknesses, and we can try to find correlations between a more abstract representation of parameters (e.g., the curvature of the road) and error.

Moreover, we can also explore how the genetic algorithm works for test case optimization so that we can fine-tune the iteration or population count and other parameters. Observing the GA's performance could also reveal that the problem formulation was not optimal. Many factors can influence the results, but the representation of the problem (in our case, the road parametrization) and the error function are the most important. Choosing bad geometry of the wrong range for parameters could result in invalid roads, for instance, with too much curvature. The wrong error function could make the algorithm optimize for irrelevant aspects, for example, a lane-detection component of a lane-keeping system should be more accurate for closer points, however, semantic segmentation's error metric does not necessarily encompass this property by default.

Chapter 5

Case study

In this chapter, we will detail our work and results, with the technologies, challenges, and solutions for them. The brief overview of the genetic algorithm (GA) based workflow is visible in Figure 5.1. The workflow is modular, and we used widely used standard data formats (e.g., OpenDrive for maps, FBX for 3D objects) and sometimes custom ones, like the results of Supercombo (Section 5.3). Using loosely coupled modules and standard data formats gave us great flexibility. In the future, we can easily extend it with an extra simulation execution step, or we could similarly evaluate other lane-detection models.

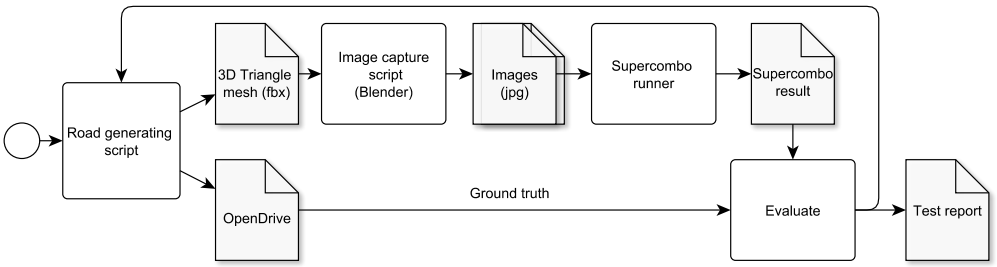


Figure 5.1: The workflow in detail

5.1 Parametrized road generation

5.1.1 Input parameters

We wanted to start the research with roads that are made of a single geometry. With this, we can achieve simple tests, and if we find critical road geometries, this way would mean that the system under test has fundamental weaknesses. To do this, the most parameterizable path geometry provided by the OpenDrive format should be used, which is the parametric cubic curve.

Using Equation 2.2 and Equation 2.3 meant that such a path could be described by 8 parameters. However, constraints could be introduced for some of the parameters at the beginning. The parameters aU and aV are equal to 0, as these parameters only affect the shift of the curve, which will not change the shape of the curve itself. In addition, it was also necessary that the curve and the y-axis were parallel at the origin so that the camera representing the car would be facing the correct direction on the road. Of course, the camera could be rotated according to the parameters, but this would lead to extra

calculations that would slow down the test generation, and more parameters would have to be taken into account, of which it was advisable to introduce as few as possible.

Using the Figure 2.2 in our case to determine the correct direction meant the following:

$$u'(p) = bU + cU * p + dU * p^2 \tag{5.1}$$

$$v'(p) = bV + cV * p + dV * p^2 \tag{5.2}$$

$$\frac{(u'(p))}{(v'(p))} = \frac{(bU + cU * p + dU * p^2)}{(bV + cV * p + dV * p^2} \tag{5.3}$$

$$\frac{(u'(0))}{(v'(0))} = \frac{bU}{bV} := inf \tag{5.4}$$

This is possible if $bV = 0$ and $bU \neq 0$, because if both parameters were 0, there would be a vertex of the curve at that point.

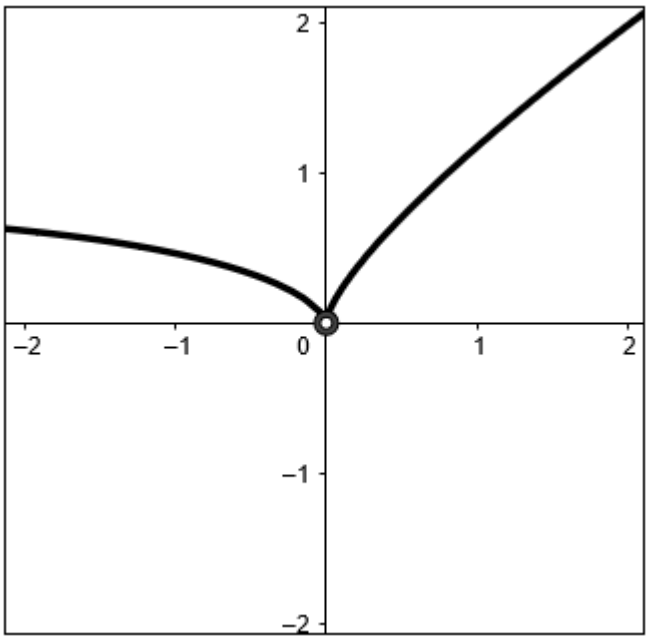


Figure 5.2: Derivative of the cubic curve

Ultimately, only five parameters of the cubic parametric curve are the ones that actually had to be taken into account.

5.1.2 Bézier curve as input

The problem with the previously described solution was that a small change in parameters could result in a large change in the appearance of the curve, so the parameters of the parametric curve were not optimal for use in the genetic algorithm. To overcome this problem, we decided to replace the parameterization of the parametric curve with the parameters of a third-degree Bézier curve.

The equation of the third-degree Bézier curve:

$$\mathbf{B}(p) = (1-p)^3 * \mathbf{P0} + 3 * (1-p)^2 p * \mathbf{P1} + 3 * (1-p) p^2 * \mathbf{P2} + p^3 * \mathbf{P3} \tag{5.5}$$

Here, the points of the Bézier curve were constrained in order for the curve to be defined in the correct heading. The first point must be at the origin, and the second point must be on the y-axis. These constraints were given for the same reasons that we were able to fix the parameters of the parametric curve. The Bézier curve thus has four points:

- **P0**=(0,0)
- **P1**=(0,y1)
- **P2**=(x2,y2)
- **P3**=(x3,y3)

Since the OpenDrive format does not support path definition with Bézier curves, the Bézier curve parameters must be converted to cubic parametric polynomial parameters before creating the test files. This can be done after the equation of the curve has been decomposed and sorted. After these are done, the parametric curve is described:

$$u(p) = y0 + 3(y1 - y0)p + 3(y0 - 2 * y1 + y2)p^2 + (y3 - y0 + 3(y1 - y2))p^3 \tag{5.6}$$

$$v(p) = x0 + 3(x1 - x0)p + 3(x0 - 2 * x1 + x2)p^2 + (x3 - x0 + 3(x1 - x2))p^3 \tag{5.7}$$

Using the parameter constraints:

$$u(p) = 3(y1)p + 3(-2 * y1 + y2)p^2 + (y3 + 3(y1 - y2))p^3 \tag{5.8}$$

$$v(p) = 3(x2)p^2 + (x3 + 3(-x2))p^3 \tag{5.9}$$

As an example, consider the Bézier curve with the following points:

