Università
della
Svizzera
italiana

**Software
Institute**

DETECTION OF SYNTHETIC FAULTS IN SELF-DRIVING CARS

Master of Science in Software and Data Engineering

Manjali Nimisha

June 2021

Supervised by
Prof. Paolo Tonella

Co-Supervised by
Dr. Gunel Jahangirova, Nargiz Humbatova

SOFTWARE & DATA ENGINEERING MASTER THESIS

Abstract

Mutation testing is a technique that is widely adopted to test the traditional software and is based upon seeding artificial faults into the program's source code. The general assumption behind this approach is that artificial faults used by mutation testing should represent real mistakes made by developers.

Currently, there are two major classes of mutation operators applied to Deep Learning (DL) systems: pre-training (source-level) and post-training (model-level). The latter operators are usually computationally cheaper as they mutate the weights or structure of an already trained network. The former are more expensive as they mutate the sources of the DL systems (original network structure and training data) prior to the training and therefore require to retrain the DL model.

Self-driving vehicles is a rapidly evolving field with a high variety of adopted DL solutions. There exists a number of quality metrics to evaluate the performance of an autonomous vehicle, for example, the deviation of a car's position from the center of a lane or the standard deviation of the speed of a vehicle. When applying mutation testing to such systems, the expectation is that the injected faults will lead to the violation of one or more quality metrics. Otherwise, if the effect of mutation is not reflected in any of the metrics, such operators are not introducing any erroneous behaviour into the DL system under test.

In this thesis we aim to apply mutation operators to the autonomous vehicles model, identify the list of mutation operators that are useful for testing of autonomous vehicles, i.e. affect the values of driving quality metrics, compare existing mutation killing definitions (not specific to autonomous vehicles) to the killing criterion based on the violation of quality metrics, extract dominant, useful and unique quality metrics in the field of autonomous driving.

Our results show that killing criterion based on the violation of quality metrics discriminate mutated DL models from non-faulty DL models better than existing mutation killing techniques in the field of DL systems. The outcome of the thesis provides researchers and practitioners with a set of highly-effective quality metrics and a new approach for testing of autonomous vehicles.

Dedicated to whoever is reading this…

Acknowledgements

I would like to express my deep and sincere gratitude to my thesis supervisor Prof. Paolo Tonella, and co-advisors Dr. Gunel Jahangirova and Nargiz Humbatova for giving me the opportunity to do this thesis and providing invaluable guidance throughout this work. It was a great privilege and honor to work and study under their guidance. I thank them for the keen interest shown to complete this thesis successfully.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Deep Learning (DL) is a new data-driven programming paradigm where the internal system logic is largely shaped by the training data. It has gained a great popularity in the various fields of science and industry. The increasing dependence of safety-critical systems, such as autonomous vehicles, on DL networks makes the testing of such systems a crucial topic.

Mutation testing is a popular method to measure test adequacy in traditional software testing. The fundamental idea behind mutation testing is to insert artificial defects called mutants into the source code of a software under test. After injecting the faults, the ratio of number of killed mutants with respect to the total number of created mutants is called the mutation score. Mutation testing relies on the assumption that test suites achieving a high mutation score are also very likely to be able to expose the real faults that affect the original program, while test suites achieving a low mutation score need to be improved, i.e. high mutation score indicates a strong test set.

An example case of mutation testing is shown in Figure 1.1. In the example, the original program compute the difference between two numbers. By introducing mutation into it, i.e, by changing the subtraction to multiplication and addition, we get mutant versions of the original program. Once these mutant versions of the program is obtained, a test program is executed over both original and mutated programs. If the result of test on mutants are different from that of the original program, the mutant is said to be killed. In the provided example, the test passes for the original and Mutant2. In case of Mutant1, the test fails as it detects the outcome is wrong, thus the mutation introduced is exposed, and therefore it is killed. While for Mutant1, even though the mutation is injected in it, the test couldn't reveal it. Hence, Mutant2 is survived.

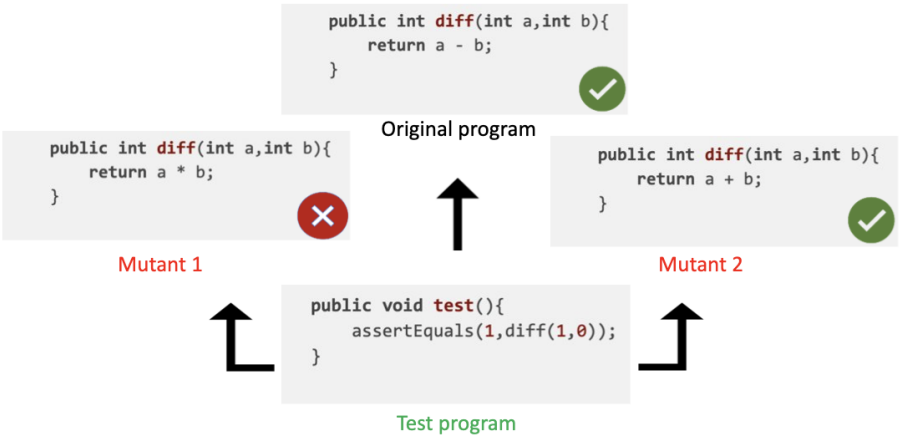

FIGURE 1.1: An example of mutation testing in traditional software

The notion of a fault in DL systems is more complex than in traditional software. Unlike traditional software systems, the decision logic of which is often implemented by software developers in the form of source code, the behavior of a DL system is determined by various components. Figures 1.2 and 1.3 demonstrate the workflow for the development process of traditional software systems and DL systems correspondingly. Specifically, the connection weights of deep neural networks (DNNs) are obtained through the execution of the training program on the training data set, whereas the structure of the model is often defined by code fragments of training program in high-level languages (e.g., Python and Java). Therefore, two major sources of faults in DL systems are the training data set and the training program. In consequence, for mutation testing of DL systems, we need mutation operators to inject potential faults into the training data or the DNN training program other than standard, syntactic operators. In fact, existing standard operators can only be applied to training programs, and they mostly lead to the injection of simple faults that either cause a crash or create syntactic changes that are unrelated to the model's behaviour being trained [9]. Numerous proposals [12], [17], [7] exist in the literature to address this issue.

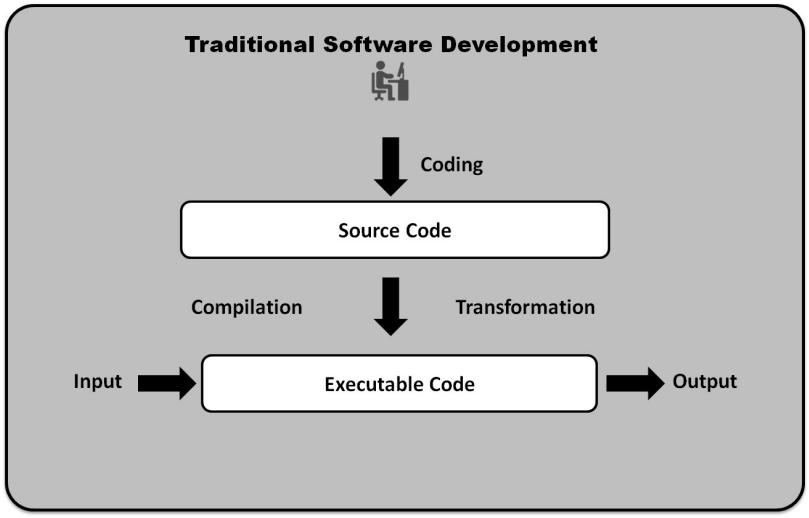

FIGURE 1.2: Traditional software development process

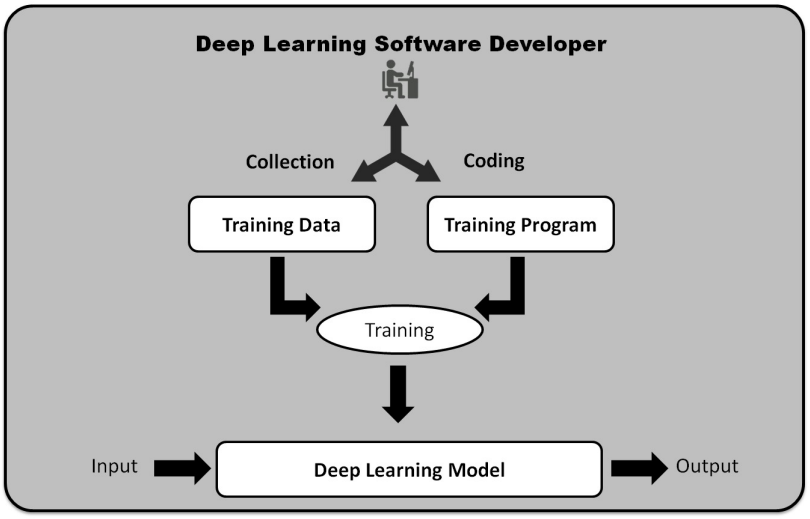

FIGURE 1.3: Deep learning software development process

To analyze whether a given test suite is able to expose faults, i.e. to kill mutants, there exist killing criteria based on the accuracy drop, where the impact of the mutation operators is measured as the difference in the accuracy of the original and mutated model. Besides the need of specific mutation operators, there is also a need to move forward from the existing mutation killing definitions when it comes to DL systems. This is because of the stochastic nature of the training process. Hence, we need to assess the likelihood that the effect observed that may introduce behavioural differences on the mutated models does not occur by chance. This problem is identified by researchers and a suitable attempt [9] is taken. The above-mentioned attempt focus on providing a statistical definition of mutation killing. Briefly, the authors perform n re-trainings of original and mutated models and check if there is a statistically significant difference between values produced by them, followed by calculating the effect size to quantify the size of the difference. We call this method of mutation killing as model-level mutation killing because it is about killing the mutants based on values retrieved from the DNN model. Comparing this to the mutation killing based on accuracy drop, the latter may result in inconsistent results across different runs, as said because of the random nature of training phase.

The focus of the thesis is to carry out mutation testing on autonomous vehicles. Mainly, we intend to propose a new killing criterion for mutation testing of DL systems, demonstrating its practicality in the field of autonomous vehicles. We introduce our new approach, called system-level killing, which assesses the behavior of a DL system based on a set of quality metrics of the domain of interest. During the operation of an autonomous car, deep neural networks are often used to identify boundaries of lanes or to predict steering angle values. We aim to analyze whether the set of existing mutation operators are able to create a diverse set of faulty driving behaviors. This analysis can be performed by observing whether the mutants cause the violation of driving quality metrics [3]. Our proposal is also combined with statistical terms of definition of mutation killing. Also, we point out how effective are quality metrics in assessing the behavior of the DL model of interest i.e. how far the quality metrics can detect misbehaviours introduced by mutation.

1.1 Definitions

1.1.1 Deep Learning Systems

Deep learning is a part of machine learning which falls under the larger umbrella of artificial intelligence. In deep learning, the features are not provided explicitly but are picked out by the neural network without human intervention. This kind of independence comes at the cost of having a much higher volume of data to train the neural network. The information processing takes place in neurons, the core entity of the neural network. An example design of a simple neural network given in Figure 1.4. The information is transferred from one layer to the next one starting from the first (input) layer over connecting channels between different layers. Weights and bias are the two main concepts of neural networks which are calculated and updated during the training process. Each channel has a value (weight) attached to it and is called a weighted channel. All neurons have an associated number attached to it called bias. The weighted sum of inputs reaching the neuron is then multiplied by this bias, which is then applied to a function known as the activation function. The result of the activation function decides whether the neuron is activated. Every activated neuron feeds information to the following layers. This is continued until the second last layer is reached. The one neuron activated in the output layer corresponds to the desired output. More specifically, for classification systems the maximum activation determines the predicted class of category. While for regression systems, which are DL systems which produce continuous variable as output, the maximum activation will be already the predicted numeric value, for example, steering angle prediction. The weights and bias are continuously adjusted to produce a well-trained network.

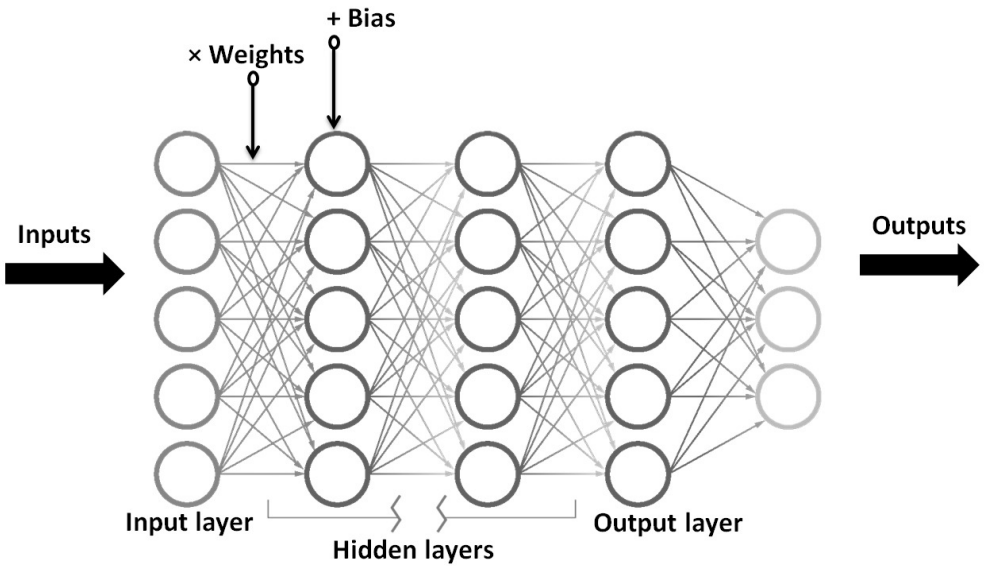

FIGURE 1.4: Deep learning neural network.

Going more in detail into the computation of a single neuron, the neuron (the circle in the Figure 1.5) have input and output edges going in and out that connects the neurons between them. The edges have weights attached to them (W_1,W_2,W_3). There are inputs (Input 1, 2, 3) into the neurons. These inputs can be from previous neurons in the network or from raw data depending on where the neuron is placed. As the first step, a weighted sum X, is calculated taking the sum of those inputs along with some bias. Thereafter, the weighted sum X is fed into an activation function for example, Sigmoid, Relu etc. which will decide the actual output of a single neuron. This is how a neuron gets activated and pass output to the next neuron. As can be noted, the weight, bias and activation function influence largely the behaviour of DL network. In our study we introduce some mutations into these constituting elements of DL networks in order for the misbehaviour detection.

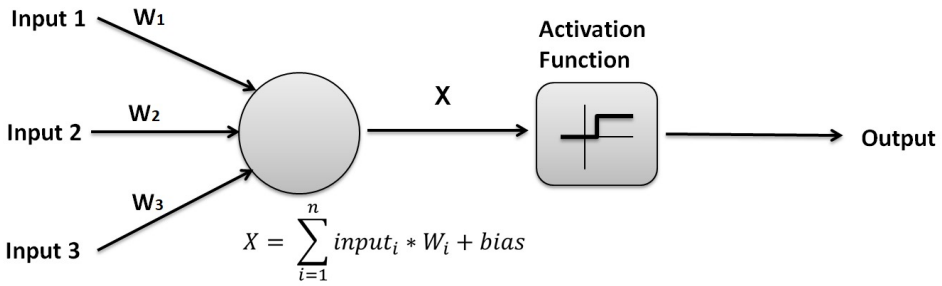

FIGURE 1.5: Functioning of a single neuron of the neural network.

