CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Benchmarking Deep Learning Testing Techniques

A Methodology and Its Application

Master's thesis in Computer science and Software Engineering

HIMANSHU CHUPHAL
KRISTIYAN DIMITROV

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2020

Master's thesis 2020

Benchmarking Deep Learning Testing Techniques

A Methodology and Its Application

HIMSNHU CHUPHAL
KRISTIYAN DIMITROV

UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Software Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2020

Benchmarking Deep Learning Testing Techniques
A Methodology and Its Application
HIMANSHU CHUPHAL
KRISTIYAN DIMITROV

Supervisor: Robert Feldt, Department of Computer Science and Engineering
Examiner: Riccardo Scandariato, Department of Computer Science and Engineering

Master's Thesis 2020
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2020

Benchmarking Deep Learning Testing Techniques
A Methodology and Its Application
HIMANSHU CHUPHAL, KRISTIYAN DIMITROV
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

With the adoption of Deep Learning (DL) systems within the security and safety-critical domains, a variety of traditional testing techniques, novel techniques, and new ideas are increasingly being adopted and implemented within DL testing tools. However, there is currently no benchmark method that can help practitioners to compare the performance of the different DL testing tools. The primary objective of this study is to attempt to construct a benchmarking method to help practitioners in their selection of a DL testing tool. In this paper, we perform an exploratory study on fifteen DL testing tools to construct a benchmarking method and have made one of the first steps towards designing a benchmarking method for DL testing tools. We propose a set of seven tasks using a requirement-scenario-task model, to benchmark DL testing tools. We evaluated four DL testing tools using our benchmarking tool. The results show that the current focus within the field of DL testing is on improving the robustness of the DL systems, however, common performance metrics to evaluate DL testing tools are difficult to establish. Our study suggests that even though there is an increase in DL testing research papers, the field is still in an early phase; it is not sufficiently developed to run a full benchmarking suite. However, the benchmarking tasks defined in the benchmarking method can be helpful to the DL practitioners in selecting a DL testing tool. For future research, we recommend a collaborative effort between the DL testing tool researchers to extend the benchmarking method.

Keywords: Deep Learning(DL), DL testing tools, testing, software engineering, design, benchmark, model, datasets, tasks, tools.

Acknowledgements

We would like to thank our supervisor Prof. Robert Feldt of the Software Engineering Division at Chalmers | University of Gothenburg for his continuous input, assistance and counseling throughout the thesis work. We would also like to thank Prof. Riccardo Scandariato of the Software Engineering Department at Chalmers University of Technology as the examiner of this thesis for all the valuable feedback. Finally, we would like to extend our appreciation to our family and friends for their moral support.

Himanshu Chuphal, Kristiyan Dimitrov, Gothenburg, June 2020

Contents

List of Figures

List of Tables

1
Introduction

1.1 Background

Over the past few years, Deep Learning (DL) systems have made rapid progress, achieving tremendous performance for a diverse set of tasks, which have led to widespread adoption and deployment of Deep Learning in security and safety-critical domain systems [2][9]. Some of the most popular examples include self-driving cars, malware detection, and aircraft collision avoidance systems [3]. Due to their nature, safety-critical systems undergo rigorous testing to assure correct and expected software behavior. Therefore testing DL systems becomes crucial, which has traditionally only relied on manual labeling/checking of data [9]. There are multiple DL testing techniques to validate different parts of a DL system's logic and discover the different types of erroneous behaviors. An example of some of the more popular DL testing techniques which validate deep neural networks using different approaches for fault detection are DeepXplore [2], DeepTest [16], DLFuzz [3], Surprise Adequacy for Deep Learning Systems (SADL) [9]. While ensuring predictability and correctness of the DL systems to a certain extent, there is no standard method to evaluate which testing technique is better in terms of certain testing properties, i.e. correctness, robustness, etc. There is a guide for the selection of appropriate hardware platforms and DL software tools [29] but no guide available as such to select appropriate DL testing techniques. Additionally, there is an increasing trend in the research work done within the field of DL testing. Therefore, there is a need for a method to benchmark DL testing tools with in-depth analysis, which will serve as a guide to practitioners/testers and the DL systems community.