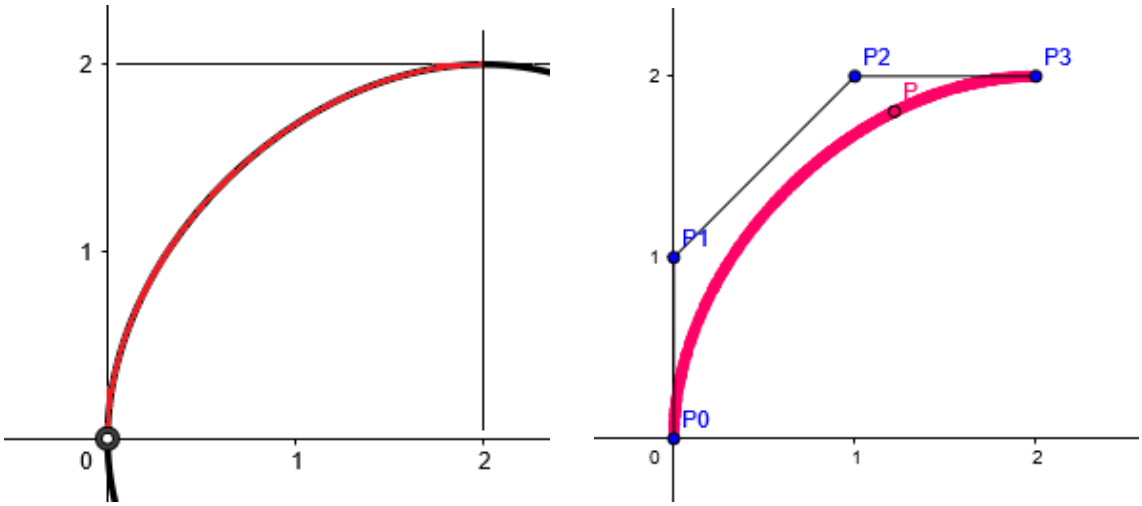
- **P0**=(0,0)
- **P1**=(0,1)
- **P2**=(1,2)
- **P3**=(2,2)

Calculating the parametric curve from this:

$$u(p) = 3 * p + (-1) * p^3 \tag{5.10}$$

$$v(p) = 3 * p^2 + (-1) * p^3 \tag{5.11}$$

By plotting the two curves, it can be seen on Figure 5.3 that they correspond to each other, but it is important to note that in general, the Bézier curve is only valid for $0 \leq p \leq 1$, so for the parametric curve, it must be compared to this subcurve.



(a) Parametric curve **(b)** Bézier curve

Figure 5.3: The connection between the two curves

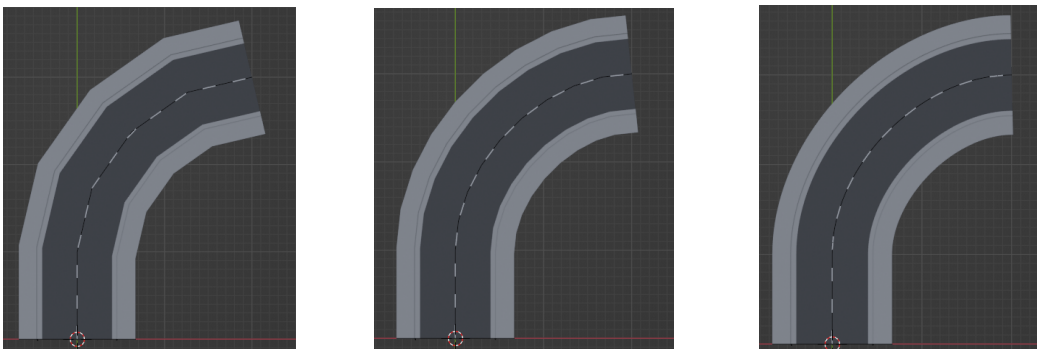
In addition to what has been described so far, the function that generates the curve must have one more parameter, which determines the range that p runs on. In addition, the functions for generating the straight line and the arc have been implemented, but since all the necessary information can be determined by simple calculations, this will not be discussed in this paper. Initially, only paths consisting of a parametric curve were tested, but the generator is also capable of generating complex curves since any number of the three different curve types can be successively connected in any order.

5.2 Visualization

5.2.1 Processing the testfiles

After generating the OpenDrive format tests, the next step is to process them to generate the three-dimensional models. For this and the generation of the models themselves, we used Blender, as it is an open-source modeling software with Python scripting support. Before processing, it was necessary to define the structure of the input files that the script could process. Here, to match the generation, we specified that the OpenDrive file should be a single two-lane road consisting of straight lines, arcs, and/or parametric cubic curves. In addition, the file may also contain a slope profile to simulate the topography. The Blender script runs through the geometry nodes in the OpenDrive XML file after the file has been read. In each case, it checks to see what kind of geometry it describes and then extracts the appropriate parameters from the node based on the geometry. After each geometry, a plotting function is called, which plots a curve on the x-y plane using the geometry parameters. Since the format also has a heading parameter, which gives the starting direction of the curve segment, this can be read out, and successive curve segments can be rotated accordingly.

The input file is now processed, and Blender no longer needs it to create the three-dimensional models. In addition to the road curve, in the future, the number of lanes that the road consists of or whether there is a pavement on the side of the road could be used from the input files. These will help to generate even more varied cases. Currently,



(a) Quality = 5 **(b)** Quality = 10 **(c)** Quality = 50

Figure 5.4: Different quality setting

a two-lane road with a pavement is the only one that can be generated. For this, the Blender script creates a road profile consisting of the following:

- Road
- Gutter
- Pavement
- Sidewalk

This is necessary because in Blender, models are created using a ”sweep” operation, and for this, you need the profile in addition to the path curve. In the script, it is also possible to set a ”quality” parameter, which specifies how many points each geometry should consist of when generating the curve. The quality of the models will depend on this parameter as well. As this parameter affects the computation cycles required for generation, it can be used to speed up/slow down the generation somewhat, allowing the user to tailor it to their own system resources.

It can be seen on Figure 5.4b that if the resolution is set to 10, a very realistic path can be achieved. In addition, it is useful to add a small straight section to the path at both the beginning and the end of the path to facilitate proper orientation at the endpoints of the path.

5.2.2 3D model generation

In Blender, the ”sweep” operation used to create path models can only be defined for a specific curve and a specific profile. In this case, these two curves will be mapped to each other, which means that the elements that build up the road have to be swept separately. Also, it is important to note that this curve will determine the height at which the sweep operation will create the model. Therefore, for example, the curve used to generate the sidewalk should be shifted by a small value in the z-direction. To make this easy to implement, the functions that generate the geometry can be given a z offset.
The profile that will be swept by the program consists of the elements mentioned above and looks like on Figure 5.5.