As already mentioned, training data plays a major role to make possible this kind of automation in prediction of desired output. It is the initial dataset used to train or teach a neural network to have accurate predictions. The data should be complete, clean and properly annotated thus the training data preparation is an important step in building a good DL network. The amount and characteristics of data needed, depends on domain or use cases and how confident we want our predictions to be. For instance, if the domain is autonomous vehicle, the data required will include images or videos labelled to identify street signs, roads, other cars or pedestrians. Once the training phase is complete, the performance of the trained

DL network is evaluated by running it against a test dataset. If the quality of prediction is low, usage of more training data is a recommended solution to ensure more accuracy. At the same time, more training data can also lead to the problem of overfitting.

Overall, deep learning has a vast scope in various fields including image recognition, diagnosis and treatment of diseases in medical field, autonomous driving, speech recognition etc. This is due to the fact that neural networks have the capability to operate in a constantly changing and unpredictable environments. As a result, it is desirable to certify them for high-reliability tasks in safety-critical systems. However, as the solutions to the problems are learned implicitly from training data in deep learning related approach, direct evaluation of their correctness is not possible. This raise concerns when the resulting systems are applied to support safety-critical operations, such as autonomous driving of automobiles. The faults in such systems can lead to serious consequences such as property damage, loss of life etc. Therefore, it is necessary to test the performance of DL systems and ensure that they are of high quality.

1.1.2 Testing Levels of DL Systems

Traditional software testing can be performed at different levels of software development process. Testing at multiple levels helps in early identification of faults and improves the quality of software applications. Unit, integration, system, and acceptance testing constitutes classic software testing levels. All modules or components are tested in isolation during the first level of *unit testing*. The developers create a test script that tests particular method of application. Unit testing has the advantage of detecting errors early on, saving time and money in the process of fixing them. However, the integration issues are not detected at this stage, which is a limitation. As a consequence, while a module may work perfectly in isolation, it can have problems when interacting with other modules. *Integration testing* aims at finding interfacing issues between the different modules of the application. *System testing* examines the entire integrated application as a whole. It is performed on the entire system in the context of either system or functional requirement specifications, or both. It is carried out in a setting that is quite similar to that of its production environment. The final step, acceptance testing, is usually performed by the client to ensure that the program meets a predetermined standard of quality, after which it can be delivered to production.

Despite advancements in software testing, DL-based systems pose particular issues because their behavior is determined by both the code that implements them and the data that is used to train them. All typical software testing levels can be utilized for DL, though they may require further specialisation. Different levels of testing in DL systems are described by Riccio et al. [16] in their study. Model level testing in DL systems is one of the testing methods that can be compared to unit testing in traditional software testing for units that rely on training process such as DL model, training data, hyper-parameters. Model-level testing involves training DL models with a training dataset and by specifying hyperparameters, and then evaluating them with a test dataset. In the case of classification DL systems, the accuracy is measured, whereas in the case of regression systems, the metrics such as mean squared error and etc. are measured using the ground truth dataset in order to perform the evaluation. System level testing in DL systems can be done similarly to traditional systems by testing the DL system as a whole by keeping it in the environment where it is supposed to operate. Integration testing in DL systems, on the other hand, focuses on problems when many models and components are combined, even though each of them functions correctly in isolation. According to findings by Riccio et al., models are typically tested "in isolation" i.e. by model-level testing. To date, the model-level technique is the most extensively used testing method for evaluating DL systems. It is also crucial to look at how component failures that cause model-level errors affect the overall behaviour of the system in its environment.

1.1.3 Mutation Testing of DL Systems

The main idea behind mutation testing is to evaluate the performance of test suites by injecting artificial faults into the original version of the program and identify to what extent test suites could reveal them. A test suite is said to have good quality if it can kill many mutants.

With regard to DL systems, there are numerous operators proposed over time [12], [17], [7]. The behavior of DL systems is determined by different constituents such as training data, training program, structure and connection weights of the DL model, rather than just the source code as in the case of standard mutation testing. Hence, DL mutation operators target all such constituents.

The work by Ma et al. [12] introduces the idea of mutation testing specialised for DL systems. They have contributed both source-level operators which act on data and code, and model-level mutation operators that can be applied directly to an already trained model. They also showed the usefulness of their implemented operators and propose two metrics to quantitatively evaluate the test quality. MuNN [17], Mutation Analysis of Neural Networks, also proposes some mutation operators which can be injected to an already trained model. DeepCrime [7] proposes 24 mutation operators extracted from the real faults. These operators need to be applied before the training of the model. Other than proposing operators, several attempts [9], [17] have been made in the literature on how to determine if an injected mutation is exposed. These mutations killing techniques include killing a mutant, (1) if there is a drop in value of performance indicator, for instance, calculating the accuracy of predictions made by mutant model and comparing it to accuracy of original model [17], (2) if there is a statistically significant difference between original and mutant model values of accuracy [9]. Sections 2.2 and 2.3 contain more detailed information on mutation testing of DL systems.

1.1.4 DL-based Autonomous Vehicles

Human error is a major contributor to automobile accidents. No matter how safe a vehicle is, one thing that manufacturers have no control over is a driver mistake, or do they? Advanced driver assist systems come in various levels of automation to assist the driver in identifying and responding to risks. These are six levels of driving automation (see Figure 1.6) described by the Society of Automotive Engineers (SAE) international standard J3016 ranging from level 0 *no automation* to level five *full automation*. Levels 0 to 2 are considered as driver support features, while levels 3 to 5 are classified as automated driving features. Level zero features are limited to providing alerts, such as lane departure warning. At level 1 *driver assistance*, the driver retains complete control of the vehicle but receives some assistance, for example vehicle intervening with a single function such as steering or speed control. Level two *partial automation* provides a higher level of vehicle control than previous layers. Again, constant attention is required by the driver, but the vehicle handles a combination of functions based on the environment. For example, both steering and speed control. The third layer is *conditional automation*, where the vehicle manages all aspects of driving in certain conditions and alerts the driver to resume driving when those conditions change. Thus, readiness to take control remains key of the third layer. Level 4 *high automation* is a fully autonomous experience, but with a driver attention. The vehicle can mostly drive itself and handle even the most difficult conditions on highways and in city traffic. During the drive, the driver can focus on other things and is not required to constantly monitor traffic. However, Level 4 is only works for trips that are constrained to a specific location, like driving from point A to point B and back. Finally, without the need for a driver, in level 5 *full automation*, vehicles can operate under any driving condition where vehicles are completely self-driving automobiles. As the automotive industry strives for greater levels of automation in driving, the challenges to be solved become more complicated, appealing for the usage of deep learning methods. Over the last decade, autonomous vehicle technology had a rapid progress largely due to advances in deep learning.

The first ever DNN-based autonomous vehicle was brought into the field at the end of the 1980s by Pomerleau [15]. Today both in hardware and software aspects, autonomous cars underwent several years

of progression in technology. Recent advancements in Deep Neural Networks (DNN) led to development of neural network driven autonomous vehicles that can analyze in real time data collected using sensors like camera, LIDAR, etc., without any human intervention. With Graphics Processing Units (GPUs), the training costs of those DNNs has been significantly diminished.

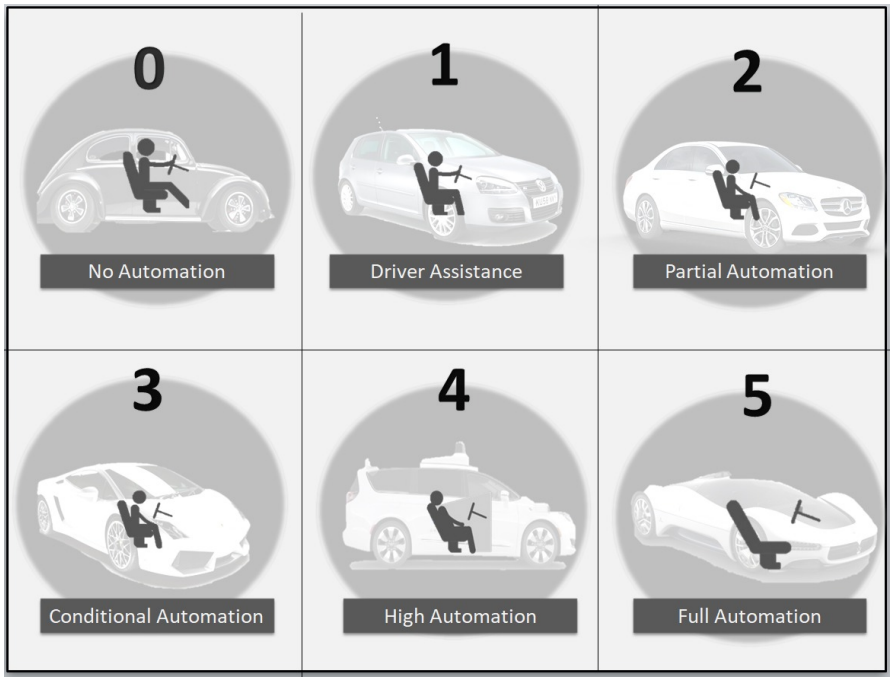

FIGURE 1.6: Automation levels in self-driving cars

In general, deep learning based autonomous driving system consist of three main phases [2] : perception, planning and action (see Figure 1.7). The first step is about discovering the environment and detecting the obstacles. The autonomous vehicles rely on sensors to capture and convey that information. The examples of sensors used are Radar, Camera, Lidar etc. The Lidar and Radar sensors provide information about localizing the obstacles in the environment, road segmentation, help in speed estimation, and many more involved tasks which are very mature today. Camera provides texture information, helps not only to figure out where the objects are, but also what they are, supporting, for example, lane detection, traffic signs/light detection etc. The data from these sensors are captured and streamed to the next phase which is the planning. This is the brain stage where deep learning plays a major role. The whole process including training, evaluation of DL model is done in this phase to make accurate predictions for example, prediction of steering angle. Once the predictions are made, the appropriate action is sent to the car, thus enabling automation. To date, autonomous driving has moved from the realm of science fiction to a very real possibility largely because of the rapid developments in areas constituting it such as deep learning approaches, sensors etc.

For the cases where DL based techniques are used in autonomous driving, unreliable sensors, problems with the data used for training, limitation in the algorithms and flaws in their design can all be sources of failures. The race to get DL autonomous vehicles onto public roads is driven by a promise that they would be as safe as or safer than human drivers. Therefore, proper testing of such systems to assure proper safety and reliability is crucial.

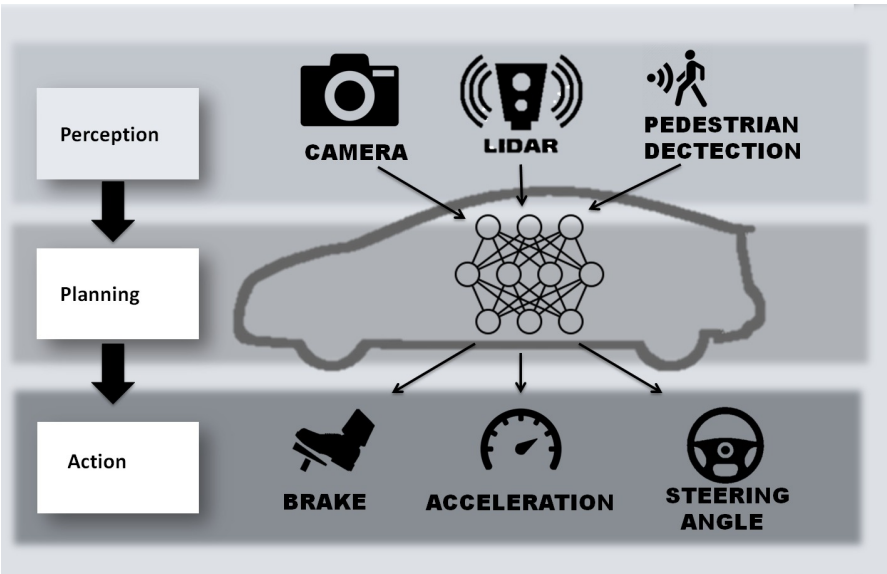

FIGURE 1.7: Deep learning based autonomous driving system overview

1.2 Research Questions

We address the following rersearch questions in this thesis:

- **RQ1**. *How does system-level misbehaviour compare to model-level mutation killing?* As we introduce a new approach for mutation testing in the context of DL systems, we are interested to understand how promising it is compared to the existing model-level approach.
  - **RQ 1.1**. *Do the mutants killed at the model-level always lead to a crash or an out-of-bounds scenario?* The crashes and out of lane bounds are the very first metrics used in our approach to expose the mutants. We would like to determine if the existing model-level approach is capable of exposing the mutants killed by these obvious driving quality metrics.
  - **RQ 1.2**. *Do the mutants killed at the model-level lead to a statistically significant change in the values of quality metrics?* This part of research concentrates on whether the metrics other than crashes and out of bounds kill the mutants using the definition of system-level mutation killing. We analyse how successful this is in comparison to model-level killing.
- **RQ2**. *Which driving quality metrics are the most effective in exposing system-level misbehaviour?* Since we use driving quality metrics in evaluation of the behaviour of autonomous vehicles, we want to find the metrics which are more effective to expose the injected faults.
- **RQ3**. *What is the minimal set of metrics that kills all mutants?* We are also interested to find out if there is a small-sized set of combined metrics that are able to reveal all injected faults.

1.3 Structure of the Thesis

This thesis is organized into the following chapters:

- **Chapter 2** provides a brief review of existing literature with the objective of presenting related works and how they paved the way for my thesis.

- **Chapter 3** describes the main idea of this thesis. Precisely, it explains our definition of system-level mutation killing, gives mathematical notation of the proposed approach and defines how the notion of system level killing is associated with autonomous vehicles.
- **Chapter 4** gives the description of the procedure followed to perform our experiments.
- **Chapter 5** presents the results obtained from experiments, and answers our research questions.
- **Chapter 6** wraps up the thesis, and provides an idea of the future work related to this thesis.

Chapter 2

Related Literature

2.1 Faults in DL Systems

One of the papers which investigate the faults that can emerge in DL systems is by Humbatova et al. [6]. The authors manually analyzed 1,059 artifacts collected from GitHub issues, commits of repositories and related Stack Overflow posts that use the most popular DNN frameworks such as TensorFlow, Keras and PyTorch. Moreover, they conducted interviews with 20 developers inquiring for the problems that they faced while developing such systems. This helped the authors to come up with a comprehensive taxonomy of faults in DL systems. Also, they point out that, they extended their research to interviews in addition to Stack Overflow and GitHub, as the latter sources missed issues that can be encountered during the training or model structure definition phases of deep learning systems. This highlights the fact that faults in deep learning systems are not only limited to code-level problems, as faults in traditional software systems. Their aim by these analyses was to understand the fundamental reasons behind the issues, so as to make a hierarchical classification of the faults or problems that can occur in DL systems. The taxonomy is structured in a way that groups of similar faults are differentiated first, and they are then categorized according to an "is a" relation.

The taxonomy is formed of 5 high-level categories which cover the faults related to the structure and properties of deep learning models, problems related to wrong shape, type or format of data, issues related to training process, problems concerning GPU usage, issues emerging from usage of framework's API. In addition, to validate the effectiveness of their taxonomy and to ensure that the discovered categories reflect the real faults in DL systems, the authors conducted a survey. This survey was performed among developers, different from those who were involved in the interviews carried out previously to extract problems. The survey asked the participants if they have ever experienced the issues from each category. The results show that there are no fault categories that the participants have not experienced while developing DL systems.

To demonstrate the usefulness of the taxonomy, the authors also provide a mapping between a number of mutation operators from the literature and a matching category from the taxonomy. At the same time, the authors underline that not all categories from the taxonomy have a corresponding mutation operator. For instance, there is not even a single matching category for post-training operators.

Another work which studies the faults related to deep learning systems is by Zhang et al. [22]. The authors manually examined issues from DL applications found in the TensorFlow library. They gathered 175 faults from Stack Overflow posts and GitHub projects. An analysis into the origins of faults and into the impacts of these faults on development process was performed by the authors. Furthermore, they identify challenges that can arise in detection and localization of faults and corresponding strategies applied by TensorFlow users to handle them.