1.2 Statement of the Problem

DL systems are rapidly being adopted in safety and security-critical domains, urgently calling for ways to test their correctness and robustness. Consequently, there is an increase in the cumulative number of publications on the topic of testing machine learning systems between 2018 and June 2019, 85% of papers have appeared since 2016, testifying to the emergence of new software testing domain of interest: machine learning testing [30]. Recently, a number of DL testing tools have been proposed such as DeepStellar [13], DeepFault [18], DeepRoad [17], and DLFuzz [3]. However, there is no guide available as such to select an appropriate DL testing techniques based on testing properties (e.g., correctness, robustness, and fairness), testing components e.g., the data, learning program, and framework), testing

workflow (e.g., test generation and test evaluation), and application scenarios (e.g., autonomous driving, machine translation) [30]. As stated by Xie et al. [24], due to a lack of comparative studies on the effectiveness of different testing criteria and testing strategies, many challenges and questions are left open and unresolved, such as whether the existing proposed testing criteria are indeed useful. Moreover, selecting a testing tool for real-world scenarios is a big investment. Since, there is no standard method or guidelines available to evaluate DL testing tools, the selection of an appropriate DL testing tool or technique for a relevant DL use case remains a challenge for practitioners.

1.3 Purpose of the Study

The purpose of the study is to design a method for benchmarking DL testing tools. The benchmark method shall be constructed following the general guidelines for benchmarks in software engineering proposed by Sim et al. [31]. The selected testing techniques will be evaluated using a sufficiently complex task sample as to bring out their effectiveness. The method would provide academia with a standard that can be used to verify the effectiveness of both existing and newly aspiring DL testing techniques. Afterward, we hope to receive feedback from at least one industry source on the usability of the method within an industrial setting and use that feedback to extend the method. In addition, the results of the study can be used in itself by industrial experts to support the selection of an appropriate DL testing technique.

1.4 Hypotheses and Research Questions

Our benchmarking evaluation is designed to answer the following 3 research questions (RQ1-3):

RQ1: What existing state-of-the-art and real-world applicable DL testing tools are available?
We investigate the correctness, fairness, efficiency, and robustness tested properties by the existing DL testing tools and decide which testing tools are most suited for a relevant case.

RQ2: To what extent do different testing techniques and workflows of the DL testing tools perform better towards DL benchmarking tasks?
Our goal is to design a benchmark methodology DL to evaluate DL testing tools. We have defined two sub-hypotheses to help answer the research question.
Hypothesis 1: There are significant qualitative differences in the selected DL testing tools in terms of the properties tested by the tools.
Hypothesis 2: There is a significant difference in test input generation of selected DL testing tools for a given type of dataset.

RQ3: How can the proposed benchmarking methodology of DL testing tools help

practitioners as a reference for selecting an appropriate testing technique?
We aim to assess the benchmarking methodology using different DL testing tools for an industry-grade dataset. The benchmarking methodology will be presented to two industry contacts to receive feedback and assess the feasibility of its application in real-world DL scenarios.

1.5 Report Structure

The rest of this thesis is structured as follows: Chapter two describes the related work and gives background information. Chapter three explains the research methodology we followed to design the DL testing tools benchmarking and also presents the DL testing requirements, testing scenarios, and benchmarking tasks. Chapter four summarizes the results obtained from the pre-study and the benchmarking research method of the DL testing tools and techniques. Chapter five summarizes the results of the benchmarking method. Thereafter, Chapters six and seven address validation and discussion of the entire benchmarking process of the thesis. Chapter eight explains threats to validity. Finally, Chapter seven concludes the thesis and mentions future work and possible research opportunities.

2
Related Work and Background

This section starts with an introduction about testing in general, followed by DL testing, explaining emerging research in the field of DL testing tools and techniques. Furthermore, DL testing workflow, components, and testing properties are presented. Thereafter, the key differences between traditional software testing and DL testing, as well as challenges in DL testing are introduced. Finally, DL testing tools research trends and benchmarking theory are presented.

2.1 Testing

Testing is the process of executing a program with the intent of finding errors. Software testing is a technical task, yes, but it also involves some important considerations of economics and human psychology [1]. In an ideal world, we would want to test every possible permutation of a program. In most cases, however, this simply is not possible. Even a seemingly simple program can have hundreds or thousands of possible input and output combinations. Creating test cases for all of these possibilities is impractical. Complete testing of a complex application would take too long and require too many human resources to be economically feasible. A good test case is one that has a high probability of detecting an undiscovered error. Successful testing includes carefully defining expected output as well as input and includes carefully studying test results.