The different colors in the picture represent the different building blocks:

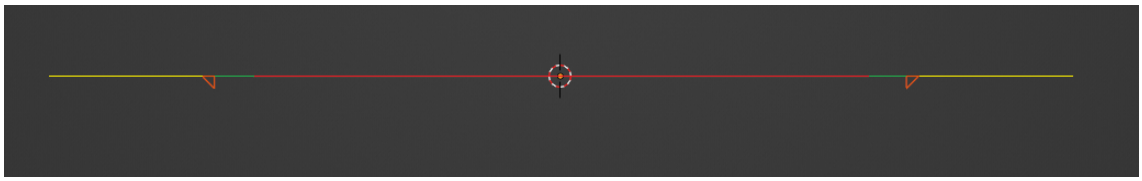
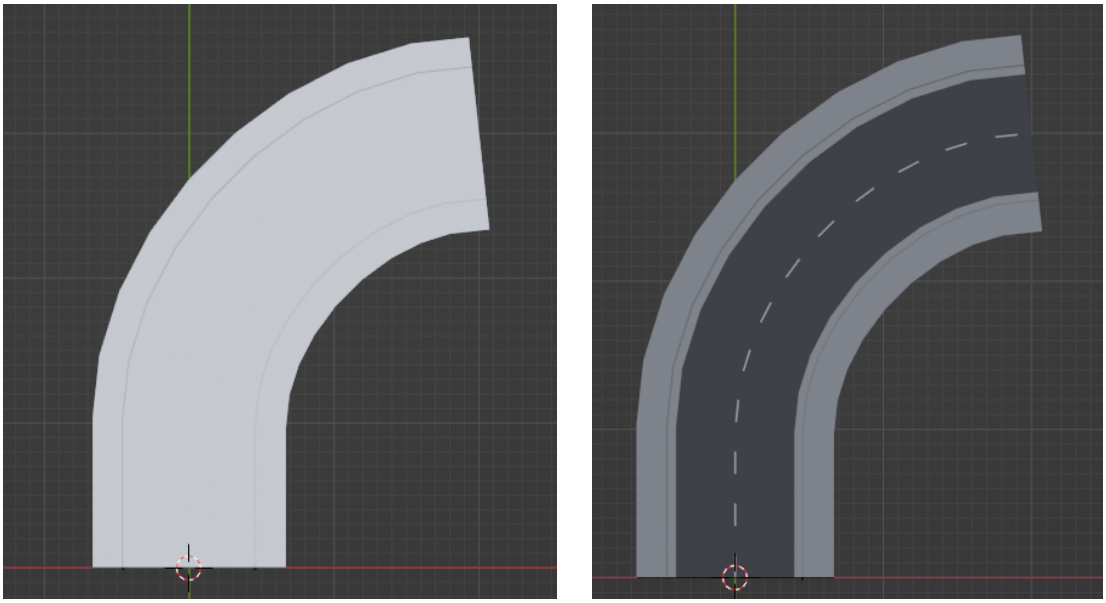


Figure 5.5: Profile of the road



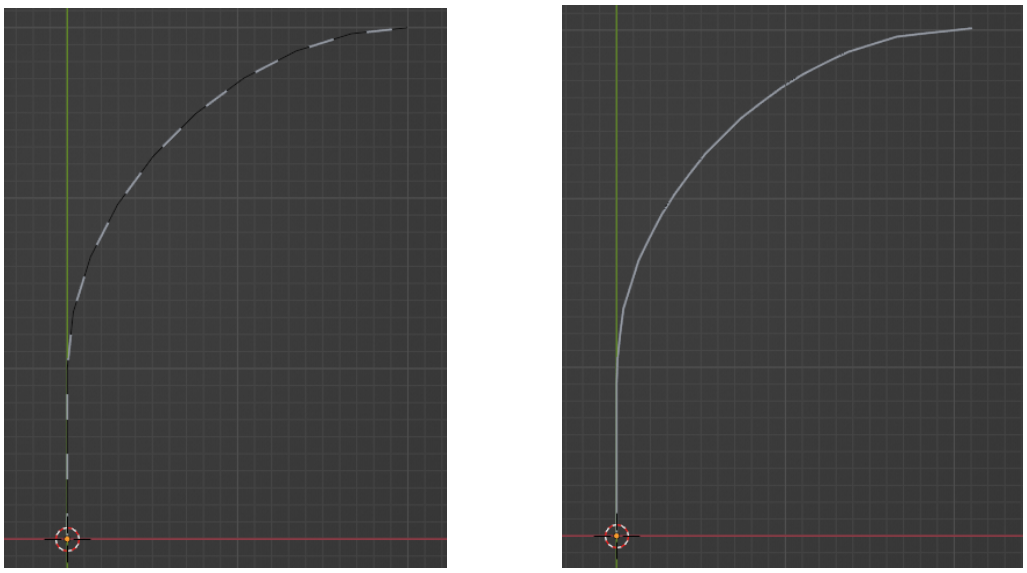
(a) The generated road without textures **(b)** The road with textures applied

Figure 5.6: The generated road

- Road - Red
- Gutter - Green
- Curb - Orange
- Sidewalk - Yellow

Once both the profile and the curves have been created, the sweep can be made. The model created by this operation does not have any textures yet, and the lane marking has not yet been created. The generated model can be seen on Figure 5.6a.

We can create the lane marking using the modifiers available in Blender, which are automatic operations that affect an object's geometry in a non-destructive way. To do this, we first need to create a cuboid that will represent a single strip of the painting. Then, two modifiers must be assigned to the cuboid. The Array modifier creates an array of copies of the base object, with each copy being offset from the previous one in any of a number of possible ways. This will implement the cloning of a model according to some pattern. The other modifier is the curve, which will define the curve along which the cloning will take place. In addition, the array modifier needs to be configured to specify how far apart the copies should be and how many copies should be made in order to display them correctly. By varying the distance between them, you can even achieve a continuous line.



(a) Lanemarking **(b)** Continuous line

Figure 5.7: Different lanemarkings

The number of strips can be set by assigning a curve so that it can change dynamically depending on the length of the path. Figure 5.7

After the model itself has been created, the textures can be assigned to the different parts of the model. A total of three simple monochrome textures were used for the road:

- Dark gray - Road
- Light gray - Gutter, Pavement, Sidewalk
- White - Lanemarking

Of course, these textures can be easily changed within the script, so you can easily customize the look of your roads. For now, the textures can be set with RGB values inside the script, but in the future, it is planned to extend the script so that more complex textures can be used in the generation. The road with textures applied can be seen on Figure 5.6b.

5.2.3 Creation of environment

So far, we have a three-dimensional road model, but it does not yet include any environment that could affect the output of the tracking algorithm. First, we created a customizable skybox and set a sun direction, after which the model and its environment can be seen on Figure 5.8a.

Then, all that remains is to create the ground. Here, we have proposed three different land generation scenarios, of which two have been implemented.

1. Lane/Curve following (implemented, see on Figure 5.8b): Here, the ground is prepared by sweeping, just like the road itself. This achieves the most realistic ground, but here, the sweeping causes the model's point mesh to overlap, which can cause visual distortions.

2. Vertical (implemented, see on Figure 5.8c): In this case, the earth is defined as a distant wall. This is not a problem when using a simple texture, but with more complex textures, this 90-degree rotation may cause problems.