The first challenge the authors note is that in traditional systems the correctness criteria of a program are clearly identified, i.e. a program is considered buggy if it produces wrong output given an input. But

in case of Tensorflow programs, the correctness criteria is probabilistic as it cannot be defined deterministically. For example, if the deep learning model predicts an input incorrectly, it doesn't necessarily mean that there is a bug in the program. To address this, the strategy followed by Tensorflow users is to rely on metric values like accuracy or loss. The second challenge is about coincidental correctness. This means that no failure is detected even though a bug is triggered. In deep learning systems there is a high risk of this taking place but remaining unnoticed due to the large complex computational model of a neural network. Third challenge is regarding the random nature of training process, which can produce different results in different executions thus making it hard to reproduce bugs. There are no successful ways taken by TensorFlow users to deal with this problem. The paper mentions that further studies are required to solve the issue of non-determinism in TensorFlow applications. Fourth and fifth challenges focus on fault localization. In traditional systems it's more straightforward to slice, extract and localize the fault. The fourth challenge brings out the need for specialized debugging techniques for deep learning systems, as code slicing doesn't help much due to the strong interconnected dependencies of neural networks. Therefore, the criterion of code slicing makes little sense when it comes to DL systems.

Another way of debugging in traditional programs is to check a program at a particular point by comparing expected values to with actual values of variables. This approach is not possible in deep learning systems because the behavior of the program is sensitive to hyperparameters set during training, and it is hard to predict a certain value at certain point of the neural network. This latter issue corresponds to the fifth challenge. The strategies taken to address this problem by TensorFlow users includes replacing hyper-parameters, switching training dataset or examining the distribution of variable values. Replacing hyper-parameters means changing the parameter values or functions used during training. For example, changing loss function, changing value of batch size, changing learning rates etc. Switching training set means changing between different datasets. For example, training data and validation data, to see if the problem is due to the training process. Another approach taken by a TensorFlow user to deal with this issue was to inspect the values of variables by a visualization tool of TensorFlow. By this way, even though the prediction of a precise value is not possible, one can examine some relations between different variables in different iterations or stages during the training process. These challenges reflect the shifts from traditional systems to deep learning systems, which emphasize that the faults in deep learning systems are different from those in traditional systems.

Islam et al. [8] address some research questions related to faults that can occur in popular DL systems. The authors analyzed Stack Overflow posts and GitHub bug fixing commits of five DL frameworks such as Caffe, Keras, Tensorflow, Theano, and Torch for their study. First they categorize faults into different groups. These groups of faults comprise *API Bugs*, *Coding Bugs*, *Data Bugs*, *Structural Bugs* and *Non-Model Structural Bugs*. *API Bugs* include faults caused by DL API, *Coding Bugs* include faults due to mistakes in coding, *Data Bugs* involve faults in data if it is not cleaned or well formatted. *Structural Bugs*, further divided to 4 more sub categories that define faults due to wrong design of structure of model, *Non-Model Structural Bugs* include all faults except the ones that fall in training or prediction phase. Categorizing the faults made it easier for the authors to understand the root causes of each bug, which is their second research question. Instances of root causes of faults identified by authors are absence of compatibility between different libraries, wrong documentation, inefficiency of model structure, wrong model parameters used in training. Subsequently, impact of these faults such as bad performance of DL model, crashes, incorrect functionality. are also studied. By categorizing all possible bugs and providing percentages of occurrence of each bug for each deep learning framework, they show that unlike traditional systems the notion of faults on deep learning systems mainly lies in data and structural logic parts. The authors also conclude that the Structural Inefficiency (SI) and Incorrect Model Parameter (IMP) are primary causes of the majority of faults and, crashes are the main effect of the faults. In addition, their study finds that a significant percentage of bugs happens in data preparation and training stage. Finally, they check if there is a common pattern followed by faults i.e., if there are any similarities between faults occurring in different DL libraries, as well as how these patterns change over time. This led to detection of some common

antipatterns such as Cut-and-Paste Programming, Dead Code, Spaghetti Code among different libraries.

These three above described studies [6], [22], [8] contribute to our thesis as many of these faults served as a basis for mutation operators that we applied during our experiments. The studies also underline to what extent DL based faults differ from the traditional ones.

2.2 Mutation Operators for Deep Learning Systems

As a response to the need of mutation operators specific to deep learning systems, there exist several contributions in the literature.

The first attempt to create mutation testing techniques specialized for deep neural networks is introduced by Ma et al. [12]. This tool consist of two categories of mutation operators: source level and model level operators. The authors initially design eight source-level mutation testing operators that directly manipulate the training data and the training program so as to introduce potential faults into them. The examples of training data mutations are data repetition, label errors, missing data, data shuffle, and noise perturbation. On the other hand, the program mutation operators change the network's structure by layer removal, layer addition or activation function removal. In the process of source-level mutation testing, original data and original program are mutated by injecting either data or program mutation. Then, the training process is re-executed by using mutated versions of model or data, or both. Once mutants are generated after retraining, they are evaluated using the test set and their performance is compared to the one of the original DL model. The more behavioral differences are detected, the higher is the quality of the test set. However, the generation of source-level mutations for DNN models can be computationally intensive, i.e. the training process can take minutes, hours, or even longer. Therefore, the authors further designed additional eight model-level mutation operators that mutate an already trained DL model by modifying its weights, biases or network structure. In contrast to source-level mutation ones, model-level operators do not require a retraining process since these mutation operators are applied directly to the trained model and therefore are more efficient. Once mutants are created, the remaining steps of evaluation are the same as defined for the source-level operators.

The paper shows the usefulness of their implemented mutation operators by applying and evaluating them on two popular datasets and three DL models. The authors show that the injected mutation operators helped them to capture the difference in test suites, which were constructed to be different in quality by design. For instance, in the evaluation of a mutated model against a test data made of uniformly sampled data i.e. which contains uniform amount of data for all classes, a high mutation score is achieved. The effect of mutation operators were studied also using test data of various sizes. They show that the size of test data might not be sufficient for uncovering more faults. Overall, the authors demonstrate that the framework can be used effectively to bring out the weak tests such that it helps developers to identify and improve them. In addition, the authors demonstrate that mutants generated by model-level and source-level are associated with different misbehaviours, which points out the need of both operators that act on data and DL models.

The work by Ma et al. was later extended into a mutation testing tool for DL systems named DeepMutation++ [5]. This is a model-level, post-training mutation testing tool which manipulates a pre-trained model to produce its mutant versions. DeepMutation++ implements model-level operators from the ones introduced by Ma et al. [5] and 9 new operators designed for stateful recurrent neural networks (RNNs). RNN mutation operators are of two types: static and dynamic. Mutation operators acting on weights belong to the static group, whereas the operators associated with the dynamic group include the ones which mutate the run-time state and the ones which clear, shuffle and reduce gate values. In order to study the effect of the implemented mutation operators, the authors applied the '*Gaussian Fuzzing*' Operator to DNN models under different mutation ratios. As a result, the authors note that number of killed mutants increase

as there is an increase in applied mutation ratio. We used the first 8 operators of DeepMutation++ which act on neurons, weights and layers of the model, as representatives of post-training mutation operators in our case study.

MuNN [17] is another approach for the mutation testing of neural networks which contains five operators. These operators are also applied to an already trained model. The operators are *Delete Input Neuron*, *Delete Hidden Neuron*, *Change Activation Function*, *Change Bias Value* and *Change Weight Value*. The operators perform changes to the original model by deleting the neurons in input and hidden layers, changing biases, weights, and activation functions. These mutation operators are proposed by the authors after in-depth examination of how a of DL network functions differently from traditional code logic. The authors observe that to make a qualitative change in the output, it is required to mutate the characteristics of a neural network. For example, for a model which performs classification the final output is decided based on the data and parameters we provide during the training process. The continuous output is discretized to obtain the final result. So, even if the model is mutated, the mutation may affect the continuous values only slightly and thus the discrete value, i.e. final outcome may remain the same. In consequence, no observable change will take place. An observable change is said to happen only if there is a variation in the final discrete output. Thus, they state that this change takes place only if mutation is carried out on the characteristics of the trained model such as input dimension, the connection weights, the structure, the bias of neural network. The authors performed experiments to answer two main research questions. The first one is to study the effect of mutation operators on DNN models. The authors highlight that mutation analysis is strongly influenced by the characteristics of the domain. This further provides support for our proposed technique based on domain-specific metrics. The second research question was analysing the effect of neuron depth on mutation. The authors point out that the network depth is a sensitive factor in mutation testing as the mutation effects are gradually weakened with the deepening of the neurons. This is because there is a different influence depending on the depth of the injected mutation. For example, the authors introduced a mutation to DNN model where weight of the neurons are modified along with adding more hidden layers to the network. The authors notice that if more layers are added, the lower is the ratio of weight values to be modified to obtain the same mutation effect that they get without adding more layers to the model. This is due to the fact that the value produced by weight multiplication grows together with the increase of depth of neural network. This might indicate that neural networks of different depth require different ways of mutating them.

DeepCrime [7] is a mutation tool for deep learning systems, which contains 24 pre-training mutation operators rooted on real DL faults reported in 3 existing studies by Humbatova et al. [6], Islam et al. [8] and Zhang et al. [22]. During the extraction of operators from the studies of faults, the faults that lead to crashes are excluded as mutants generated by such mutation operators are immediately killable and will not help to evaluate the quality of a test set. The authors analyze two properties of their mutation operators. The first property states that the mutation operators they implement are killable. This means that there is at least one configuration of each mutation operator that makes that operator killable by the training dataset, i.e. the dataset to which the model is the most sensitive. The second property states that all operators are non-trivial, i.e. they are not killable by any random input.

In this tool, the mutation operators are categorized into groups such as Training Data Operators, Hyper-parameters Operators, Activation Function Operators, Regularisation Operators, Weights Operators, Loss Function Operators, Optimisation Operators and Validation Operators. The operators from the "Training" group mimic the issues related to the dataset used to train a DL system. For instance, the operator '*Change labels of training data*' alter the labels of training data. The operator '*Remove Portion of training data*' imitate a situation where enough data is not available for the training phase. '*Unbalance Training data operator*' deletes some data that belong to less frequent classes/categories, thus creating an imbalance of instances of classes within the data. The operator '*Make output classes overlap*' forces similar data belonging to the same class, to have different labels assigned to them. The operator '*Add Noise to Training Data*' introduces

some noise to reduce the quality of training data. Hyperparameter related operators simulate the choice of suboptimal values for the hyperparameters. For example, the operator '*Change number of epochs*' modifies the number of epochs for which the training process is carried out. The operator '*Decrease learning rate*' creates problems by introducing very low learning rates. The operator '*Change Batch Size*' changes the number of samples processed before the model is updated for each training stage. Activation function operators imitate wrong choices of activation function for specific layers in a model. As examples, the operator '*Remove Activation Function*' removes the activation function of a layer. In contrast, the operator '*Add activation function*' adds an activation function for a layer and the operator '*Change activation function*' changes the activation function of a layer randomly or as defined by the user. The operators belonging to the category '*Regularisation operators*' alter layer regularisations by adding, changing or removing regularisation. The operators belonging to '*Weight Operators*' add or remove bias values from/to a layer. The operator '*Change Loss function*' changes the loss function to a user specified or randomly chosen one. The '*Optimisation operators*' act on the original optimizers, changing them. And finally '*Validation Operators*' remove the validation data during the training stage.

The authors have assessed 20 mutation operators out of 24, in order to understand their characteristics, and confirmed that most of these operators are killable and non-trivial. Moreover, the authors showed that DeepCrime's mutation operators can very effectively discriminate between a weak and a strong test set. One of the research question (RQ2) the authors put forward was a comparative study of their framework to the DeepMutation++ tool. The authors carried out the experiment using a strong and weak test set in order to answer this question. They conclude that DeepCrime mutation operators are more likely to discriminate the test data quality. Also, authors identify that pre-training operators are more sensitive to changes in the quality of test data than post-training operators. The pre-training operators we use in our thesis are from DeepCrime tool.

2.3 Mutation Killing for Deep Learning Systems

While the usage of mutation testing for DL systems is potentially an effective technique, the familiar notions of mutation killing is not always easily transferable to DL systems. This is due to the difference in the nature of such systems, as well as to the randomness associated with their training process [9]. In fact, upon re-training, even the same neural network can have a slightly different performance. Following are some approaches proposed by researchers considering this problem.

In MuNN [17], the authors perform mutation killing in order to demonstrate the effectiveness of the mutation operators they propose. The impact of mutation operators is measured as the difference in accuracy of the original and mutated model. To account for the random nature of the operators, each mutant was run 10 times and the mean value of accuracies was calculated. However, the authors do not mention which threshold they used to consider the mutant killed.

In the work of Ma et al. [12] there is no practical notion of killing an individual mutant. Rather, the authors measure how many mutants were killed overall. This results in the calculation of the mutation score metric. The authors provide a definition of mutation score for classification systems. More specifically, in case of a k-classification problem with a set of classes $C = c_1, ..., c_k$, a test input $t \in T$ kills the pair (c, m) with class $c \in C$ and mutant $m \in M$, if t is correctly classified as c by the original model and if t is misclassified by the mutant model m. Based on this, the mutation score is calculated as the ratio of killed classes per mutant m over the product of the sizes of M and C.

$$MutationScore(T, M) = \frac{\sum_{m \in M} |KilledClasses(T, m)|}{|M| \times |C|}$$

For example, if the training data has 10 classes and there are 5 generated mutant models, and for each of them the test data is able to kill 3, 4, 5, 6 and 7 classes, the mutation score will be (7 + 5 + 3 + 4 + 6)/(5 × 10) = 0.48. This definition does not provide any information on whether a given single mutant is killed. Moreover, in their analysis the authors just computed the average values of mutation score across 5 runs. In addition, the authors compute the average error rate (AER) to ensure that the behavior differences introduced by mutants are not too large. Given a test data point $t \in T$ and Mutant model $m \in M$, the average error rate is defined as,

$$AveErrorRate(T, M) = \frac{\sum_{m \in M} |ErrorRate(T, m)|}{|M|}$$

If the average error rate is large for a generated mutant model, the authors remove those mutants for further analysis. The authors demonstrate that the data points lying close to the decision boundary of the original model are more likely to kill more mutant models. This results in increased mutation score and average error rate.

The DeepMutation++ [5] introduces a new notion of mutation killing for a test input. The authors propose two killing score metrics and use them to do the mutation analysis. For a mutant m that belongs to a set of mutant models M, original model n and a test input t, $KScore1$ is defined as follows: the mutant m is killed by the test input t if the output of the mutated model is different from that of the original model.

$$KS_1(t, n, M) = \frac{|\{m | m \in M \Lambda n(t) \neq m(t)\}|}{|M|}$$

The second metric $KScore2$, aims to analyse the RNN models by segments of data for example, segments of an audio input, at runtime. Killing of a mutant at the level of segment by $KScore2$ takes place if there is a divergence in prediction probability at the output of mutated RNN model given the i_{th} segment of data between mutated and original model. For the i_{th} segment t_i of a test input t, a mutant m in a set of mutant models M and original RNN model n, $Ksore2$ is defined as

$$KS_2(t_i, n, M) = \frac{\sum_{m \in M} ||prob(t_i, n) - prob(t_i, m)\}||_p}{|M|}$$

The less the values of $KScore1$ and $KScore2$, the more weak the test input is against the mutation.