In general, it is impractical, often impossible, to find all the errors in a program. Two of the most prevalent strategies include black-box testing and white-box testing.

Black-Box Testing
One important testing strategy is black-box testing (also known as data-driven or input/output-driven testing). The goal here is to be completely unconcerned about the internal behavior and structure of the program. Instead, concentrate on finding circumstances in which the program does not behave according to its specifications. To use this method, view the program as a black box. In this approach, test data are derived solely from the specifications (i.e., without taking advantage of the knowledge of the internal structure of the program) [1].

White-Box Testing
Another testing strategy, white-box (or logic-driven) testing, permits you to examine the internal structure of the program. This strategy derives test data from an

examination of the program's logic (and often, unfortunately, at the neglect of the specification). The goal at this point is to establish for this strategy the analog to exhaustive input testing in the black-box approach. Causing every statement in the program to execute at least once might appear to be the answer, but it is not difficult to show that this is highly inadequate [1].

Traditional software testing is done through the use of test cases against which the software under test is examined [5]. Typically, a test case will either be error-revealing or successful. The successful test-cases are usually less important than the error-revealing test cases because they often provide little information regarding the state of the system. Even if all tests are successful, it does not guarantee that there are no bugs in the system. As the complexity of the system rises, the harder it can get to detect the bugs. However, the error-revealing test cases help improve the system by detecting the bugs. This implies that the test set needs to be 'adequate' in identifying program errors to ensure program correctness [8]. For that purpose, the concept of the Test Adequacy Criteria has been introduced. Test adequacy criteria are 'rules' and conditions that the test set needs to comply with to improve the quality of testing. For the purpose of fine-tuning the test cases different approaches have been proposed like differential testing, metamorphic testing, etc.

2.2 DL Testing

2.2.1 Definition

Definition: DL testing is the process of executing a DL program with the intent of finding errors in a DL system.

A DL system is any software system that includes at least one Deep Neural Network (DNN) component. A DNN consists of multiple layers, each containing multiple neurons. A neuron is an individual computing unit inside a DNN that applies an activation function on its inputs and passes the result to other connected neurons. Overall, a DNN can be defined mathematically as a multi-input, multi-output parametric function composed of many parametric sub-functions representing different neurons. Automated and systematic testing of large-scale DL systems with millions of neurons and thousands of parameters for all possible inputs (including all the corner cases) is very challenging.

Datasets play an important role in any DL system, which is basically a set of instances for building or evaluating a DL model. At the top level, the data could be categorized as Training data (the data used to train the algorithm to perform its task), Validation data (the data used to tune the hyper-parameters of a learning algorithm), and Test data (the data used to validate DL model behavior).