3. Horizontal (not implemented): The counterpart of the vertical solution. Here, a simple plane will be the ground. The problem with this solution is that the height of the road would have to be taken into account during generation, and the size of the plane would have to be increased/decreased based on this. This would mean extra computations, which we wanted to avoid in order to speed up the genetic algorithm.

After this, a green texture was added to the ground in order to achieve a grass-like look. Figure 5.8 shows the generated road with the environment.

(a) A road with a skybox **(b)** The lane following method **(c)** The vertical method

Figure 5.8: Different environment additions

With these, the road and its surroundings are created. In addition, to take the pictures, a camera had to be added in the Blender script in the right position and with the right settings.

5.3 Running the lane detection component: Supercombo

Supercombo serves as the core of the openpilot (see Section 4.3.1) ADAS. It is a pre-trained end-to-end neural network responsible for making the decisions for the driver assistance tasks. While the model is open-source, the training data and the training process are not available to the public. The study by [5] explores the possibilities of using, retraining, and evaluating the model on public benchmarks, providing valuable insights and extensions to the limited official documentation.

5.3.1 Inputs

Openpilot preprocesses the input data from the recorded image stream and user commands, known as desires. This data is then used to run the Supercombo model in a loop.

- **Input images**: Two consecutive images (256 * 512 * 3 in RGB) recorded at 20 Hz. Field of view (fov): 40°
- **Wide input images**: Two consecutive images with the same resolution, but they have a wider field of view (120°).
- **Desire**: A one-hot encoded vector to command the model to execute certain actions.

- **Traffic convention**: A single bit indicates left- or right-handed traffic.
- **Initial state**: The recurrent state vector that is fed back into the Supercombo model for temporal context.

5.3.2 Outputs

Openpilot processes the Supercombo model's outputs to perform the desired tasks. Additionally, Supercombo generates outputs for visualization purposes, which can be considered as the "knowledge" upon which the model makes decisions. This visualization capability is particularly important for making an end-to-end black box model slightly more interpretable.

- **Lane Lines**: The model generates information about the detected lane lines, allowing it to maintain the vehicle's position within the lane. It identifies 32 points for both the left and right lanes at fixed distances.
- **Road Edge**: Supercombo identifies the edges of the road to provide a better understanding of the driving environment. Similarly to lane lines, it also contains 2x32 points at the same distances.
- **Lead Car**: The model detects the presence and location of a lead car in front of the host vehicle, enabling the system to maintain a safe following distance when using adaptive cruise control (ACC) or lane keeping.
- **Meta data**: A recurrent state vector is output and fed back into the GRU for maintaining temporal context.
- **Pose**: The model outputs the vehicle's pose, which includes information about its position, orientation, and velocity.
- **Plan**: Based on the input data and its internal knowledge, the model generates a plan for executing the desired actions, such as adjusting speed, steering, or warning the driver of potential hazards.

5.3.3 Running supercombo

The model is available in the official repository[1] of openpilot, in Open Neural Network Exchange (ONNX) format. The Open Neural Network Exchange is an open-source standard for machine learning models, it enables interoperability between different AI tools by defining a common set of operators and a common file format. ONNX models can be created and utilized across various supported frameworks, like PyTorch, TensorFlow, or Keras. This makes the model portable and easy to run in different environments. We used `onnxruntime` python package to run the model.

To run Supercombo in a standalone way, we developed a wrapper for inputs and outputs, providing a user-friendly interface for interaction. The input wrapper receives two consecutive images for both fields of view (narrow: 40°, wide: 120°). The resolution is not important in this step, as the wrapper handles the proper scaling. Optionally, you can provide the remaining input parameters: the traffic convention (right-handed by default),

[1] https://github.com/commaai/openpilot/blob/master/selfdrive/modeld/models/supercombo.onnx

initial (recurrent) state (array of zeros by default), and desire (also array of zeros by default). After observing the Supercombo's architecture[2], it is visible that the latter three inputs will not affect the outcome of the lane or road detection.

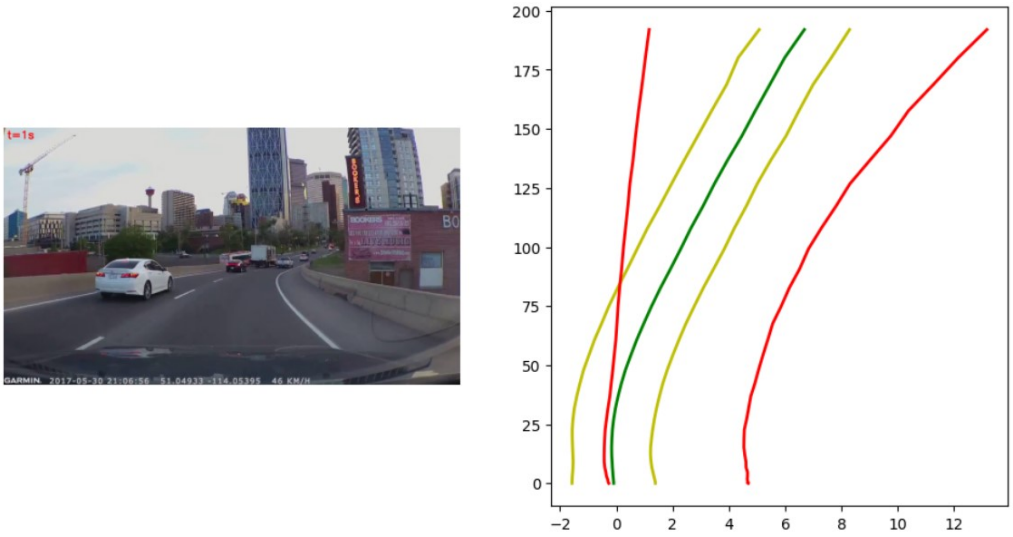


Figure 5.9: Input image and Supercombo's prediction: Yellow: lane-lines, Red: road-lines, Green: average of the yellow lines

5.4 Evaluating supercombo result

Evaluating the result of the Supercombo is not straightforward, as it has diverse output: lane and road lines, lead car, actuator signal, and also a recurrent state vector, but fortunately, the different types can be separated easily from each other. We focused only on the lane detection part of the mode, which used the curve-fitting approach (Section 2.1.3). Also, we can extract the ground truth with several different methods. The first idea was to use a semantic segmentation camera and see whether the identified lane points were on the correct color. Achieving the comparison required an overlay method, which can accurately map the Supercombo's output on the semantic segmentation image while taking into account the camera parameters, like position, field-of-view, and distortion. In some cases, testers may only have semantic segmentation images as ground truth. However, this solution is not adequate in all cases because lane markings can be not only continuous but also dashed lines. A detected lane point could fall between the two dashes, but this approach would naively consider it a misclassified lane line point. Fortunately, we have the road geometry as ground truth in a standardized format so that we can apply more adequate approaches.

The evaluation of the model's output in our work is based on the OpenDrive map descriptor file. To compare the actual map with the result, we can accurately extract the geometries and calculate the distances between the output lane line points and the exact locations.