Jahangirova & Tonella [9] propose a new definition of mutation killing which considers the stochastic nature of DL systems. Their technique requires applying the mutation multiple times, which they obtain by creating n random samples of training and testing data using cross-validation. Once n original and mutated models have been trained, the authors check if there is a statistically significant difference between their accuracies (i.e. p-value $<= 0.05$). Then they measure the effect size [10], i.e. the magnitude of the difference between two distributions, and require it to be non "negligible". Given the set of original models P, set of mutated models M and test data $TestD$, the following formula is adopted by the authors as mutation killing definition.

$$isKilled(P, M, TestD) = \begin{cases} true & \text{if } effectSize \geq 0.2 \\ & \text{and } p\_value < 0.05 \\ false & \text{otherwise} \end{cases}$$

To ensure that a high p-value is not due to insufficient statistical power, the authors also calculate statistical power (β) and make sure there is enough data points to reach $\beta > 0.95$. This definition of mutation killing is also adopted in DeepCrime [7].

One of the research questions (RQ2) of the study by Jahangirova & Tonella [9] is entirely dedicated

to the comparison between their new approach to mutation killing and existing approaches in literature. Specifically, the authors compare threshold based mutation killing to the proposed statistical way of killing using the values of accuracy. In order to perform comparison between the two approaches, they perform an experiment injecting mutations by DeepMutation[12], MuNN[17] to classification and regression DL systems. For classification systems they decide if a mutant is killed based on the difference between accuracies using three different thresholds (1%, 5%, 10%). For regression systems, a mutant is killed if the predicted value varies from the ground truth value more than a given threshold. The authors have used a self-driving car predicting steering angle for these experiments and considered three different threshold values (0.1, 0.25, 0.5). Thereafter, since there were n outcomes for threshold approach as n retrainings were needed by the statistical approach, while a single boolean value represents the output of the statistical approach, they had to introduce an agreement rate between the two approaches. The agreement rate is defined such that if the accuracy value is higher than threshold m times out of n retraining, the rate is set to m/n. For instance, if rate equals to 1, the mutant is always killed by threshold approach across n runs. The results show that in most cases with the threshold based approach the agreement rate was either set to 0 or it was rarely equal to 1. This indicates that the threshold approach often fluctuates its values across n retrainings, while the statistical approach deals well with the stochastic behavior of DL systems by definition. We adopted the proposed statistical killing as our criterion of mutation killing even when using system-level (as opposed to model level) values of quality metrics (i.e. domain specific values).

2.4 Mutation Testing of Autonomous Vehicles

Deep Learning (DL) is getting increasingly adopted to solve complex tasks in autonomous driving to make real-time driving decisions. Mutation testing can be beneficial to test deep learning based autonomous vehicles. In the literature there is only one targeted application of mutation testing to such systems [9]. Mainly, this work performed an empirical evaluation of mutation operators taking classification and regression systems as case studies. The authors use a DL model that predicts steering angle of a vehicle from the road images as a representative of a regression model for the evaluation of the DL-specific mutation operators implemented in DeepMutation [12] and MuNN [17]. Here, the performance of the autonomous vehicle is measured as the mean squared error (MSE) of steering angle prediction. The MSE measure the mean distance between original and predicted steering angles. The authors propose statistical definition of killing a mutant using accuracy values in the case of DL classification system and, using MSE values for regression system. However, a metric like MSE only indirectly characterizes the quality of driving. The authors do not provide any information on whether the mutants with an increased value of MSE lead to simulations which indeed demonstrate low quality of driving. In addition, they also perform threshold based mutation killing in context of autonomous vehicle, as an instance of their regression system case study. In the threshold based approach, they decide a mutant is killed if the predicted steering angle varies from the expected value more than the defined thresholds (0.1, 0.25, 0.5). However, the results showed that all mutants can be killed with this approach as there was always at least a single input image on which the definition of killing is satisfied (i.e. the prediction difference is greater than 0.1,0.25,0.5). In contrast, in our work we analyse system-level killing using driving quality metrics which strongly indicate the performance of driving, instead of offline context-unaware metrics.

Driving is a complex activity, the quality of which is not easy to quantify. Jahangirova et al.[3] provide a list of metrics that are specific to the driving task and have been used in the literature to evaluate the quality of driving performed by humans. The authors carried out a detailed study of the literature to identify driving quality related metrics from the studies involving driving tasks. The considered papers were extracted performing a query in Scopus, an abstract and citation database used by researchers and authors to find, locate or evaluate the research output from around the world. The search string for the query was formed by putting together terms related to driving quality. The authors manually analyzed each of the extracted papers in order to verify if they indeed evaluate the quality of driving using some

metrics. Then, a full text analysis was performed to extract the driving quality related metrics. The metrics also include information on events that takes place while driving. For example, an action taken by driver (*Brake*) and a consequent effect on the vehicle (*Collision*). The metrics are grouped into families such as generic metrics, speed related metrics, steering related metrics. *Speed*, measured as the distance travelled by a vehicle in a unit of time, is spotted as the most often used metric in the literature, when combined with operators such as Mean, Std (standard deviation) or Min(minimum). Another example of a popular metric is the *Steering Angle* (SA), defined as the angle between the front of the vehicle and the steered wheel direction and combined with the operator Std. For most of the primitive metrics there are aggregation operators that make them effective indicators, such as *minimum of speed*, *standard deviation of acceleration*, *mean of lateral speed*, etc. Other than that, some metrics come along with conditions. For example the metric *TimeC(Speed > 60)* computes the time for which speed was greater than 60. Afterwards, the authors performed a human study to analyze whether the values of the extracted metrics when applied to an autonomous vehicle correlate with the human perception of driving quality. To achieve this, the authors integrated a DL model into Udacity simulator and recorded videos of the car driving in different tracks. The videos reflect different ways of driving such as good driving, poor driving. The participants were asked to deliver their assessment using a 5-point Likert scale to evaluate the driving quality. Subsequently, the answers of participants were compared with the values of quality metrics recorded by the simulator during simulation in the same road sector, track, time frame. The authors used Pearson's correlation [11] and statistical tests in order to measure correlation between answers of participants and values of metrics recorded by the simulator. The metric with the highest correlation with human assigned scores is the *mean of the lateral position*, where lateral position is defined as the distance between the center of the car and the center of the driving lane. The analyzed metrics are then used to generate threshold-based oracles that aim to kill as many mutants as possible. The authors focus on oracle generation and therefore the effect of various mutation operators on each driving quality metric is not analyzed. Also, the authors didn't perform mutation killing using the values of quality metrics, which is instead a major contribution of this thesis.

2.5 Offline and Online Testing

The existing literature defines two general ways to test DNN based systems: Offline Testing and Online Testing [4]. Offline testing is an approach where DNNs are tested in 'isolation' using datasets that were acquired independently of the tested DNN. Online testing, on the contrary, assesses the network's behaviour during the constant interaction with the environment of the application into which the DNN under test is integrated. An example of offline testing could be measuring the error in the DNN's prediction of a steering angle based on the dataset of images compiled beforehand, while an example of online testing could be evaluating the processing of subsequent test inputs either in a simulator or in a real-world setting. This way it is possible to track the dependence of the behaviour of the system from the outcomes of the DNN and to understand whether the system is behaving in line with the safety/quality requirements.

Haq et al.[4] differentiate offline and online testing. The authors performed a case study to identify similarities and dissimilarities between the two DNN testing approaches. They conducted the experiment in the domain of autonomous vehicles. Online testing is carried out by integrating DNN models which predict the steering angle into a simulator. The technique called *end-to end control of a vehicle* is adopted in a context where the DNN generates the next command for the vehicle actuators after receiving the images from camera, to enable automated driving. On the other hand, offline testing is performed by evaluating the DNN model under test by using pre-prepared datasets (datasets generated by simulators and datasets built from real-life driving).

The results of their case study show that while offline testing appears to be faster and cheaper, online

testing discovers safety violations that are not reflected in prediction errors during offline testing. Moreover, large prediction errors obtained with offline testing do not always lead to the detection of safety violations in online testing. These findings suggest the practical importance of online testing in addition to the more widely used offline approach and provide motivation for the comparison of the drop of performance of a DL model to the online violation of oracle metrics and safety constraints during the simulation. The terms offline and online testing in their work correspond to model-level and system-level testing in our study respectively. The main research question they raise is the difference between offline and online testing, which is in line with what we do in this thesis. However, our work focuses on comparison of the impact of fault injection into a DL model setting at the model and system level. Differently from this work, the authors use only Maximum Distance from Center of Lane (MDCL) in online testing to characterise the quality of driving while we use more metrics like crashes, out-of-bounds, standard deviation of the steering angle, maximum speed etc. For the evaluation in offline testing, they use Mean Absolute Error (MAE) to evaluate the model's performance, whereas our study focuses on Mean Squared Error(MSE) as performance indicator at the model-level but more importantly such a metric is further evaluated in a statistical way in our study. Hence, another important difference is that we have a statistical notion of mutation killing i.e. misbehavior detection at both model and system level, while they set threshold values for MAE and MDCL to decide if the prediction indicates a misbehavior in the comparison between online and offline testing.

Chapter 3

System-Level Detection of Misbehaviours

To date, all proposed approaches of mutation testing for DL systems are applied only at the model-level. Model-level mutation killing relies on measuring the drop in the value of model's performance indicator, for example, accuracy of the DL model of interest. Moreover, the model-level approach is carried out in a setting where neural networks are tested in isolation using static datasets. Whereas, the main idea of this thesis is to perform system-level mutation testing by deploying the DL systems in the environments they are designed to be used. This will allow us to investigate whether the proposed mutation operators are successful in injecting faults that cause misbehaviours at the system level. Moreover, analysing whether the visibility of faults varies between model and system level will provide an insight into the necessity of system level testing for DL systems.

Note that our approach requires the definition of a set of quality metrics that will assess the behaviour of the DL model at the system level. These metrics are domain dependent.

3.1 The Definition of System-Level Killing

Let P be our original DL model. Let's assume we have m quality metrics $QM = QM_1, ..., QM_m$ that assess how well P operates when deployed in its corresponding environment. To account for the stochastic nature of DL systems, we execute the training process of P and its mutant M n times, getting a set of original models $P = P_1, ..., P_n$ and a set of mutants $M = M_1, ..., M_n$. Then, we collect the values of quality metrics for each element in P and M, obtaining the values of each quality metrics QM_i for the original and mutant models respectively: $QM_{iP} = QM_{iP_1}.....QM_{iP_n}$ and $QM_{iM} = QM_{iM_1}.....QM_{iM_n}$, where $i = (1..m)$. We say that the mutant M is killed if there exists at least one quality metric QM_j for which the difference of its values in the original and mutant models is statistically significant, i.e. the p-value is less than 0.05. Moreover, we require that the effect size is not negligible.

$$isKilled(QM_{j_P}, QM_{j_M}) = \begin{cases} True, & \text{if } effectSize(QM_{j_P}, QM_{j_M}) \geq 0.5 \\ & \text{and } p\_value(QM_{j_P}, QM_{j_M}) < 0.05 \\ False, & \text{otherwise} \end{cases}$$

In our experiments, we use 20 as the value of n. Generalized linear model (GLM) [13] is used for the calculation of statistical significance. To measure effect size, i.e. to determine if the difference obtained is negligible or considerable, we use Cohen's d [10].

$$cohen's\ d = \frac{\overline{X_1} - \overline{X_2}}{\sqrt{\frac{s_1^2 + s_2^2}{2}}}$$

where, $\overline{X_1}$ and $\overline{X_2}$ are the mean of original model's metric values list and mutated model's metric values list respectively. And, s_1 and s_2 are standard deviation of original model's metric values list and mutated

model's metric values list respectively. The qualitative interpretation of the magnitude of the effect size [10] can be derived from the table 3.1.

negligible	$\|d\| < 0.2$
small	$\|d\| < 0.5$
medium	$\|d\| < 0.8$
large	$\|d\| \geq 0.8$

TABLE 3.1: Interpretation of Cohen's d: magnitude of the effect size

To explain our approach more in detail suppose we have an original DL model to predict a steering angle for an autonomous vehicle given an image of a road and its mutated version created by injecting the *'Remove portion of training data'* operator. To account for stochastic nature of DL models, we train both the original and mutation 20 times. The columns "*MSE values of original model*" and "*MSE values of mutated model*" in Table 3.2 contain 20 values of Mean Squared Error (MSE) of steering angle prediction for the original and mutated instances respectively. Subsequently, we apply the formula for statistical model-level killing and compute the *p*-value and effect size. As reported in column *isKilled* the mutant is not killed at model-level.

Killing at the system level evaluates the behaviour of a DL system using a set of domain-specific quality measures. Therefore, in our case we use three example quality metrics in a setting of autonomous driving which are listed in column "*Metric*" of Table 3.3. These values are obtained by running each of the 20 original and mutant models in the simulator and recording the values of each metric. The columns *'Metric values of original model'* and *'Metric values of mutated model'* in Table 3.3 report the 20 values for each metric. We then apply the formula for statistical killing to each of the metrics and compute the *p*-value and effect size. As the Table 3.3 reports, mutant is killed by two of the metrics and is not killed by the metric *Standard Deviation of Speed*. For the metric which calculates the *Mean of Steering Angle* and *Maximum of Lateral Position*, one can note the values of *p*-value and effect size in columns "*p-value*" and "*EffectSize*" respectively and confirm that the mutant is killed by the statistical test.

As can be observed, the injected mutation is not exposed using a performance indicator (MSE) of the model, but leads to the statistically significant change in the values of two quality metrics.

MSE values of original model	**MSE values of mutated model**	***p*-value**	**EffectSize**	**isKilled**
[0.6, 3.7, 3.0, 0.2, 1.26, 1.96, 0.67, 1.33, 2.01, 3.5, 2.31, 0.04, 1.44, 0.20, 1.01, 0.34, 1.77, 2.10, 2.03, 0.07]	[0.58, 3.1, 3.4, 0.26, 1.29, 1.96, 1.20, 0.12, 1.54, 0.199,1.05, 0.1, 1.49, 0.25, 1.05, 0.38 1.81, 2.91, 2.03, 0.19]	0.39	0.12	False

TABLE 3.2: An example of model-level killing using values of MSE

Metric	Metric values of original model	Metric values of mutated model	p-value	EffectSize	IsKilled
Maximum of Lateral Position	[1.2401, 1.0512, 1.0512, 1.0512, 1.0512, 1.0768, 1.1253, 1.0512, 1.0512, 1.0512, 1.0512, 1.0512, 1.0512, 1.1393, 1.0512, 1.0512, 1.0512, 1.0512, 1.0512, 1.0512]	[1.1981, 1.2832, 1.444, 1.0512, 1.0512, 1.0512, 1.8644, 1.0512, 1.0512, 1.0512, 1.0602, 1.0512, 1.0512, 1.0512, 1.0512, 1.0512, 1.0512, 1.3455, 1.5342, 1.1987]	0.033	0.67	True
Mean of Steering Angle	[-0.0266, -0.007, -0.0443, -0.069, 5.556e-05, -0.05, -0.038, -0.030, -0.044, -0.003, 0.0006, -0.003, -0.050, -0.020, -0.023, 0.0044, -0.05, -0.053, -0.022, -0.053]	[-0.0178, -0.028, -0.035, -0.0238, -0.04671, -0.0209, 0.036, 0.0011, -0.0142, -0.0121, 0.0051, 0.0038, 0.0090, 0.0247, -0.0335, -0.030, -0.017, -0.0437, -0.0447, 0.0083]	0.036	0.69	True
Standard Deviation of Speed	[0.038, 0.036, 0.0341, 0.0578, 0.092, 0.1061, 0.1310, 0.040, 0.062, 0.1517, 0.0346, 0.0778, 0.079, 0.061, 0.1050, 0.0510, 0.0335, 0.0554, 0.063, 0.0533]	[0.0750, 0.028, 0.0728, 0.076, 0.081, 0.2018, 0.038, 0.064, 0.051, 0.065, 0.275, 0.307, 0.04, 0.082, 0.083, 0.048, 0.067, 0.028, 0.065, 0.0478]	0.251	0.36	False

TABLE 3.3: An example of system-level killing using quality metric values

3.2 Case Study

In this thesis we apply the idea of system-level killing to a case study of a DL-based autonomous vehicle. To the best of our knowledge, no existing approach has applied system-level mutation testing to autonomous cars. Rather, previous works in mutation testing of DL systems were limited to the model-level definition which relies on performance indicators such as Mean Squared Error (MSE), Mean Absolute Error (MAE) of steering angle prediction. Therefore, these approaches only evaluate how injected faults affect DL model's steering angle predictions for a given set of images. However, they lack the evaluation of injected faults' impact on the DL-based autonomous vehicle's actual quality of driving.