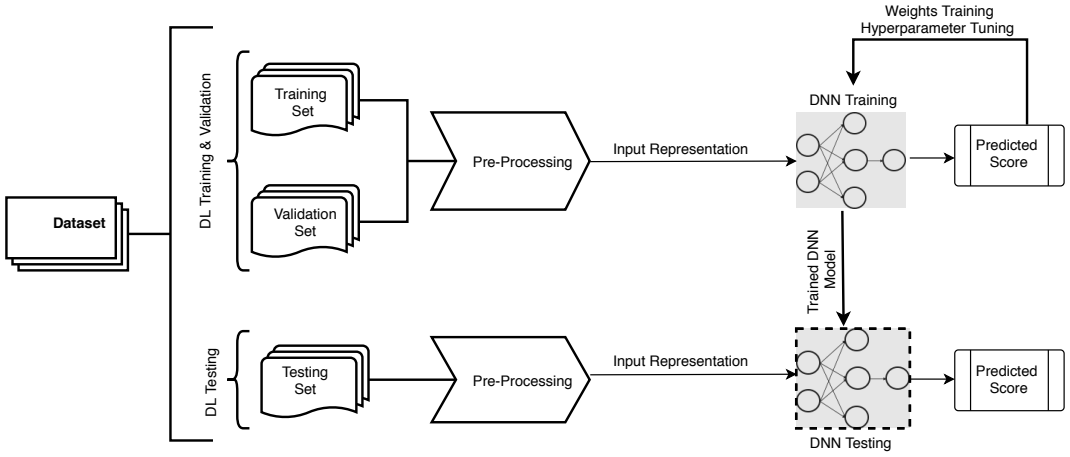

Figure 2.1: DL System Phases

The proposed system uses a typical machine learning protocol comprised of two phases as shown in Figure 2.1. The two phases are (i) a training and validation phase used to train the model and tune the hyper-parameters, and (ii) a testing phase in which the model is evaluated. The data is first divided into three disjoint sets for training, validation, and testing. In the training and validation phase, the datasets are first passed through a pre-processing stage converting the raw data into their corresponding input representations. These input representations are then fed into a DNN model that tries to predict the assessment ratings. The data in the training set is used to train the model parameters. The model performance on the validation set is used to tune the hyper-parameters. The testing set data is used to evaluate the model performance on unseen data following the same steps as above.

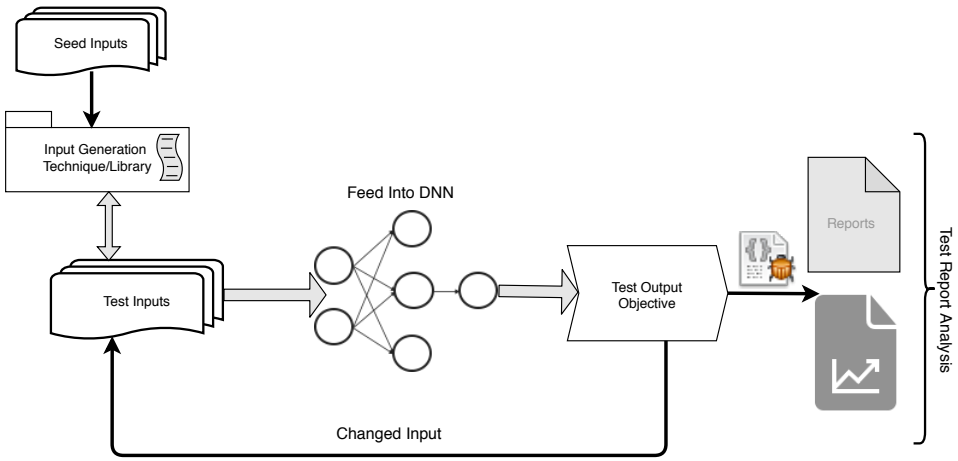

Figure 2.2: Overview of DL Testing Tools Process

Figure 2.2 depicts the overview DL testing tools process in general. DL testing tools and techniques vary in how each activity is executed, the process generally involves the generation of new testing input whether by image-transformation [16], mutation [3][14], gradient descent[2], etc. of the original input or generating entirely new synthetic input based on the original [17]. The test cases are made out of that input and

are ran through the DL system to fulfill a testing objective, i.e. discover robustness related behavioral errors, increase neuron coverage, etc. Finally, either input are retained for a new iteration of the test case seed or a bug report is generated as to show the DL system's performance.

Existing techniques in software testing may not be easily applied to DL systems since they are different from conventional software in many aspects. Until now, many researchers have devoted their efforts to enhance the testing of DL models. Some of the differences are listed in Section 2.3.

2.2.2 DL Testing Workflow

The DL testing workflow is about how to conduct DL testing with different testing activities. In this section, the five key activities in DL testing and the approaches that are involved with those activities are introduced. Figure 2.3 shows different DL testing workflows.