Error metric In our approach, we evaluate the estimation accuracy of OpenPilot's SuperCombo model by comparing the individual lanes with the ground truth from the OpenDrive map. We define a geometric error metric, the normalized difference between

[2]Even the simplified version of the model would be too complex to include here, but onnx models can be visualized online, for example with netron app.

the estimated lane positions ($\bar{x}$) and the true lane positions (x). Specifically, the error (err) is calculated as follows:

$$err = \sum_{n=1}^{32} \frac{|x_n - \bar{x}_n|}{192(n/32)^2} \tag{5.12}$$

Supercombo applies a curve-fitting lane detection approach(Section 2.1.3), it is trained to predict the lateral distance from the center for certain distances. The concrete distances are defined in their repository by this $index\_function$, for given index idx. Supercombo uses the default values for maximum value max_val and maximum index max_idx, too. The model predicts 32 lane line points, in closer areas, the sampling is more dense. The detection distances are present in Figure 5.10 as horizontal lines. The function is defined as:

```
def index_function(idx, max_val=192, max_idx=32):
  return (max_val) * ((idx/max_idx)**2)
```

This function provides the normalization factor in the error metric, offering the distance-based importance weights.

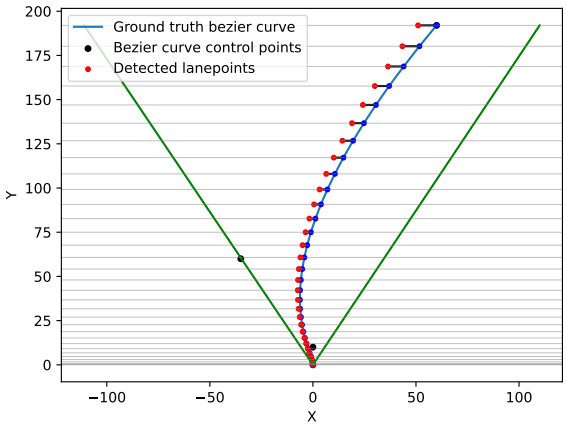


Figure 5.10: Ground truth and detected lane points

To apply the error metric to the minimizing genetic algorithm, we reformulated the error:

$$errForNSGA = MaxErr - err \tag{5.13}$$

where, MaxErr has the value of $32 \cdot \tan(60°) \approx 55,42$. With this error, anything above 35 is fine for the ADAS functionality, and anything below 30 is critical. The MaxErr value came from the field of view of openpilot's wide camera. We considered the maximal error when it identifies a straight lane as if it were on the edge of the visible area. Technically, this could be even higher when the detected (or the actual) road is outside the visible area. This means that $errForNSGA$ could theoretically have a negative value, but our measurements showed that this does not happen when the road is in the visible area. Also, NSGA-II can handle negative values, however, this is not true for all GAs.

5.5 Statistical analysis of the results

When analysing the output of the system, it is clearly visible that the error decreased over time to around 20 generations. After this, the error value stabilized, which meant that the algorithm found a minimal value where the lane detection works the least effectively.

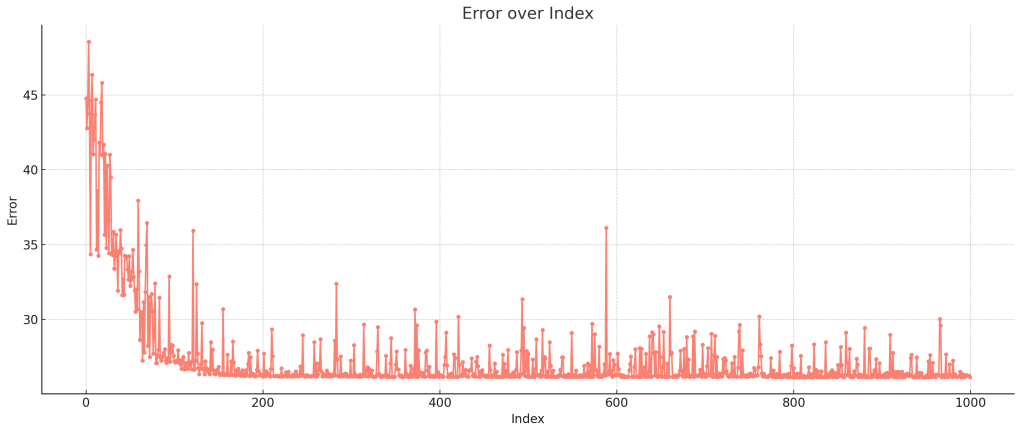


Figure 5.11: Error value from the genetic algorithm over time

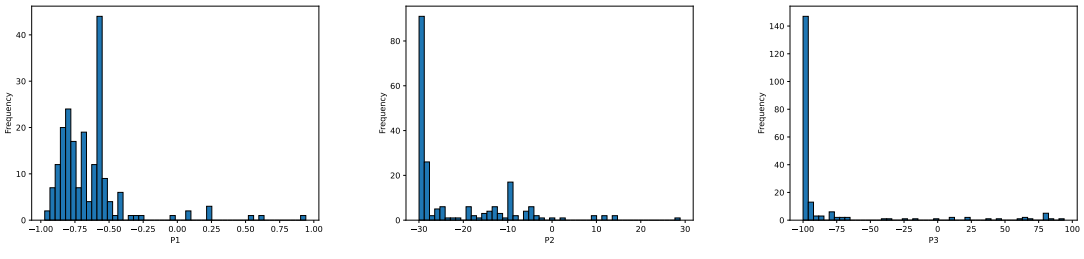


Figure 5.12: Occurance of parameter values for the first 20 generations

As we can see on Figure 5.12, there is an early convergence. The results are interesting but slightly expected. The algorithm finds the boundary values early.

We decided that to represent a certain Bézier curve, we would use the maximum absolute curvature of that curve. Compared to the multiple parameter representation, this is easier to represent and can give more adequate results. To find the curvature of a curve at a certain point, we used the following equation, where u(p) and v(p) are the curve's x and y coordinates over p parameter:

$$kappa = (u'(p)v''(p) - v'(p)u''(p))/(v'(p)^2 + u'(p)^2)^{3/2} \tag{5.14}$$

TESTRUN 1-3 are three examples of the evaluation of the results. On all of the TESTRUNs the first coordinate of the Bézier curve was set at (0,0) in order to make the curve start from the origin. For TESTRUN 1 the remaining points and parameters were set the following way:

- P1 - (p1,10)
- P2 - (p2,50)

- P3 - (p3,200)