We perform our experiments by embedding our DL model in Udacity simulator, where it predicts subsequent actions by processing images received from camera while driving in the simulated environment, therefore we perform system-level testing. The work by Jahangirova et al. [3] contributes a list of quality metrics to the literature, through which high or low quality of driving can be detected. We adopt these metrics in our study as the quality metrics for the system level behaviour of the autonomous vehicle. We picked quality metrics instead of relying only on the safety oracle i.e., collisions with other vehicles, pedestrian etc., because we needed a fine-grained set of metrics that reveal more about driving quality. Our system-level mutation killing is particularly interesting for autonomous vehicles, as their system-level behavior might be different from what is measured at the model-level due to complex interactions with the surrounding environment.

3.2.1 Udacity Simulator

We use Udacity simulator which is an open source simulator for self-driving cars. The simulator contains different tracks and two modes, namely, training mode and autonomous mode (Figure 3.1). In training mode the car can be driven manually with the use of a keyboard to collect the training data which then can be used to train the DL model. In contrast, in autonomous mode the car is driven by an already trained DL model where we can test and see how a DL model performs after being trained with collected data. There are three closed-loop tracks available in the simulator. We use the default "Lake Track" (Figure 3.2) which features a wide lane, has gentle bends and a bridge. As the name of the track suggests, the road for driving is placed at the shore of a lake. So if the DL model does not perform its task well, the car has a risk of falling into the lake.

3.2.2 Deep Learning Model and Dataset

We chose DAVE-2 as our deep learning model, since it is publicly available and an extensively used model in DNN testing papers [14] [20] [21] [19]. DAVE-2 gives realistic behaviors in simulated platforms, as well as realistic performance degradation when not appropriately trained [19], or when the model's architecture or training data get corrupted [9]. The model (Figure 3.3) consist of three 5x5 convolutional layers with stride 2 plus two 3x3 convolutional layers, followed by five fully-connected layers with dropout and ReLu activation function. Images are provided to the model which then produces steering angle predictions. The produced predictions are compared to desired outcomes for that image and the weights of the CNN are adjusted to bring the CNN output closer to the desired output.

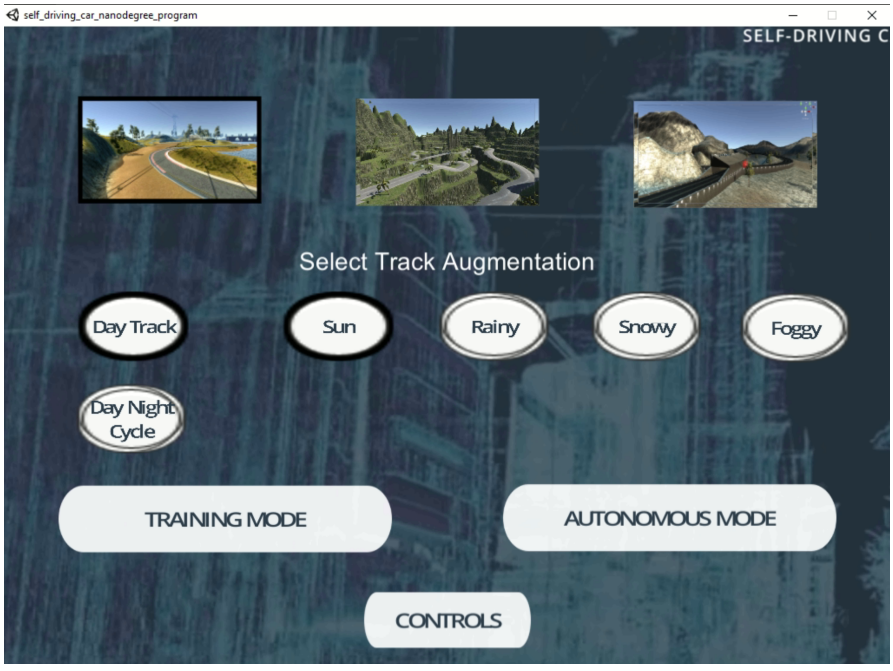

FIGURE 3.1: The starting settings window of Udacity Simulator.

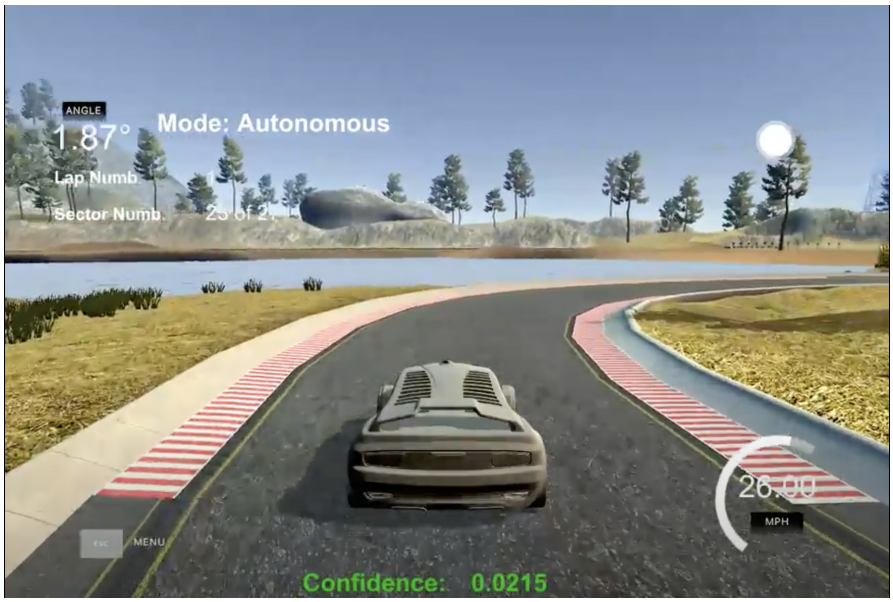

FIGURE 3.2: Udacity Simulator. A scene taken from Lake Track

For the training we use the dataset from the work by Stocco et al. [19] which was obtained by driving the car in training mode in the Udacity simulator for 10 laps on the track, using two different track orientations: normal and reverse. In total, the authors generated a dataset of 23,905 training images (at 10-13 frames per second). In order to permit a smooth driving and a rectified behavior capture (i.e., lane keeping), the maximum driving speed was set to 30 mph, the default within the Udacity simulator.

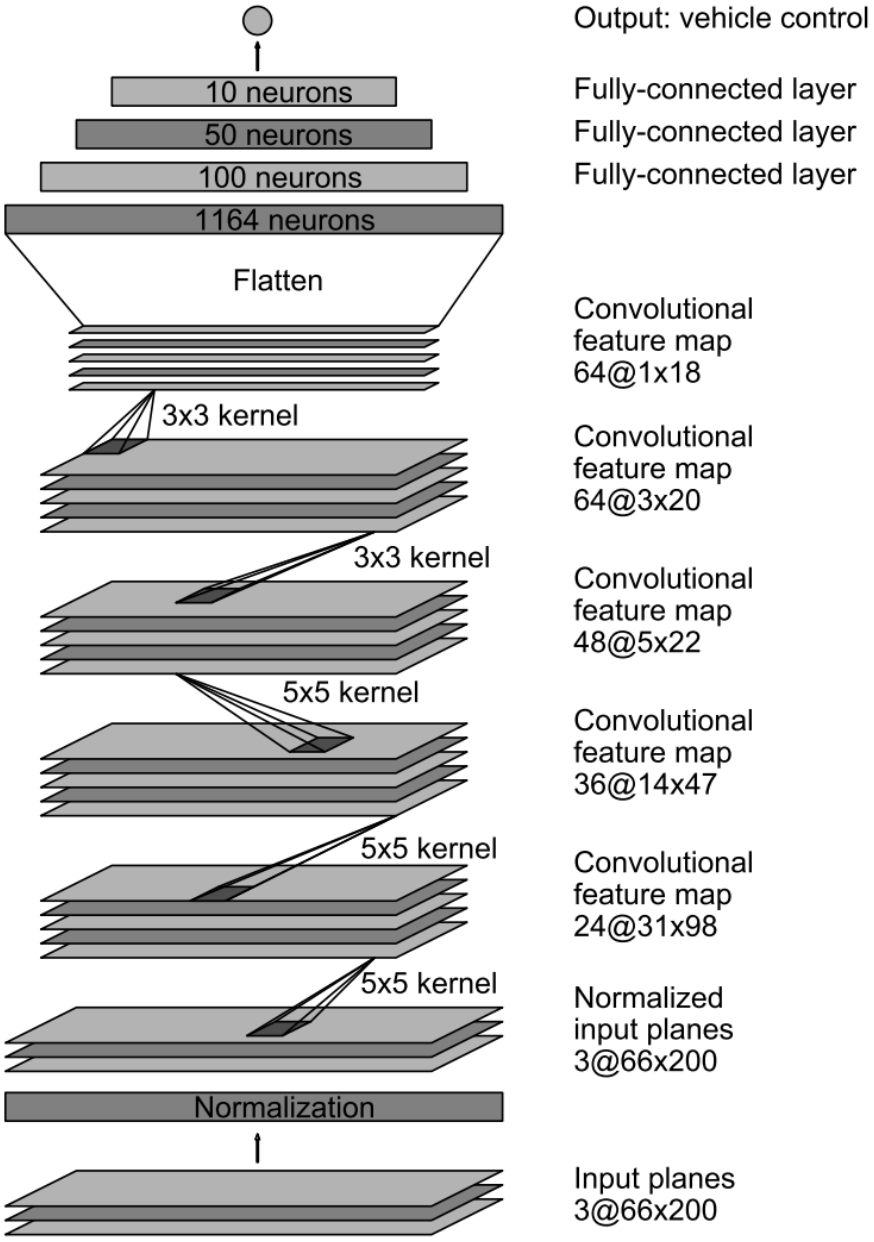

FIGURE 3.3: NVIDIA DAVE-2 model architecture [1]

Chapter 4

Experimental Procedure

The experimental procedure carried out in this thesis is summarized in Figure 4.1 and 4.2. Figure 4.1 shows the steps that are needed in order to perform system-level killing. The first step is to use the DeepCrime and DeepMutation operators to generate mutant versions of the original model. After that, simulation is run for each original and mutant models, integrating them in the Udacity simulator in order to collect values of quality metrics. Following that, we do a statistical definition of mutation killing relying on the quality metrics values obtained. Then in the last step, we try to extract some useful and effective metrics based on how they behave to mutants.

Figure 4.2 illustrates the phases involved in model-level killing. The first procedure is the same as in the previous case, wherein we obtain 20 original and mutated model instances.While in the second step, we decide whether a mutant is killed by running a statistical test of mean squared error values, of original and mutated models against the test dataset. The sections that follow provide detailed information on each stage.

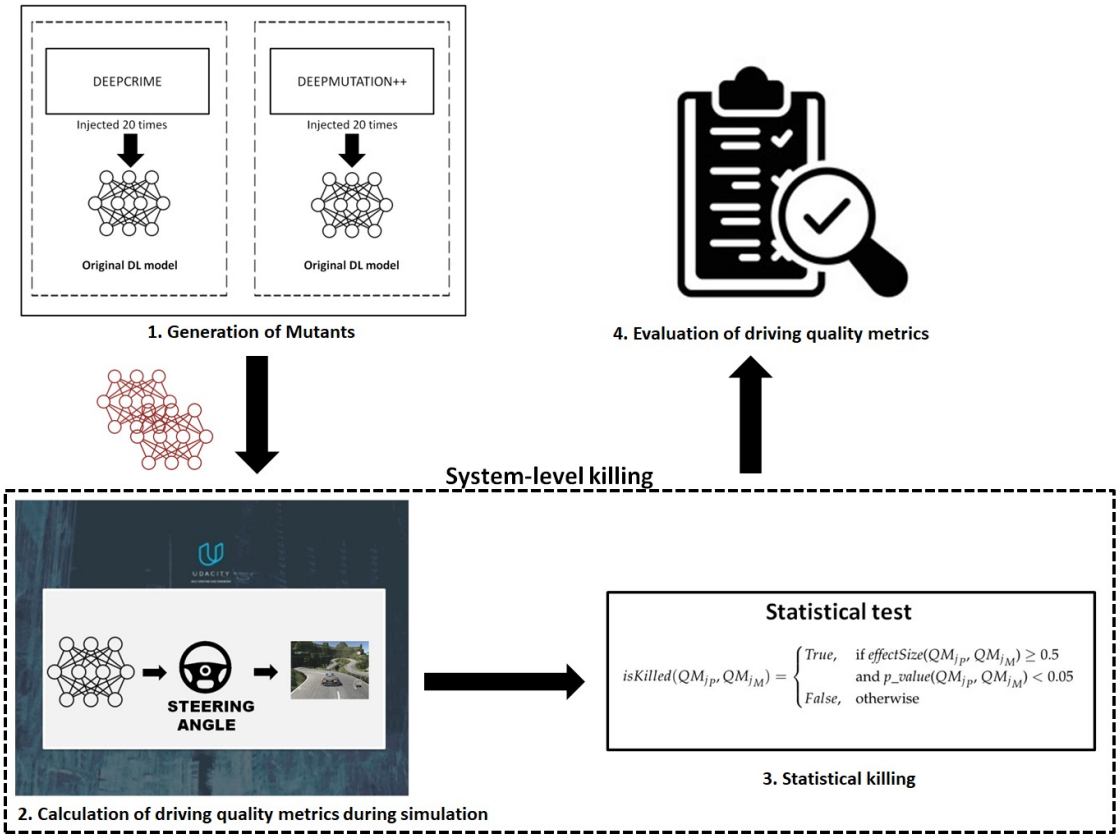

FIGURE 4.1: Overall experimental procedure for system-level killing

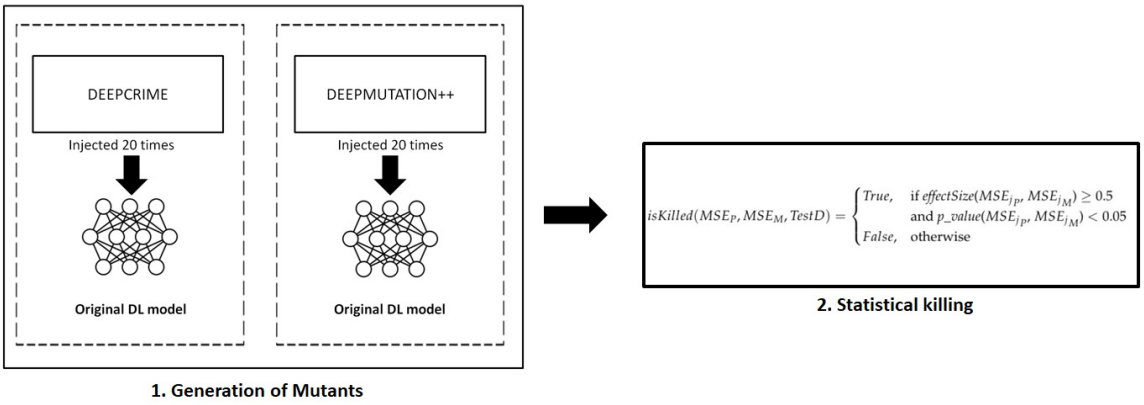

FIGURE 4.2: Overall experimental procedure for model-level killing

4.1 Generation of Mutants

For our experiments we used two state-of-the art tools for mutation testing of deep learning systems: DeepCrime [7] and DeepMutation++ [5].