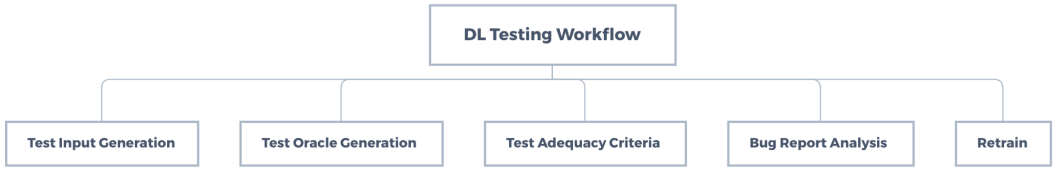

Figure 2.3: DL Testing Workflow

The first activity of testing a DL system as per the deduction of various testing technique papers is the Test Input Generation. This activity usually consists of modifying the existing testing input set [3][16] or generating new testing input [17]. Under normal circumstances, the testing set of data is a portion of the training data that is taken out as to be able to test the system for abnormal behavior. However, for the system to satisfy certain properties like robustness, the raw testing set is usually not enough to ensure testing data quality. For that purpose various techniques use input generation as to improve that quality [3][2][13][16][18]. The implications and reasoning behind test input generation are further covered in Section 2.4.
The second activity of importance would be to establish an oracle. After all, a test generally needs to satisfy a condition to be able to pass, without having such a condition it cannot be established if a test found a fault. However, DL systems widely suffer from the oracle problem and therefore require other means of establishing an oracle. This is further depicted in Section 2.4 due to it being a major constraint when testing a DL system.
The third activity to consider is the definition of test adequacy criteria. As mentioned in Section 2.1, test cases need to conform to a set of rules to ensure that the tests are qualified to find errors. These rules come in the form of Test Adequacy Criteria and while their purpose is self-explanatory, due to the large gap in structural logic between DNNs and traditional software, new adequacy criteria need to established. SADL [9] proposed a set of novel test criteria under the argument that

the test input needs to contain inputs that are 'surprising' to the system.
The fourth activity of the workflow is the Debug Analysis Report. Once an input that induced erroneous behavior within a DL system has been found, it needs to be retained because that input becomes interesting. In this particular case, interesting is used to depict the possible wide usability of the error inducing input. DeepStellar [13] uses the error inducing input as a seed for its repeated test generation to improve overall coverage. Whereas other techniques like SADL [9] retain the inputs as valuable for retraining, which brings us to the final activity.
The final activity of the workflow is retraining. This activity is rather simple, yet incurs significant complications. Once error-inducing inputs are retained they can be used for retraining and improving the model. This has been by far the most standard usage of inputs. The implication that this process holds is the fact that it involves manual labeling. The retained inputs need to be manually labeled in most cases to improve the model and eliminate the detected behavioral errors. Making this process automated would greatly improve the testing process as it would automatically fix behavioral errors per test iteration. Currently, most techniques are unaware of run-time what inputs are entering, greatly due to the automated input generation that is meant to cover as much of the testing space or as effectively as possible.

2.2.3 DL Testing Components

The DL testing components are components in a DL system towards which testing is aimed at. In this section, the three key components in DL testing are introduced. Figure 2.4 shows different key DL testing components.

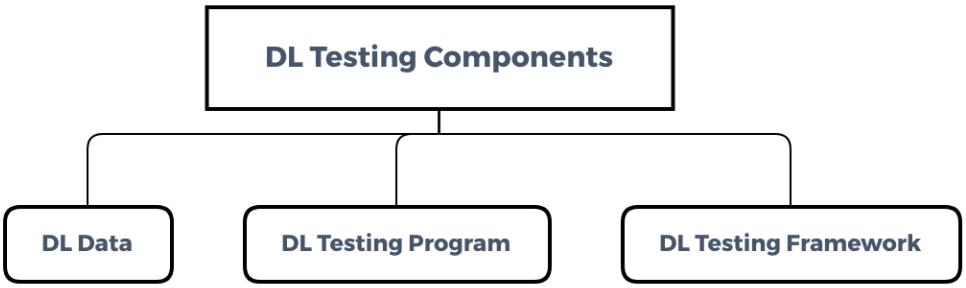

Figure 2.4: DL Testing Components

DL testing components comprise of testings components, i.e. DL data, DL testing learning program/script, and DL testing framework, for which a DL testing tool might find a bug. The procedure of a DL model development requires interaction with several components such as data, learning program, and learning framework, while each component may contain bugs [30]. Therefore, when conducting DL testing, developers may need to try to find bugs in every component including the data, the learning program, and the framework. In particular, error propagation is a more serious problem in DL development because the components are more closely

bonded with each other, which indicates the importance of testing each of the DL components.

DL Data
DL datasets are used for building or evaluating a DL model. Datasets are categorized as Training data, Validation data, and Test data. The Test data is used to validate DL model behaviour.

DL Testing Program
A DL testing program is the script/program written by a DL software engineer to build and validate the DL system. As shown in Figure 2.1 and Figure 2.5, a learning program is required to first build and validate a DNN model, which is once trained can be used to test against testing input seeds. The program or the script can be programmed in any high-level programming language such as Python [42], etc.