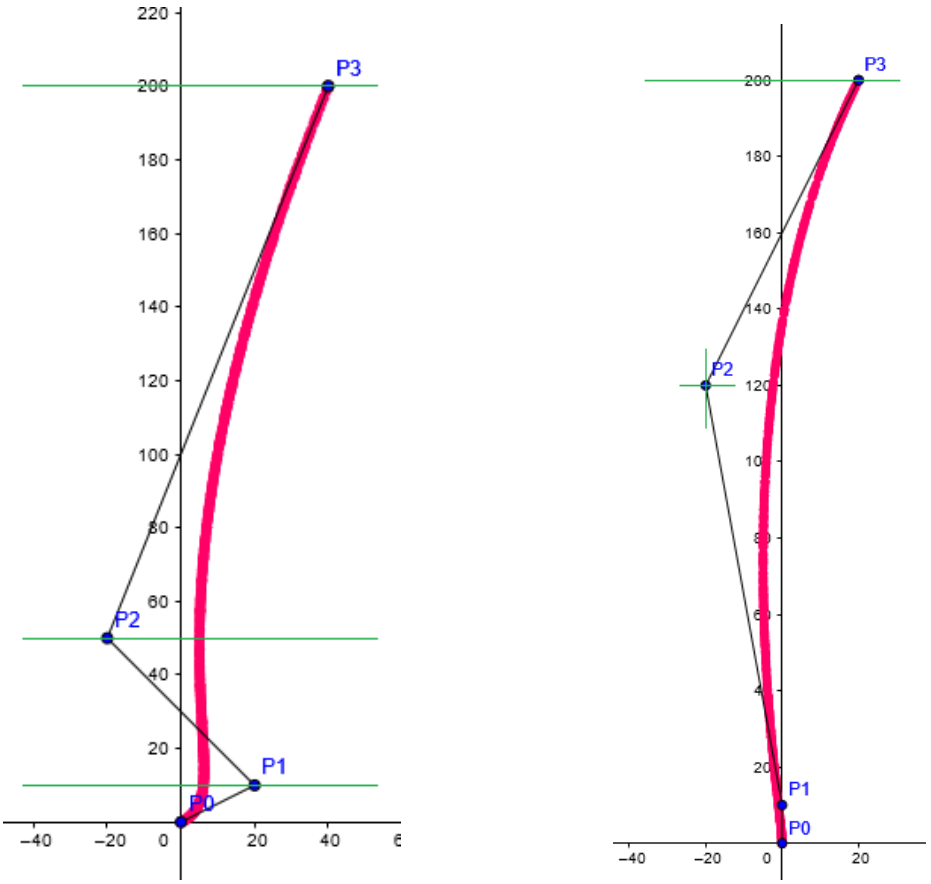
It meant that we have set y coordinates, and only the x coordinates can change with a certain limit on the range of values, so the road stays on the screen.

For TESTRUN 2 and 3, the points and parameters:

- P1 - (0,10)
- P2 - (p2,p1)
- P3 - (p3,200)

Here, the points make the starting direction of the road to be parallel to the y-axis. But P2 can be moved on the y-axis as well as on the x-axis. The differences between the two runs are the starting parameters and the number of generations.

The way the points can be set can be seen on Figure 5.13, the points can be shifted in the direction of the green lines.



(a) Points of TESTRUN 1 **(b)** Points of TESTRUN 2-3

Figure 5.13: Points of the TESTRUNs

The derivatives for the Bézier curve with our fixed parameters for TESTRUN 1:

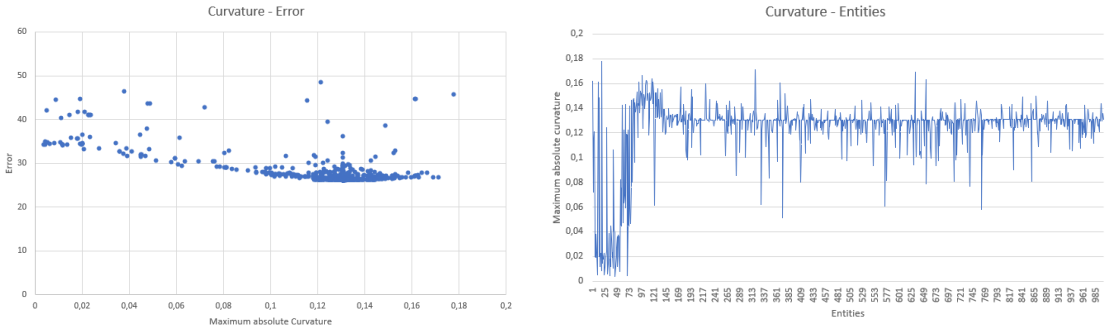
$$u'(p) = 30 + 600p + 240p^2 \tag{5.15}$$

$$u''(p) = 600 + 480p \tag{5.16}$$

$$v'(p) = 3(x1) + 6(-2 * x1 + x2)p + 3(x3 + 3(x1 - x2))p^2 \tag{5.17}$$

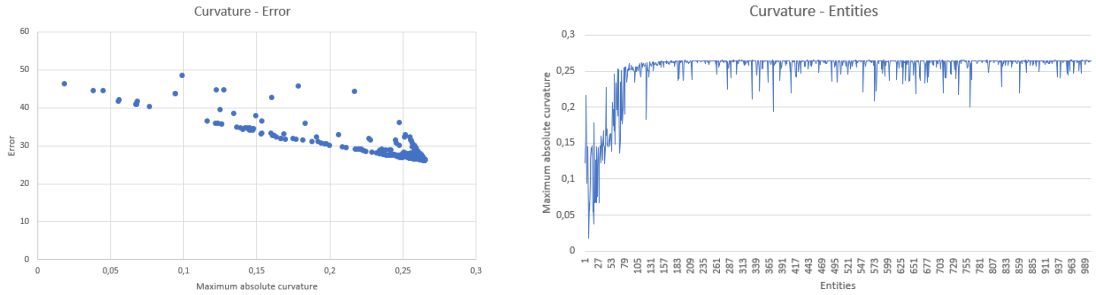
$$v''(p) = 6(-2 * x1 + x2) + 6(x3 + 3(x1 - x2))p \tag{5.18}$$

From these, we were able to calculate the curvatures for each parameter set. For the other TESTRUNs, the calculations were similar to TESTRUN 1, just using different parameters.



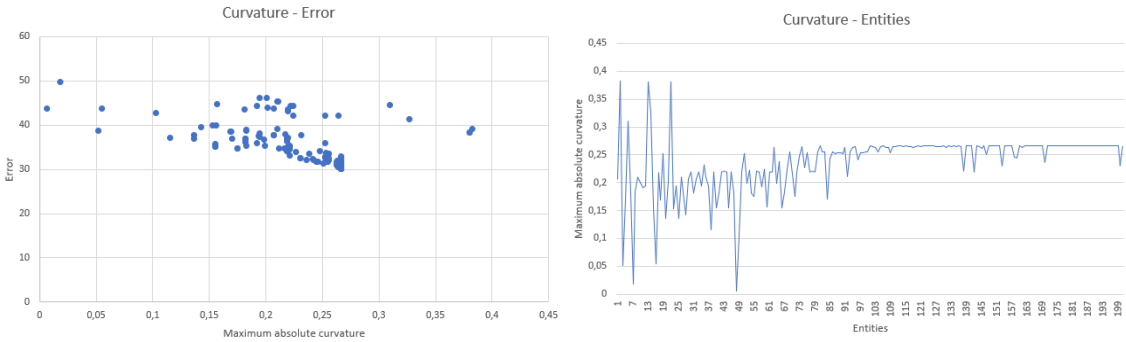
(a) Curvature compared to error **(b)** Curvature over time

Figure 5.14: TESTRUN 1



(a) Curvature compared to error **(b)** Curvature over time

Figure 5.15: TESTRUN 2



(a) Curvature compared to error **(b)** Curvature over time

Figure 5.16: TESTRUN 3

Figure 5.14a, Figure 5.15a, and Figure 5.16a show the maximum absolute curvature of a parameter set compared to the error value of the genetic algorithm. As is visible on the graphs, higher maximum absolute curvature values of the curve meant lower error