4.1.1 DeepCrime

There are three large scale studies available in literature that have explored real faults which can arise in DL systems [6] [22] [8]. DeepCrime is the first open source, Python-code level mutation tool, based on real faults built from the analysis of faults extracted from these studies. DeepCrime operators are pre-training operators, i.e. they are injected before the training process. The main idea of applying DeepCrime mutation operators is shown in Figure 4.3. The training program and a set of training data are prepared during the setup phase. The original DL model is created after the training process by running a training program on training data. Then, either the original training data or original training program are mutated by injecting mutation operators. Therefore, to generate each mutant we need to re-train the DL model.

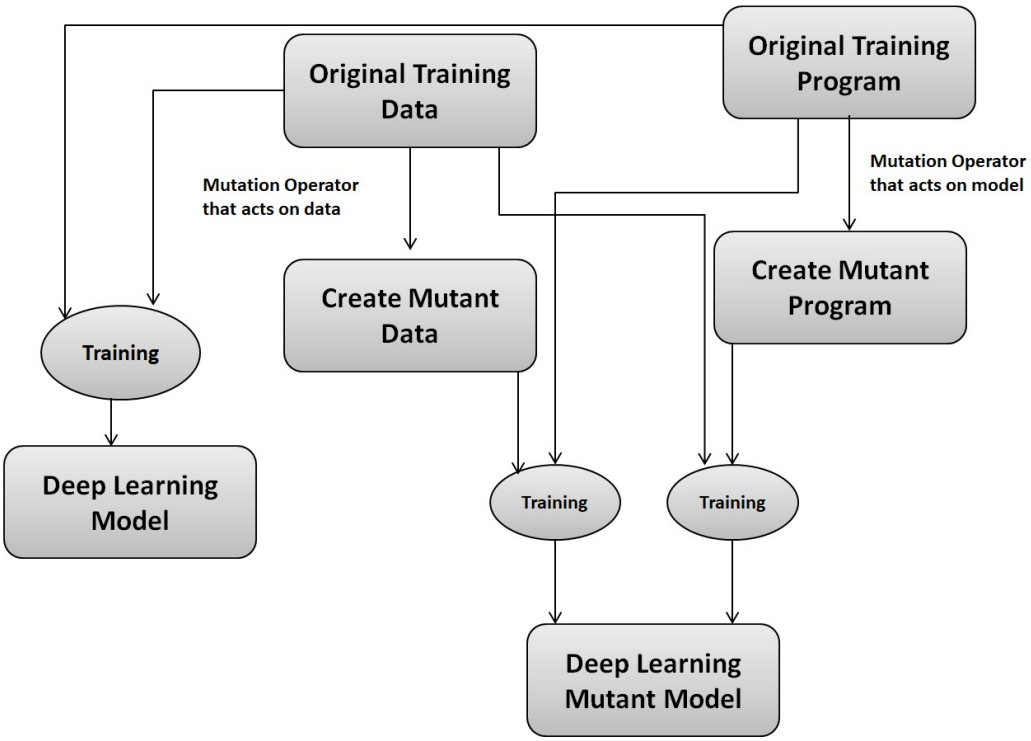

FIGURE 4.3: Process of injection of pre-training DeepCrime mutation operators

Table 4.1, 4.2 summarizes all the DeepCrime mutation operators applied to the Udacity model as well as the parameters of each operator. The column "Group" lists the group to which the mutation operators belongs. The operators are organized into groups on the basis of their scope of application. For example, the operators changing the labels of training data, deleting portion of training data, etc., belong to the group of training data operators. These operators alter the training dataset with the aim of mimicking the possible problems that can occur in collection of data used for the training process of neural networks. The column parameters lists the applicable parameters for each operator. Most of the operators proposed in DeepCrime have parameters that need to be configured. For example, for the mutation operator '*Remove Portion of Training Data*', the percentage of the training data to delete should be specified as a parameter. If the values for parameters are not provided explicitly, the tool performs an automated search in the space of possible values for the parameter. Finally, the column 'ST' indicates which type of search can be performed for each parameter.

The applicable search method for each mutation operator parameter is determined based on whether the parameter is range-based, list-based or user-specified. The *range-based* group includes the parameters, the possible values of which are limited to a specific range. For instance, the parameter 'percentage' of '*Change Label*' operator indicates the portion of training data for which to change labels, and it can have a value in the range from 0 to 100. Furthermore, in order for the parameter to be considered range-based, its impact on the output of the operator should grow monotonically with higher/lower values of the parameter. In case of the '*Change Label*' operator, a higher percentage of changed labels produces a more destructive mutation. In contrast, if the possible parameter values are confined to a predetermined list, they fit the list-based group. For example, the operator *'Change Activation Function'* has a parameter *'new activation function'* the value of which can be any activation function available in Keras. The parameters, for which the values need to be specified by a user in an ad-hoc manner, are included in the *user-specified* group. For example, the dropout rate takes values in the range (0,1], but at the same time it doesn't have increasing effect across the range, hence it is not range-based. Moreover, there is not a finite list of values

that this parameter can take. As a result, the list of values for this parameter can only be provided by the end user of the tool.

For range-based parameters we apply *binary search*. For example, for ''*Remove Portion of Training Data*'' operator, if it is killed at 5% and 15% then it means it is killed for all values between 5% and 15%. The idea of binary search is to find a point in the parameter value space where the mutation switches from "not killed" to "killed". The algorithm first checks if the mutation operator is killable at the middle point of the range. If it is killable at this point, the search is applied to the first portion of the range. Otherwise, it shifts to the second portion of the range. This operation continues recursively till the range value becomes less than or equal to a pre-selected precision ϵ, which gives the granularity of the smallest possible change percentage to be performed.

For mutation operators with list-based parameters, we performed an exhaustive search. However, if 2 parameters of an operator need an exhaustive search, and one of the parameters is a layer, we chose the layer number randomly and performed exhaustive search on the other parameter. For example, for the '*Change Activation Function*' operator, the layer is chosen randomly, and each possible activation function was applied for the *new activation function* parameter. For user-specified parameters, we defined a set of values and applied each of them. In order to acquire values for the operator '*Change dropout rate*', the layer is chosen randomly, and the new dropout rate is obtained by performing exhaustive search on 4 pre-defined values distributed evenly in the range (0,1].

For the mutation operators having range-based parameters, the range used is from 0 to 1.0. The only exception is for the operator '*Remove Portion of Training Data*' in which range is set from 0 to 0.99 since it is not possible to remove the whole training data. For '*Change Learning Rate*' operator the upper bound is set to the learning rate in the DNN model while the lower bound is set to a number close to 0. For the '*Change Number of Epochs*' operator, the upper bound is set to the number of epochs in the DNN model and the lower bound is set to 1.

Overall, we got 98 mutants by supplying different parameters to 20 applicable operators of DeepCrime. As we trained 20 instances for each mutant, 1960 (98 * 20) mutant instances were generated for our case study. We used an Alienware Aurora R8 (3.60 GHz Intel Core i9-9900K, 8 cores, 32 GB RAM, NVIDIA GeForce RTX 2080 Ti 11 GB) machine for these experiments.

4.1.2 DeepMutation++

The second set of mutation operators we used are from the DeepMutation++ tool that has eight model-level operators for feed-forward neural networks (FNNs) and additional nine operators specialized for stateful recurrent neural networks (RNNs). The operators vary between weight, neuron and layer level applications and mutate the parameters and structure of a DNN model. The mutation operators applied in our work are the first eight operators proposed for feed-forward neural networks (FNN). These operators are post-training i.e. operators are injected to an already trained model. The brief idea of applying DeepMutation++ mutation operators is shown in Figure 4.4. Unlike Deepcrime, which modifies the original training data and training program, Deepmutation mutate directly the model obtained from the training process using original data and program.

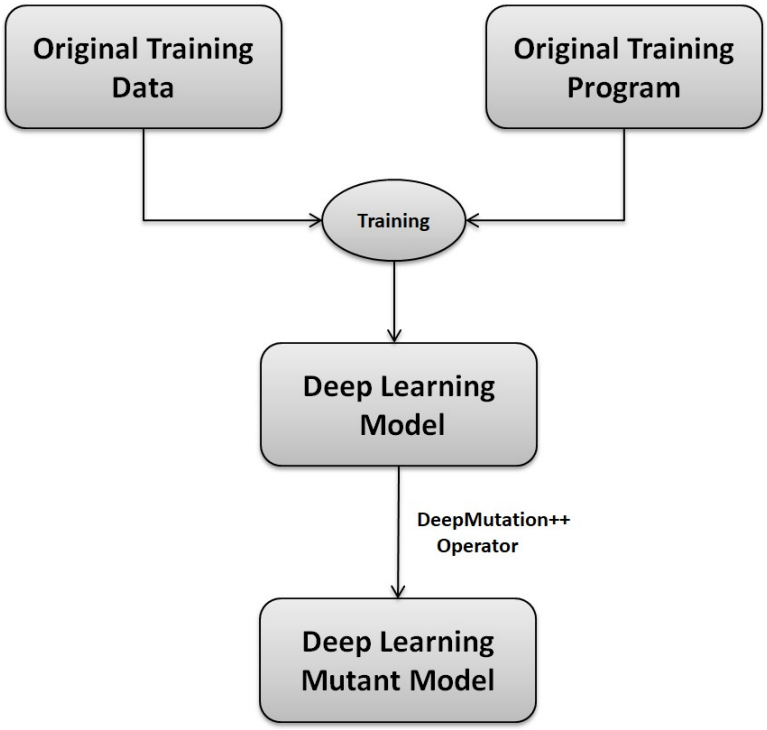

FIGURE 4.4: Process of injection of post-training DeepMutation++ mutation operators

Table 4.3 lists the DeepMutation++ operators used in our work. The operator *'Gaussian Fuzzing'* (GF) acts on the decision logic of DNNs, i.e. the weights which reflect the connection between neurons. It uses a Gaussian distribution to mutate the value of weights, thus changing the connection importance between neurons.The operator *'Weight Shuffle'* (WS) shuffles the weights of randomly chosen neurons to mutate current weights. The operator *'Neuron Effect Blocking'* (NEB) blocks the propagation of the results computed by a neuron to its next layers by resetting the connection weights of the next layers to zero. In this way the contribution of a neuron is removed which may affect the final decision of the neural network. The operator *'Neuron Activation Inverse'* (NAI) inverts the activation status i.e. changes the sign of the output value of a neuron prior to application of its activation function. The operator *'Neuron Switch'* (NS) switches two neurons inside the same layer so as to swap their roles which can influence their subsequent layers. The operator *'Layer Remove'*(LR) removes a layer of the model, whereas the operator *'Layer Addition'* (LA) adds a layer. The operator *'Layer Duplication'* (LD) inserts a copy of an original layer. The three last mentioned operators LR, LD and LA can damage the model structure.

To use the tool, the user only needs to provide some command options to choose mutation operators and the path to original model. A user-specified ratio can be used for the operators GF, WS, NEB, NAI and NS to indicate which portion of weights/neurons should be affected by the mutation. We applied the mutation ratio values of 0.01, 0.03, 0.05 for these operators. The values of ratios are chosen such that they are the ones used in the original DeepMutation++ [5] work, thus we assume that these are the most appropriate values. For the operators LR, LA and LD we did three runs so that the operators are applied to different layers. Overall, 24 mutants were obtained. To account for the stochastic nature of the mutation operators and of DL training, we applied each mutant to each of the 20 re-trainings of the original model, therefore getting 20 instances of each mutant.

Group	Operator	Parameters	ST
Training Data	Change labels of training data	Percentage — a percentage of data for a given label to mutate	B
	Unbalance training data	Percentage — a percentage of training data of underrepresented/selected labels to remove in order to unbalance the training data	B
	Make output classes overlap	Percentage — a percentage of training data to mutate	B
Hyperparameters	Change learning rate	new learning rate — new learning rate to be used to train the system under test	B
	Change number of epochs	new number of epochs — new number of epochs to be used to train the system under test	B
Activation Function	Change activation function	layer — number of the layer with non-linear activation function to mutate; new activation function — new activation function for the layer under mutation	EL, EL
	Remove activation function	layer — number of the layer mutate	EL

TABLE 4.1: DeepCrime Operators. Column "ST" indicates the type of search used (B = Binary search; EL = Exhaustive on list; EU = Exhaustive on user provided values).

4.2 Calculation of Driving Quality Metrics

In order to assess the effect of applied mutation operators, we extract the values of driving quality metrics during simulation of trained model. We adopt the metrics proposed in the existing work [3]. Table 4.4 lists the 19 metrics we used in our experiments. These metrics are directly recorded for each frame by the extended Udacity simulator provided in the work by Jahangirova et al. [3]. The speed takes a value between 16 and 48 km/hr. The steering angle (SA) value predicted by the DNN model ranges from -25 to +25 degrees. Calculation of linear interpolation between the minimum and maximum speed, ensuring the car decreases the speed when the steering angle increases for instance in a curve, allows us to get the value for throttle (TPP). The metrics related to acceleration (ACC), lateral speed (LS), change of steering angle per unit of time (SAS) requires knowledge of the minimum unit of time; this was obtained by dividing the total time of the simulation by the number of frames. The metric lane position is calculated by taking the distance from the center of the car's cruising position to the center of the road on the ideal trajectory between the planned route given by two consecutive waypoints [18].

Using an automated script, we run a separate simulation for each original and mutant models. For each simulation, data is collected into an Excel format file containing the values of each quality metric for 28 sectors of the road. If the metrics data is missing for any of the 28 sectors of road or any of 20 instances of the mutated model, those mutation operators are not considered for further analysis. This missing information is due to the fact that the mutant corrupts the model in a way such that it is not able to drive properly. In the case of DeepCrime tool, 12 mutants out of 98 had missing information on any of the 28 sectors or 20 instances of the mutated model. For DeepMutation++ tool, 14 mutants out of 24 had incomplete information. Therefore, we performed our further analysis on 96 remaining mutants.

Group	Operator	Parameters	ST
Regularisation	Add weights regularisation	layer — number of the layer with no weights initialisation to mutate; new weights regulations — type of weights regularisation to be added for the layer under mutation	EL, EL
	Change dropout rate	layer — number of the dropout layer; new dropout rate — new dropout rate for the layer under mutation	EL, EU
Weights	Change weights initialisation	layer — number of the layer to mutate; new weights initialisation — new type of kernel initialiser for the layer under mutation	EL, EL
	Remove bias from a layer	layer — number of the layer with bias to mutate	EL
Loss function	Change loss function	new loss function — new loss function to be used to train the system under test	EL
Optimisation Function	Change optimisation function	new optimisation function — new optimisation function to be used to train the system under test	EL
Validation	Remove validation set	_	_

TABLE 4.2: DeepCrime Operators Cont. Column "ST" indicates the type of search used (B = Binary search; EL = Exhaustive on list; EU = Exhaustive on user provided values).

4.3 System-Level Killing

Existing mutation killing criteria for DL systems are based on the performance indicator of the DL model under test. We call such an approach a model-level killing. In contrast, in this thesis we propose to use system-level killing where killing is defined based on metrics obtained at the system level rather than at the DNN model level. Once we generated 20 instances of each mutant, we obtain the values of the metrics for each instance from the launched simulations. This is achieved by analyzing the 20 values obtained for each quality metric retrieved from 20 retrainings of each mutation operator.

We start the analysis by organizing the collected metrics into 3 groups : mutants which have crashes or out-of-bounds on all 20 models (Group 1); mutants which do not have any crashes or out-of-bounds (Group 2); mutants which have crashes or out-of-bounds in some of 20 models (Group 3). This is done simply by taking the counts of metrics Crashes and Out-of-bounds. The mutants belonging to Group 1 are considered to be killed based on the idea that, if all 20 instances of the mutant lead to a crash or getting a car out of lane bounds, then it has definitely introduced a faulty behavior. The remaining Group 2 and Group 3 mutants are subject to a more detailed analysis on whether they are killed or not using the values of the remaining quality metrics.