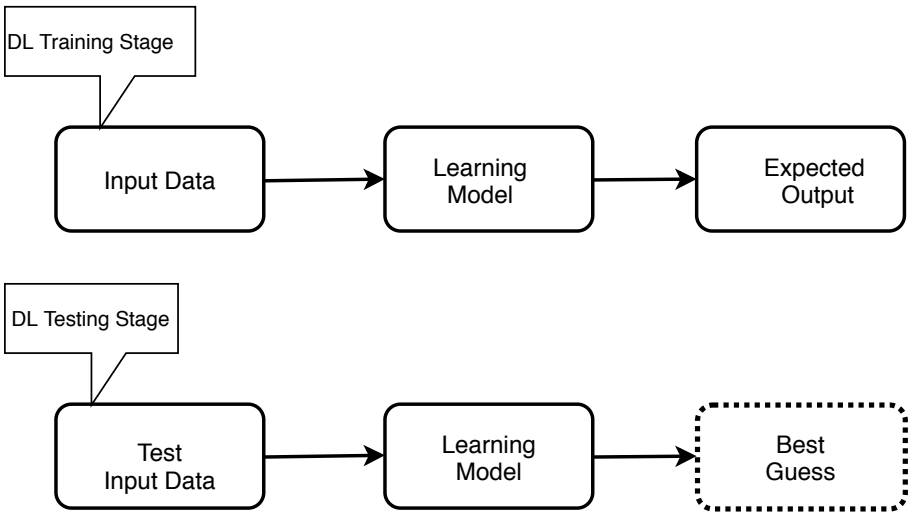

Figure 2.5: DL System Stages

DL Testing framework
DL Testing framework is the library, or platform being used when building a DL model for testing, for example TensorFlow [39], Keras [38], and Caffe [40], Scikit-learn [41], etc. Keras [38] is a Python framework for building DL systems. It is a convenient library to construct any DL algorithm. The advantage of Keras is that it uses the same Python code to run on CPU or GPU and allows training of state-of-the-art algorithms for computer vision, text recognition, etc. Keras is used in organizations like CERN, Yelp, Square or Google, Netflix, and Uber.

2.2.4 DL Testing Properties

Testing properties refer to what quality characteristics to test in a DL system: What conditions DL testing needs to guarantee for a trained DNN model. This section

lists typical properties that the literature has considered. The properties are classified into basic functional requirements (i.e., correctness and model relevance) and non-functional requirements (i.e. robustness, efficiency, generality, and reliability). These properties are not strictly independent of each other when considering the root causes, yet they are different external manifestations of the behaviors of a DL system and deserve being treated independently when testing a DL system. Figure 2.6 shows different functional and non-functional properties in DL testing.

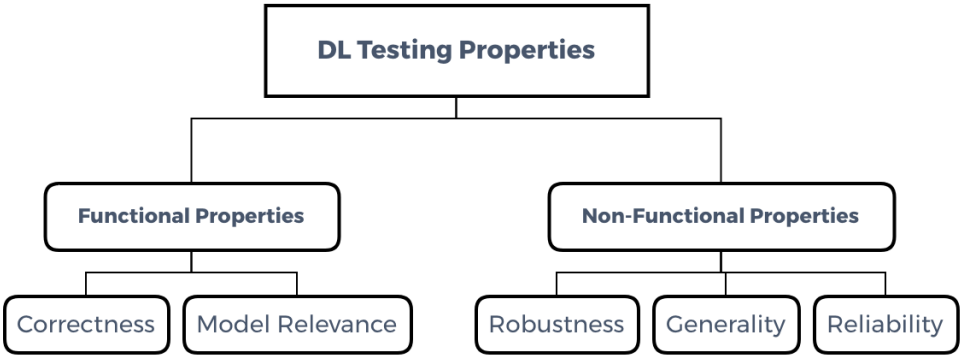

Figure 2.6: DL Testing Properties

Non-functional Properties:

- **Robustness:**
  The robustness of a DL system, in broader terms, is related to the system's capability of withstanding corner-case scenario input. A DL system is essentially a trained DNN model which when given an input, is meant to recognize that input correctly. But the problem lies in the variations and conditions of that input. Taking image classification as an example, when looking at a 'car', the image we see through our eyes can have a wide variation depending on the weather condition or other outside influences. A human can recognize a car even if it is raining and the vision is slightly obscured, but the same cannot be always said about a DL system which may give an output different than a car if such image obstructions are in place. For that purpose, the DL system needs to be trained as to be able to recognize a 'car' regardless of other such outside influences. Intuitively, this implies that better quality training data would lead to fewer robustness issues. An example of a robustness measurement is DeepXplore [2], which both utilizes neuron coverage to measure the parts of the DL system exercised by test inputs and multiple systems with similar functionality to discover robustness faults within the DL system.