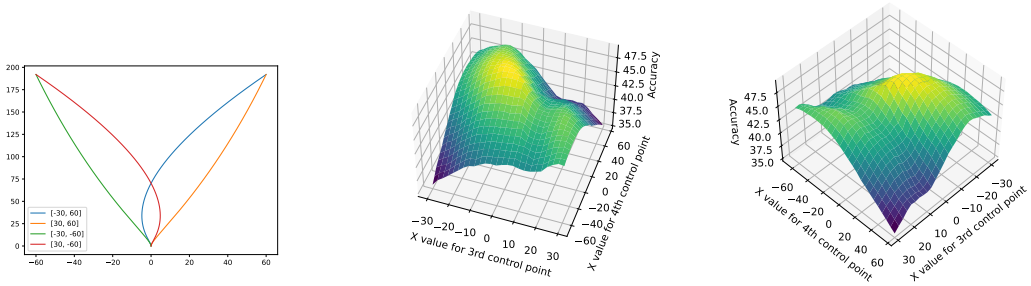
values. This might be the case because, so far, our test roads are quite simple, and they only consist of a simple geometry. From the curvature-error comparison, we can deduce that the error value of the algorithm and the curvature of the road have a connection. It can also be said that the system under test performed weaker on roads that have higher maximum curvature.

Figure 5.14b, Figure 5.15b and Figure 5.16b show the maximum absolute curvature of a parameter set over time. Each generation of the genetic algorithm consists of 10 entities, so for 100 generations, we had 1000, for 20 generations, we had 200 entities. It is visible that over time, the curvature stabilized, which means that the algorithm was successful in finding less recognizable roads. Examining the signed curvature values, we noticed that they are negative values. In terms of road geometry, it meant that left turns were harder to be recognized by the system.

For TESTRUN 1, the correlation between the error and the signed curvature is around 0,8. For TESTRUN 2, around 0,9, and for TESTRUN 3, around 0,62. From these, we can deduce that by allowing the y coordinates of the points to be changed, we can achieve a higher correlation. In addition, if we run the algorithm for a longer time (more generation), the correlation increases as well.

The most challenging road shape for the Supercombo can be found after 10-20 generations with a population size of 10 in each generation. Running the algorithm is time-consuming, the aforementioned 10 generation takes 7-8 minutes[3], but running 100 generations took around 1 hour and 15 minutes.

After the conclusion that Supercombo's lane detection is the weakest, when it tries to evaluate the most left-leaning lanes, we tried to evaluate this asymmetry by visualizing lanes using a reduced parametrization. The newly generated road geometries had the Bézier-type parametrization, with the control points (0,0), (0,10), (x3, 60), (x4, 192), where x3 ranges from -30 to 30, and x4 ranges from -60 to 60. The most extreme lane-line geometries are presented in Figure 5.17a. This measurement was just an explanatory analysis of the asymmetry, but as visible in Figure 5.17b and Figure 5.17c, from different sides, the error values based on these parameters are continuous (with this rate of sampling). The origin of the x and y pane should mark the central point of symmetry, but as visible from the plot, the accuracy is far from symmetric. Interestingly, around x3 = 10, the model had decreased accuracy, but the corner cases finally overtook. For this measurement, we only used two parameters for visualization purposes, but the genetic algorithm runs explored a broader input space, with denser sampling around the weaker spots.



(a) Cornercase road geometries **(b)** Accuracy 2 variables **(c)** Accuracy 2 variables, other side

Figure 5.17: Asymmetry evaluation

[3]On a regular PC, specs: GPU: Nvidia 1660 super, CPU: AMD Ryzen 5 3600x, RAM: 32 GB

Chapter 6

Discussion

After running the genetic algorithm to explore roads with challenging curves for the Supercombo (Section 5.3), we gathered interesting information, about the system under test, and about our framework, too.

First, the early version of our testing workflow relied on raw parametric cubic curves. We limited the possible values of the parameters, but the algorithm quickly found the invalid parameters for the road-line geometry, which were self-intersecting or not visible from the camera's perspective with openpilot's field of view. These resulted in huge errors, and showed, that the algorithm can clearly optimize towards huge errors. To generate better test cases, we switched to the Bézier curve-based road geometry description. With this approach, we were able to provide proper upper and lower bounds to each parameter. With every combination of the potential parameter values, the synthesized road is visible and does not have too sharp curves or self-intersections.

6.1 Conclusion

Running Supercombo on the proper, visible roads had pretty good accuracy, the accuracy did not go below the critical level (Equation 5.13), and we could not identify any adversarial examples within the pre-specified input space.

Based on the statistical analysis, it can be said, that there is a connection between the error value of the algorithm and the curvature of the road, more curvature resulted in weaker (but still not critical) accuracy. The direction of the road curve also affects the results of the algorithm. We also discovered that on roads with a leftward skew, lane detection consistently shows reduced accuracy compared to the mirrored lanes.

20 generations were usually enough, for the genetic algorithm, to find the corner cases, the roads with the most curvature. After this was found, the diversity of the inputs significantly fell, because the crowding distance mechanism only works, when the inputs' are performance-wise similar, but here, the corner cases have notably worse performance, than the others, so the algorithm will not choose the very different, but easier road as test input. This means, that there are more samples taken from the critical areas.

- The genetic algorithm found challenging but not critical inputs for the system within the range of the input space.
- Interesting finding: Supercombo's lane detection is asymmetric; it performs worse on left-leaning lanes.

- After the genetic algorithm explores the worst-performing input boundary, the diversity of the newly generated inputs significantly falls, in return the model's accuracy is much weaker on these inputs, and the critical input boundaries will be tested more thoroughly.

6.2 Future work

The synthetic environment is simple, there are no roadside objects or other vehicles, also the weather conditions and the visibility are all the same during each generated road. This ensures that the search-based approach only takes the road curvature into consideration, but openpilot has to deal with more complex environments in the real world.

Our current work only focused on testing supercombo's lane detection capabilities, however, we could use our testing workflow, to evaluate dynamic behavior, by running openpilot or other ADAS in a simulator. In order to achieve this, we need to adapt our current execution and evaluation steps to deal with the simulation and handle dynamic, driving-oriented error metrics, such as time-to-collision, end-to-end lateral deviation metric[25] or others.

We only tried to use a genetic algorithm in order to achieve a challenging and diverse test set. The literature showed that search-based approaches performed better than random sampling, but we have not compared our approach with other methods yet. In the future, we want to compare the test set generated by our GA-based approach with others.

In terms of future work, the following areas can be further investigated:

- Adding diverse environmental elements, such as vehicles, pedestrians, trees, traffic signs, and other objects could enhance realism.
- Dynamic behavior evaluation of lane-keeping systems in a simulator, with search-based approach.
- Comparing the test cases generated with our method against the result of other approaches.