For each mutant in Group 2 and Group 3, we apply the definition of statistical mutation killing using 20 values of each metric. For each sector, we do statistical test between 20 values of quality metrics of the mutated model and 20 values of metrics of the original model, iterated 28 times (there are 28 sectors of road in total). If there is at least one sector where the difference of values of original and mutated model's metric is statistically significant, i.e. if p-value $<= 0.05$ and if the effect-size is not negligible the mutant is considered killed. For some metrics their value should increase if the quality of driving becomes worse and for some their value should decrease, i.e. the effect size should either be positive or negative. In the work by Jahangirova et al. [3] one of the research question analyses which quality metrics correlate with human assessed quality of driving. The authors collected human assessment of the driving videos in a

Mutation Operator	Level	Description
Gaussian Fuzzing (GF)	Weight	Fuzz weight by Gaussian Distribution
Weight Shuffling (WS)	Neuron	Shuffle selected weights
Neuron Effect Block. (NEB)	Neuron	Block a neuron effect on following layers
Neuron Activation Inverse (NAI)	Neuron	Invert the activation status of a neuron
Neuron Switch (NS)	Layer	Switch two neurons of the same layer
Layer Deactivation (LD)	Layer	Deactivate the effects of a layer
Layer Addition (LA)	Layer	Add a layer in neuron network
Layer Removal (LR)	Layer	Remove a layer in neuron network

TABLE 4.3: DeepMutation++ Operators.

Group	Abbr.	Metric Name	Operators
Generic	_ _	Crash Out of Lane Bounds	Count
Speed	_ ACC TPP	Speed Acceleration Throttle Pedal Position	Max, Std Max, Min, Mean, Std Mean, Std
Lateral Position	LS LP	Lateral Speed Lateral Position	Mean, Std Max, Mean, Min, Std
Steering	SA SAS	Steering Angle SA Speed	Max, Mean, Std Mean, Std

TABLE 4.4: Driving Quality metrics applied in our work

simulated environment and computed the correlation between each metric and the average human score. We use these correlation coefficients (*cc*) to determine whether the value of the metric should increase or decrease. If the value of *cc* is negative, we use negative effect size and if the value is positive, we take the positive one.

4.4 Evaluation of Driving Quality Metrics

We provide a comprehensive evaluation of the driving quality metrics based on their ability to detect system-level behavioural differences. For each metric we determine the number of mutants it kills and order the metrics according to their mutation killing capability. The metrics with the highest misbehaviour detection rate are the most sensitive to the faults that can take place in autonomous vehicles.

Moreover, we determine the minimal set of metrics required to kill all the mutants. For this, we first identify the metric that kills the highest number of mutants. We then exclude all mutants killed by that metric from the killed mutants list, and look for the next metric that kills the highest number of mutants. We continue this process in an iterative way and stop once there are no killed mutants left in the list.

4.5 Data Collection and Analysis

We collect the data by using the publicly available code for DeepCrime and DeepMutation++ and the publicly available version of improved Udacity simulator that records the values of required quality metrics. Once we generate all the mutants, we use the SikuliX[1] tool to automatically run the simulation for each

[1] http://www.sikulix.com/

original and mutant model. SikuliX is a tool that locates images on a screen (for example, the logo of the simulator) and can run the mouse and the keyboard to interact with the identified GUI elements. It identifies GUI elements using image recognition powered by OpenCV.

The code required to perform the data analysis such as calculating mutation killing using values of metrics, analysis of driving quality metrics, performing comparison of our approach with existing model-level killing approach and generating the summary of the results, is implemented in Python version 3.6.8. One of the main reasons we chose Python for our work is that the the mutation tools we use are also implemented in Python. Moreover, the code required to run the Udacity simulator is also written in Python. Other reasons include the availability of numerous tools and frameworks, which simplifies the data analysis process.

The data obtained from two tools DeepCrime and DeepMutation++ is analysed in separate scripts. However, the functions that are required for the analysis of both tools are kept in the common script file *utilities.py*. For example, the function to extract minimal set of metrics that kills all mutants, the function which performs statistical definition of killing, etc., provide the same functionality which are required to extract results from both tools. While the individual scripts perform the whole flow of analysis required for each tool by putting together some specialized functions needed by them individually and also, importing the common functions from utilities script.

An example of a specialized function needed for DeepCrime was in the case of binary search mutation operators, where we decide whether a mutant is killed based on the range. While this was not required for DeepMutation++ operators since the parameters applied for operators do not require binary search. In most cases, the intermediate results were stored in Python *dictionaries* and the final results in Python *Pandas dataframe* structures, which then were converted into CSV files.

We use Generalized Linear Model function from the Python module *statsmodels* for calculating the p-value between quality metrics values of original and mutated models. To calculate effect size, we adopted the implementation used in DeepCrime [7], modifying it to account for correlation coefficient values of quality metrics obtained from the study of Jahangirova et al. [3].

Chapter 5

Results

In this section we report the results we have obtained by conducting experiments to answer each of the research questions. All of our experimental results are made publicly available[1].

5.1 RQ1. How does system-level misbehaviour compare to model-level mutation killing?

The figure 5.1 depicts the proportion of mutants belonging to various groups based on the number of crashes/out-of-bounds they encountered.

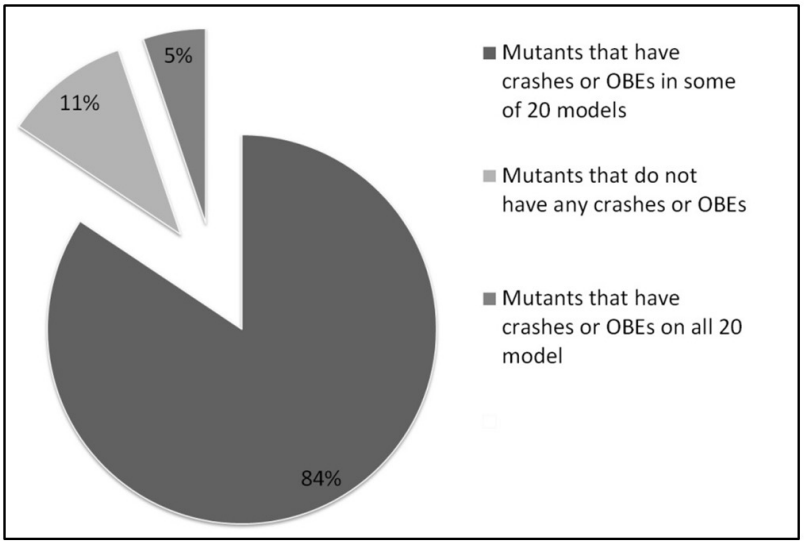

FIGURE 5.1: Percentage of mutants belonging to various groups

As the figure shows, out of 96 analyzed mutants, five (5%) had crashes/out-of-bounds on all of their 20 instances. Table 5.5 lists the mutants killed as a result of crashes/out-of-bounds on all instances of the mutated model, with the parameter value applied in column '*Parameter value*'. Figure 5.2 demonstrates the case when the car gets out-of-bounds. The scene is taken from the Udacity simulator. In contrast, 10 mutants (11%) didn't have crashes/out-of-bounds in any of the 20 instances. The remaining 81 mutants (84%) had crashes/out-of-bounds in some of 20 instances. Figure 5.4 depicts a scene in which the Udacity car drives away from the center of the lane as a response to the mutations introduced, but does not crash. The mutants who fall into the category of having some crashes/out-of-bounds are grouped together in the bar chart 5.3 by the number of crashes/out-of-bounds they have experienced. For example, the first

[1]https://github.com/nimishamanjali/thesis-detection-of-synthetic-faults-in-self-driving-cars.git

bar represents the number of mutants having crashes on one model, the second bar among the double bar indicates the number of mutants having out-of-bounds on one model. This is due to the stochastic nature of DNN models where the retraining may lead to different results. Moreover, mutation operators used in our analysis involve randomness. For example, the operator '*Remove Portion of Training Data*' from DeepCrime may delete different portions of data at each execution. Similarly, DeepMutation++ operators might alter different weights at each re-application.

FIGURE 5.2: A scenario of Udacity car driving out of lane bounds

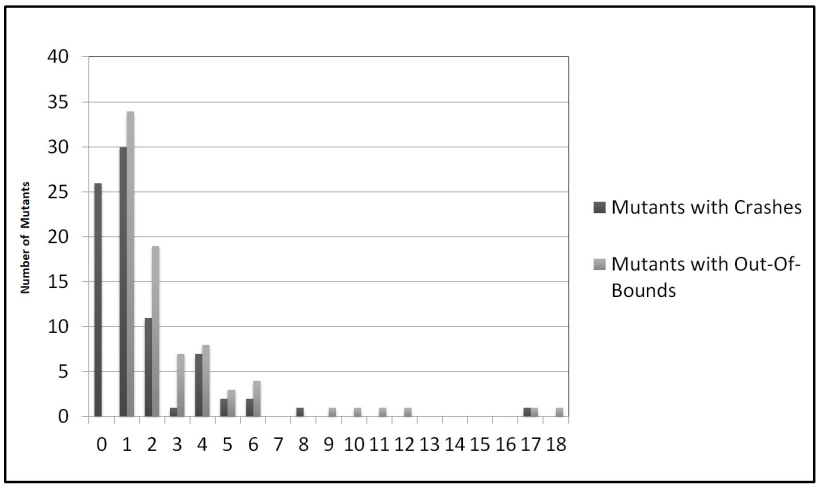

FIGURE 5.3: Chart on how many mutants had crashes or OBEs on how many instances of retrained model

FIGURE 5.4: A scenario in which an Udacity car deviates from the center of lane, but does not cause a crash

The Table 5.1 lists the binary search operators along with parameters applied in column *'# of parameters applied'*, for each operator a lower and upper bound of search in columns *'Lower Bound'* and *'Upper bound'* respectively and the configuration starting from which the mutation gets killed at the model-level in column *'Binary search configuration'*. Whereas, Table 5.2 lists the DeepCrime operators in which exhaustive search is applied, where the second column contains the number of parameters we performed and the third column with number of killed configurations at the model-level. Table 5.3 shows the DeepMutation++ operators that were killed at the model level, with the number of parameters we applied in the second column and the number of killed configurations in the third column.

Mutation operator	**# of parameters applied**	**Lower bound**	**Upper bound**	**Binary search configuration**
Change labels of training data	6	0	100	18.75
Unbalance training data	6	0	100	12.38
Make output classes overlap	6	0	100	3.12
Change learning rate	4	1E-05	1E-04	6E-05
Change number of epochs	5	1	50	44
Remove Portion of training data	6	0	99	12.38

TABLE 5.1: Model-level killed configuration for DeepCrime binary search operators

Mutation Operator	# of parameters applied	# of killed configurations
Change activation function	8	0
Remove activation function	3	0
Add weights regularisation	2	0
Change dropout rate	3	0
Change weights initialisation	14	0
Remove bias from a layer	1	0
Change loss function	11	6
Change optimisation function	5	3
Remove validation set	1	0
Gaussian fuzzing	3	0
Weight shuffling	3	1
Neuron effect block	3	2
Neuron activation inverse	3	2
Neuron Switch	3	3
Layer deactivation	3	0
Layer addition	3	0
Layer removal	3	0

TABLE 5.2: Model-level killed configuration for DeepCrime exhaustive search operators

Mutation Operator	# of parameters applied	# of killed configurations
Gaussian fuzzing	3	0
Weight shuffling	3	1
Neuron effect block	3	2
Neuron activation inverse	3	2
Neuron Switch	3	3
Layer deactivation	3	0
Layer addition	3	0
Layer removal	3	0

TABLE 5.3: Model-level killed configuration for DeepMutation++ operators

5.1.1 RQ 1.1. Do the mutants killed at the model-level always lead to a crash or an out of bounds scenario?

In the Tables 5.4 and 5.6 the cell corresponding to *'System Level(Killed)'* and *'Model level (Killed)'* we report the number of mutants that are killed by both system-level and model-level approaches. Similarly, the cell pointing to *'System Level(Not killed)'* and *'Model Level (Not killed)'* record the number of mutants that are neither killed by system-level nor by model-level approaches and, and so on.

Table 5.4 reports results for the case when we considered a mutant killed at the system-level if all 20 instances of it lead to a crash or out-of-bounds. As the table shows, 4 of the mutants that were killed at the model-level, also lead to crashes/out-of-bounds on all 20 instances. However, system-level killing based on crashes/out-of-bounds exposes one more mutant which was not revealed by model-level killing. In contrast, there are 56 mutants that are killed at the model-level but do not lead to crashes or out of bounds. There are also 35 mutants that have not been revealed by any level of mutation killing.

	Model Level (Killed)	Model Level (Not Killed)
System Level (Killed)	4	1
System Level (Not Killed)	56	35

TABLE 5.4: Comparison of model-level killing to system-level killing based on crashes and out of bounds

Mutants	**Parameter Value**
change number of epochs	50
change loss function	mean_absolute_percentage_error
change optimisation function	adadelta
Remove Portion of training data	99
unbalance train data	100

TABLE 5.5: Mutants killed by crashes or out of lane bounds

Answering to RQ 1.1, only 6.7% of mutants killed at model-level lead to models causing crashes or out-of-bounds. The percentage of such mutants across all generated ones is 5.2%. This is an expected result, as the effect of all the injected mutants may not be so dramatic. This indicates that the definition of system-level killing should not be only defined by crashes/out-of-bounds metrics. We therefore analyze the remaining mutants using the other driving quality metrics.

5.1.2 RQ 1.2. Do the mutants killed at the model-level lead to a statistically significant change in the values of driving quality metrics?

Table 5.6 reports the results for the remaining 91 mutants which do not cause crashes or out-of-bounds on all of their 20 instances. Here the mutant is killed at the system level if there is at least one driving quality metric, the values of which in the original and mutant models satisfy the condition of statistical killing.

As the table shows, 56 mutants killed by model-level approach were also killed by our definition of system-level killing. However, 35 mutants were killed only by system-level killing. Overall, at the system level no mutant goes undetected (i.e., all are killed, as shown in the second row of Table 5.6). Therefore, system-level killing based on driving quality metrics is able to detect all faults injected into deep learning system of an autonomous vehicle.

	Model Level (Killed)	Model Level (Not Killed)
System Level (Killed)	56	35
System Level (Not Killed)	0	0

TABLE 5.6: Comparison of model-level killing to system-level killing based on quality metrics

These results are in line with the results obtained in the work by Haq et al. [4] which state that model-level testing tends to be more optimistic than system-level testing, as many safety violations identified by system-level testing are not detected at model-level. This is because small prediction errors might not be visible at the model-level when using the model's performance indicator metric. However, when deployed in a simulator, these prediction errors add up and lead to big changes in the behaviour of the autonomous vehicle. Such accumulation of errors over time is only observable in system-level testing, and this also explains why there are no cases where model-level killing kills the mutant while system-level does not.

RQ1: Model-level approach detects only 61% of the total mutants injected, while system-level approach kills all mutants, demonstrating that our proposed approach is suitable for exposing misbehaviors of autonomous vehicles.

5.2 RQ2. Which driving quality metrics are the most effective in exposing system-level misbehavior?

Table 5.7 reports the overall number of killed mutants by a specific metric in the column *'#Overall'*. The number of killed DeepCrime and DeepMutation++ mutants are indicated in the columns *"# DeepCrime"* and *"# DeepMutation++"* correspondingly.

As the results show, the metric which killed the most mutants is the standard deviation of lateral speed. When it comes to each mutation testing tool, the metric which killed the most mutants is the maximum of acceleration in case of DeepCrime and standard deviation of acceleration in case of DeepMutation++. We observed that the metrics that killed the most mutants of each tool (DeepCrime vs DeepMutation++) are completely disjoint from each other. Our explanation of this finding is that the two tools produce mutants with remarkably different behaviours.

Metric	**# Overall**	**# DeepCrime**	**# DeepMutation++**
Std(LS)	86	77	9
Max(Acc)	85	80	5
Std(Acc)	84	74	10
Std(LP)	82	78	4
Max(LP)	81	76	5
Mean(SAS)	80	70	10
Max(Speed)	79	70	9
Mean(Acc)	76	71	5
Std(TPP)	72	68	4
Std(SA)	71	69	2
Mean(LP)	70	65	5
Min(LP)	68	63	5
Mean(SA)	66	58	8
Mean(TPP)	66	57	9
Min(Acc)	66	56	10
Max(SA)	64	60	4
Mean(LS)	58	51	7
Std(Speed)	51	42	9
Std(SAS)	44	34	10

TABLE 5.7: Number of mutants killed by each metric

RQ2: The metric Std(LS) is the top metric in its capability of exposing system-level misbehaviors, as it kills the most mutants. The next metrics with performances close to Std(LS) are Max(Acc), Std(Acc), Std(LP) and Max(LP).

5.3 RQ3. What is the minimal set of metrics that kills all mutants?

As the results for RQ2 demonstrated, none of the metrics is able to kill all 96 mutants. We therefore proceeded with identifying the minimum set of metrics that kill all mutants. Table 5.8 lists this set of metrics. The metric Std(LS) alone killed 86 mutants. The other two metrics down the Table 5.8 kill each 5 more mutants.

When analysing the mutants generated by each tool, STD(LP) and Max(Acc) alone were able to expose all 86 faults injected by DeepCrime tool. Whereas in the case of DeepMutation++ operators, the size of the minimal set of metrics killing all mutants is 4 and such metrics are Std(SAS), Std(Acc), Min(Acc), Mean(SAS).

Metrics
Std(LS)
Std(Acc)
Max(Acc)

TABLE 5.8: Minimal set of metrics that kills all mutants

We were also interested to see if there are any mutants which is less likely to get killed 5.9, i.e. are killed by less metrics. To achieve this, we arranged mutants in increasing order on the basis of the count of metrics that killed them. It resulted that the 1st mutant in this list is killed by six metrics. We also took a closer look at the first ten mutants of the resulted list. We noticed that the mutation operator *'Change weights initialisation'* appeared more times, followed by the operators *'Layer Removal'*, *'Add weights regularisation'*.

Mutation Operator	**Applied Parameters**	**# metrics killed**
Add weights regularisation	layer 3; new weight regularisation - l2	6
Change number of epochs	new number of epochs - 38	8
Remove bias	layer 3	9
Layer removal	mutation ratio - 0.01	10
Change weights initialisation	layer 3; new weight initialisation - he_normal	10
Make output classes overlap	Percentage - 3.12	10
Add weight regularisation	layer 1; new weight regularisation - 3	11
Layer removal	mutation ratio - 0.03	11
Change weights initialisation	layer 4; new weight initialisation - orthogonal	11
Remove activation function	layer 3	11

TABLE 5.9: Top ten most hardly killed mutants

RQ3: Out of 24 quality metrics we have used in our analysis Std(LS), Std(Acc) and Max(Acc) were enough to expose all introduced mutants. This indicates that there is a high redundancy between driving quality metrics in their capability of detecting misbehaviours of autonomous vehicles.

5.4 Threats to Validity

Internal Validity. We used GLM to check whether there was a statistical difference between values of the original and mutated models' metrics, and Cohen's d to see if the difference was negligible or significant. These could have affected the list of mutants killed retrieved from our study could influence our choice. Because, there is a possibility that other statistical tests would have given slighlty different results.

External Validity. The approach of mutation testing proposed in this thesis is applicable to all DL systems. However, our work was tested only on autonomous vehicle's environment and only on one DL model. This pose a threat in terms of generalizability of our results to other DL systems.

Furthermore, we limited our experiment to only one track ("Lake" Track) of Udacity simulator. Thus, our findings may vary across other simulators and other tracks in Udacity simulator.

Another possible risk is, the driving quality metrics used in this work is limited as the field of autonomous vehicles is rapidly expanding. We had to exclude some metrics to adapt to the simulator we used.

Reproducibility. We have made all of our experimental results publicly available[2].

[2]https://github.com/nimishamanjali/thesis-detection-of-synthetic-faults-in-self-driving-cars.git

Chapter 6

Conclusion and Future Work

In this thesis, we introduced a new approach for mutation testing of DL systems. Its applicability was demonstrated in the context of DNN based autonomous vehicles.

We devised an experimental procedure that combines existing mutation operators for DL systems, system-level testing of DL systems, mutation killing based on a statistical definition and a set of driving quality metrics which defines the behavior of an autonomous vehicle. We studied the influence of our approach called system-level killing in comparison to the existing model-level approach in the context of autonomous vehicles. To introduce faults into our DL models, we chose mutation operators provided by DeepCrime [7] and DeepMutation++ [5] tools. We carried out our experiments using the Udacity simulator. The values of driving quality metrics [3] needed for our further analysis were recorded during the simulation. As our DL model, we adopted DAVE-2 [1] which predicts the steering angle from images of the road captured by the car's camera. The experiment started with injection of mutation operators into DL models, followed by training of models and simulation. Afterwards we applied our definition of killing on the retrieved values of quality metrics. We decide mutants are killed if it has crashes or out of bounds on all instances of the mutant obtained by retraining. The remaining mutants are considered to be killed if there is a statistically significant difference between at least one driving quality metric value of original and mutated model and, if this difference is not negligible [10].

The results prove that the proposed approach is more effective compared to the existing model-level technique, as the system-level approach was able to detect 100% of mutants generated, while model-level exposed only 61% of the mutations injected. We were also able to find out thatt Std(LS), Std(Acc), Max(Acc), Max(LP) and Std(LP) are the most effective metrics in terms of mutation killing. The higher fault detection obtained with the system-level approach indicates that DL testing should focus not only on testing the DL models in isolation using some performance indicators, but also on the system-level using the metrics that quantify the model's behaviour in its environment.

6.1 Future Work

Looking forward, many improvements are possible in different aspects related to our work. The following ideas could be considered for future research:

Extending to other tracks in Udacity. Our experiment was confined to the Lake track of the Udacity simulator. In the future, the same study can be carried out on other tracks available in the simulator. Other tracks in the simulator include Jungle and Mountain tracks. Jungle track is a difficult road course with extremely tight curves. It's set in a mountain valley and has a shorter road segment than Lake Track, as well as much worse visibility due to elevation changes. The Mountain track has one challenging and narrow curve, lengthy and gentle turns, and a three-lane road section. Driving in the jungle and mountain tracks are far more difficult than driving along the lakeside track.

Extending to various DL enabled autonomous vehicles. The same analysis can be performed for various DL enabled autonomous vehicles. In the context of autonomous driving, Chauffeur and Epoch are two

other potential candidate models that can be recommended for future testing. Chauffeur utilizes a CNN to extract features from input images and an RNN to predict steering angle based on 100 previous consecutive images. Whereas, Epoch model uses a single CNN model. Using these models to carry out our experiments can serve as an additional validation of the proposed technique in this thesis.

Simulation using different simulators. Our experiment was limited to only Udacity Simulator. Several other self-driving vehicle simulators exist such as Carla and BeamNG for self-driving cars, AirSim for drones and cars. The same experiment can be carried out in these simulators.

More driving quality metrics. We evaluated 19 driving quality metrics in this study. The experiment can be extended by integrating also other metrics such as collisions with other vehicles, objects or pedestrians, Braking etc., which also add up to provide a more detailed picture of driving quality. The use of more simulators can facilitate this.

Extending to DL systems in other domains. In our study we focused only on DL systems in the field of autonomous cars. The same approach can be tested in other domains based on DL models for instance, medical applications, drones etc. This can ensure quality of such systems, focusing on their system-level behaviors that can be reflected through some domain-specific metrics.

Chapter 7

Additional Material

Mutants	Parameter values	Model-level Killing
Add weights regularisation	layer 1,2 ; type 3	Not Killed
Add weights regularisation	layer 2 ; type 3	Not Killed
Change activation function	selu ; layer 4	Not Killed
Change dropout rate	0.125 ; layer 6	Not Killed
Change epochs	47	Not Killed
Change weights initialisation	he_uniform ; layer 3	Not Killed
Change weights initialisation	lecun_normal ; layer 3	Not Killed
Change weights initialisation	ones ; layer 4	Not Killed
Make output classes overlap	3.12	Killed
Remove activation function	4	Not Killed

TABLE 7.1: DeepCrime mutants that do not have any crashes or out-of-bounds

Mutant	Ratio	#crashes	#OBEs
Gaussian Fuzzing (GF)	0.05	4	6
Layer Addition (LA)	0.01	6	6
Layer Deactivation (LD)	0.03	1	2
Neuron Activation Inverse (NAI)	0.03	1	1
Neuron Activation Inverse (NAI)	0.01	4	4
Neuron Effect Block. (NEB)	0.03	6	9
Neuron Switch (NS)	0.03	4	12
Layer Deactivation (LD)	0.05	0	1
Layer Removal (LR)	0.01	0	1
Layer Removal (LR)	0.05	0	1

TABLE 7.2: DeepMutation mutants having crashes or out-of-bounds on some models

Mutants	Parameter value	Model-level Killing
Add weights regularisation	layer 1; type 3	Not Killed
Change activation function	exponential ; layer 4	Not Killed
Change activation function	hard_sigmoid ; layer 4	Not Killed
Change activation function	relu ; layer 4	Not Killed
Change activation function	softmax ; layer 4	Not Killed
Change dropout rate	layer 1 ; new rate - 0_6	Not Killed
Change epochs	14	Killed
Change epochs	20	Killed
Change epochs	23	Killed
Change epochs	26	Killed
Change epochs	38	Killed
Change epochs	44	Killed
Change label	12.5	Not Killed
Change label	15.62	Not Killed
Change label	25	Killed
Change label	37.5	Killed
Change label	43.75	Killed
Change label	50.0	Killed
Change learning rate	1e-05	Killed
Change learning rate	4e-05	Killed
Change learning rate	6e-05	Killed
Change loss function	logcosh	Not Killed
Change loss function	mean_absolute_error	Killed
Change optimisation function	adagrad	Killed
Change weights initialisation	glorot_uniform ; layer 3	Not Killed
Change weights initialisation	identity ; layer 4	Not Killed
Change weights initialisation	lecun_uniform ; layer 4	Not Killed
Change weights initialisation	random_normal ; layer 4	Not Killed
Change weights initialisation	variance_scaling ; layer 4	Not Killed
Change weights initialisation	zeros ; layer 5	Not Killed
Delete training data	12.38	Killed
Delete training data	15.48	Killed
Delete training data	18.57	Killed
Delete training data	24.75	Killed
Delete training data	30.93	Killed
Delete training data	49.5	Killed
Delete training data	6.19	Not Killed
Delete training data	9.29	Not Killed
Make output classes overlap	12.5	Killed
Remove activation function	1	Not Killed
Remove activation function	2	Not Killed
Remove activation function	3	Not Killed
Remove Bias	3	Not Killed
Remove validation set	_	Not Killed
Unbalance training data	25	Not Killed
Unbalance training data	43.75	Not Killed
Unbalance training data	46.88	Not Killed
Unbalance training data	50	Not Killed
Change activation function	softplus ; layer 4	Not Killed
Change activation function	softsign ; layer 4	Not Killed
Change activation function	tanh ; layer 4	Not Killed
Change dropout rate	0.25 ; layer 6	Not Killed
Change dropout rate	0.75 ; layer 6	Not Killed
Change label of training data	100	Killed
Change label of training data	18.75	Killed
Change label of training data	46.88	Killed
Change learning rate	7e-05	Not Killed
Change weights initialisation	constant ; layer 4	Not Killed
Change weights initialisation	glorot_normal ; layer 3	Not Killed
Change weights initialisation	he_normal ; layer 3	Not Killed
Change weights initialisation	orthogonal ; layer 4	Not Killed
Change weights initialisation	random_uniform ; layer 4	Not Killed
Change weights initialisation	truncated_normal ; layer 4	Not Killed
Delete training data	34.02	Killed
Delete training data	37.12	Killed
Make output classes overlap	100	Killed
Make output classes overlap	25	Killed
Make output classes overlap	50	Killed
Make output classes overlap	6.25	Killed
Make output classes overlap	9.38	Killed
Unbalance training data	37.5	Not Killed

TABLE 7.3: DeepCrime mutants having crashes or out-of-bounds on some models

Bibliography

[1] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, et al. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.

[2] Y. Dai and S.-G. Lee. Perception, planning and control for self-driving system based on on-board sensors. *Advances in Mechanical Engineering*, 12(9):1687814020956494, 2020.

[3] J. Gunel, S. Andrea, and T. Paolo. Quality metrics and oracles for autonomous vehicles testing. In *2021 IEEE 14th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 2021.

[4] F. U. Haq, D. Shin, S. Nejati, and L. C. Briand. Comparing offline and online testing of deep neural networks: An autonomous car case study. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 85–95. IEEE, 2020.

[5] Q. Hu, L. Ma, X. Xie, B. Yu, Y. Liu, and J. Zhao. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1158–1161. IEEE, 2019.

[6] N. Humbatova, G. Jahangirova, G. Bavota, V. Riccio, A. Stocco, and P. Tonella. Taxonomy of real faults in deep learning systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 1110–1121, 2020.

[7] N. Humbatova, G. Jahangirova, and P. Tonella. Deepcrime: Mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2021.

[8] M. J. Islam, G. Nguyen, R. Pan, and H. Rajan. A comprehensive study on deep learning bug characteristics. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 510–520, 2019.

[9] G. Jahangirova and P. Tonella. icst2020:an empirical evaluation of mutation operators for deep learning systems. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 74–84. IEEE, 2020.

[10] K. Kelley and K. J. Preacher. On effect size. *Psychological methods*, 17(2):137, 2012.

[11] W. Kirch, editor. *Pearson's Correlation Coefficient*, pages 1090–1091. Springer Netherlands, Dordrecht, 2008.

[12] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao, et al. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, pages 100–111. IEEE, 2018.

[13] J. A. Nelder and R. W. Wedderburn. Generalized linear models. *Journal of the Royal Statistical Society: Series A (General)*, 135(3):370–384, 1972.

[14] K. Pei, Y. Cao, J. Yang, and S. Jana. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18, 2017.

[15] D. A. Pomerleau. Advances in neural information processing systems 1. In *chapter ALVINN: an autonomous land vehicle in a neural network*, pages 305–313. Morgan Kaufmann Publishers Inc., 1989.

[16] V. Riccio, G. Jahangirova, A. Stocco, N. Humbatova, M. Weiss, and P. Tonella. Testing machine learning based systems: a systematic mapping. *Empirical Software Engineering*, 25(6):5193–5254, 2020.

[17] W. Shen, J. Wan, and Z. Chen. Munn: Mutation analysis of neural networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*, pages 108–115. IEEE, 2018.

[18] A. Stocco and P. Tonella. Towards anomaly detectors that learn continuously. In *2020 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, pages 201–208. IEEE, 2020.

[19] A. Stocco, M. Weiss, M. Calzana, and P. Tonella. Misbehaviour prediction for autonomous driving systems. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, pages 359–371, 2020.

[20] Y. Tian, K. Pei, S. Jana, and B. 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.

[21] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. Deeproad: Gan-based metamorphic testing and input validation framework for autonomous driving systems. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 132–142. IEEE, 2018.

[22] Y. Zhang, Y. Chen, S.-C. Cheung, Y. Xiong, and L. Zhang. An empirical study on tensorflow program bugs. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 129–140, 2018.