6.3 Threats to validity

Our experiments showed valuable insights into the openpilot's lane-detection component. However, our experiments were executed on a synthetic map, with only a very basic surrounding, without traffic signs, trees, other roadside objects, or dynamic actors such as vehicles or pedestrians. The visibility, like weather conditions or sun lighting, was also unmodified during the test-case generation. For strictly seeing the effect of the road geometry, the lack of variation is fine, but this is not what the system sees in the real world, and maybe in another environmental condition, the system behaves differently.

Bibliography

[1] Taxonomy and definitions for terms related to driving automation systems for on-road motor vehicles, 2018.

[2] Raja Ben Abdessalem, Shiva Nejati, Lionel C Briand, and Thomas Stifter. Testing vision-based control systems using learnable evolutionary algorithms. In *Proceedings of the 40th International Conference on Software Engineering*, pages 1016–1026, 2018.

[3] Zohreh Aghababaeyan, Manel Abdellatif, Lionel Briand, Ramesh S, and Mojtaba Bagherzadeh. Black-Box Testing of Deep Neural Networks through Test Case Diversity, December 2022. URL http://arxiv.org/abs/2112.12591. arXiv:2112.12591 [cs].

[4] J. Blank and K. Deb. pymoo: Multi-objective optimization in python. *IEEE Access*, 8:89497–89509, 2020.

[5] Li Chen, Tutian Tang, Zhitian Cai, Yang Li, Penghao Wu, Hongyang Li, Jianping Shi, Junchi Yan, and Yu Qiao. Level 2 autonomous driving on a single device: Diving into the devils of openpilot, 2022. URL https://arxiv.org/abs/2206.08176.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2): 182–197, 2002. DOI: 10.1109/4235.996017.

[7] Alexey Dosovitskiy, German Ros, Felipe Codevilla, Antonio Lopez, and Vladlen Koltun. CARLA: An Open Urban Driving Simulator, November 2017. URL http://arxiv.org/abs/1711.03938. arXiv:1711.03938 [cs].

[8] Attila Ficsor and Balázs Pintér. Simulation-based robustness testing of adas systems. In *Student Research Societies Report*, 2021.

[9] Daniel J. Fremont, Tommaso Dreossi, Shromona Ghosh, Xiangyu Yue, Alberto L. Sangiovanni-Vincentelli, and Sanjit A. Seshia. Scenic: A Language for Scenario Specification and Scene Generation. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 63–78, June 2019. DOI: 10.1145/3314221.3314633. URL http://arxiv.org/abs/1809.09310. arXiv:1809.09310 [cs].

[10] Alessio Gambi, Marc Mueller, and Gordon Fraser. Automatically testing self-driving cars with search-based procedural content generation. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 318–328, 2019.

[11] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is neuron coverage a meaningful measure for testing deep neural

networks? In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 851–862, 2020.

[12] Shruti Jadon. A survey of loss functions for semantic segmentation. In *2020 IEEE conference on computational intelligence in bioinformatics and computational biology (CIBCB)*, pages 1–7. IEEE, 2020.

[13] Cezary Z Janikow, Zbigniew Michalewicz, et al. An experimental comparison of binary and floating point representations in genetic algorithms. In *ICGA*, volume 1991, pages 31–36, 1991.

[14] Pengfei Jing, Qiyi Tang, Yuefeng Du, Lei Xue, Xiapu Luo, Ting Wang, Sen Nie, and Shi Wu. Too good to be safe: Tricking lane detection in autonomous driving with crafted perturbations. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 3237–3254, 2021.

[15] Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

[16] Lorenz Klampfl, Florian Klück, and Franz Wotawa. Using genetic algorithms for automating automated lane-keeping system testing. *Journal of Software: Evolution and Process*, page e2520, 2022.

[17] Florian Klück, Martin Zimmermann, Franz Wotawa, and Mihai Nica. Performance comparison of two search-based testing strategies for adas system validation. In *Testing Software and Systems: 31st IFIP WG 6.1 International Conference, ICTSS 2019, Paris, France, October 15–17, 2019, Proceedings 31*, pages 140–156. Springer, 2019.

[18] István Majzik, Oszkár Semeráth, Csaba Hajdu, Kristóf Marussy, Zoltán Szatmári, Zoltán Micskei, András Vörös, Aren A Babikian, and Dániel Varró. Towards system-level testing with coverage guarantees for autonomous vehicles. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 89–94. IEEE, 2019.

[19] Till Menzel, Gerrit Bagschik, Leon Isensee, Andre Schomburg, and Markus Maurer. From functional to logical scenarios: Detailing a keyword-based scenario description for execution in a simulation environment. In *2019 IEEE Intelligent Vehicles Symposium (IV)*, pages 2383–2390. IEEE, 2019.

[20] University of Toronto. Parametric curves. URL `https://www.cs.helsinki.fi/group/goa/mallinnus/curves/curves.html`.

[21] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, page 1–18, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350853. DOI: `10.1145/3132747.3132785`. URL `https://doi.org/10.1145/3132747.3132785`.

[22] Zequn Qin, Huanyu Wang, and Xi Li. Ultra fast structure-aware deep lane detection. In *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIV 16*, pages 276–291. Springer, 2020.

[23] Vincenzo Riccio and Paolo Tonella. Model-based exploration of the frontier of behaviours for deep learning system testing. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 876–888, 2020.

[24] Abbas Sadat, Sean Segal, Sergio Casas, James Tu, Bin Yang, Raquel Urtasun, and Ersin Yumer. Diverse complexity measures for dataset curation in self-driving, 2021.

[25] Takami Sato and Qi Alfred Chen. Towards Driving-Oriented Metric for Lane Detection Models. In *Conference on Computer Vision and Pattern Recognition (CVPR)*, 2022.

[26] Sepehr Sharifi, Donghwan Shin, Lionel C. Briand, and Nathan Aschbacher. Identifying the Hazard Boundary of ML-enabled Autonomous Systems Using Cooperative Co-Evolutionary Search, January 2023. URL `http://arxiv.org/abs/2301.13807`. arXiv:2301.13807 [cs].

[27] Jiawei Su, Danilo Vasconcellos Vargas, and Kouichi Sakurai. One pixel attack for fooling deep neural networks. *IEEE Transactions on Evolutionary Computation*, 23 (5):828–841, 2019.

[28] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[29] Lucas Tabelini, Rodrigo Berriel, Thiago M Paixao, Claudine Badue, Alberto F De Souza, and Thiago Oliveira-Santos. Polylanenet: Lane estimation via deep polynomial regression. In *2020 25th International Conference on Pattern Recognition (ICPR)*, pages 6150–6156. IEEE, 2021.

[30] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314, 2018.

[31] Wouter Van Gansbeke, Bert De Brabandere, Davy Neven, Marc Proesmans, and Luc Van Gool. End-to-end lane detection through differentiable least-squares fitting. In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshops*, pages 0–0, 2019.

[32] Shuai Wang and Zhendong Su. Metamorphic object insertion for testing object detection systems. In *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1053–1065. IEEE, 2020.

[33] Seungwoo Yoo, Hee Seok Lee, Heesoo Myeong, Sungrack Yun, Hyoungwoo Park, Janghoon Cho, and Duck Hoon Kim. End-to-end lane marker detection via row-wise classification. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition workshops*, pages 1006–1007, 2020.