- **Generality:**
  Generality as a DL system property is connected to the neurons in each layer and the way they are trained. More general neurons are applicable for general tasks but fail on tasks that require specifics. Therefore, the first layers tend of a DNN tend to be more general whereas the later layers go into specifics for which the system is meant to handle. Generality has been brought up in

testing tool research [14], however, its mention is brief and is used to point out the apparent problem of high-quality training and testing data evaluation. Generality would require a large amount, and of high quality, training data from which testing data can be selected out. Without a way of evaluating the quality of the training and testing data, the generality of a system cannot be ensured, hence for the need for techniques that serve to evaluate the training data like DeepMutation [14].

- **Reliability:**
  Reliability within DL refers to how error-resilient a DL system is. Similar to *generality*, *reliability* has been brought up with the DL testing technique research [3] [2] but it was not further built-upon as the main focus was *robustness*. This is due to the reliability of a DL system depending greatly on the robustness of that system and the measurements are, therefore, focused on measuring robustness.

Functional Properties:

- **Correctness:**
  The first functional property of importance is system *correctness*. Correctness is essentially the prediction accuracy of the system under test. Similar to *robustness*, DL system *correctness* is heavily reliant on the quality of the training data [12]. The more and better data the system is trained with, the better the system is expected to perform in terms of giving correct predictions. However, the reality is that *robustness* is the property that heavily influences the system *correctness*. The better the system is able to handle the wide spectrum of corner-cases around an input, the higher the chance for the system to give correct predictions. That can further be seen by how several papers on the topic of DL testing recognize both correctness and robustness as important, but highlight or focus on robustness related issues [9][11][3] as to improve correctness. An example of a widely adopted correctness measurement is AUC (Area Under Curve) which measures how well the model performed and is used by SADL [9].

- **Model Relevance:**
  Zhang et al.[30] defines *model relevance* as mismatches between model and data. It is evaluated by using techniques whose objective is to find out whether the model is overfitted or underfitted by injecting 'noise' to the training data. Overfitted models tend to fit the noise in the training sample, whereas the underfitted models will have a very low training accuracy decrease. An alternative view can be seen on *model relevance* in DL as to whether the model architecture is also fit for the task. DNN selection for a DL system has been discovered to be a potential issue [10] due to different types of models being good at different types of tasks, i.e. recurrent neural networks (RNNs) perform better at sequential input streams[27]. However, we were unable to uncover techniques that measure whether the model architecture is appropriate for the type of task that the DL system is expected to handle.

2.3 Software Testing vs. DL Testing

As defined by Guo et. al. [3], currently there exists a large gap between traditional software and DL software due to the "totally distinct internal structure of deep neural networks (DNNs) and software programs". However, despite these differences, the testing community has made progress in applying traditional testing techniques to DL systems. Table 2.1 shows the key differences between traditional software testing and DL testing.

In traditional software development, developers specify the clear logic of the system, whereas a DNN learns the logic from training data. Software testing, in theory, is a fairly straightforward activity. For every input, there should be a defined and known output. We enter values, make selections, or navigate an application and compare the actual result with the expected one. If they match, we nod and move on. If they don't, we possibly have a bug. Figure 2.7 shows the key comparison between the two systems.

Table 2.1: Key Differences : Software Testing vs DL Testing

Software Testing	DL Testing
Fixed scope under test	Scope changes overtime
Test oracle is defined by software developers	Test Oracle is defined by DL developers and also communities with labeling data
Test adequacy criteria is usually code coverage	Test adequacy is not concretely known
Testers are usually software developers	Testers include DL designers, developers and data scientists
False positives in bugs are rare	False positives if errors are frequent
Component to test include code or the software application	Component to test include both data and code

There are several prominent testing techniques that are successfully adapted from traditional software testing, i.e. differential, mutation, metamorphic, combinatorial, and fuzz testing. It is interesting to note that although some papers focus on highlighting an individual technique, it usually uses more than one technique to fulfill its purpose.

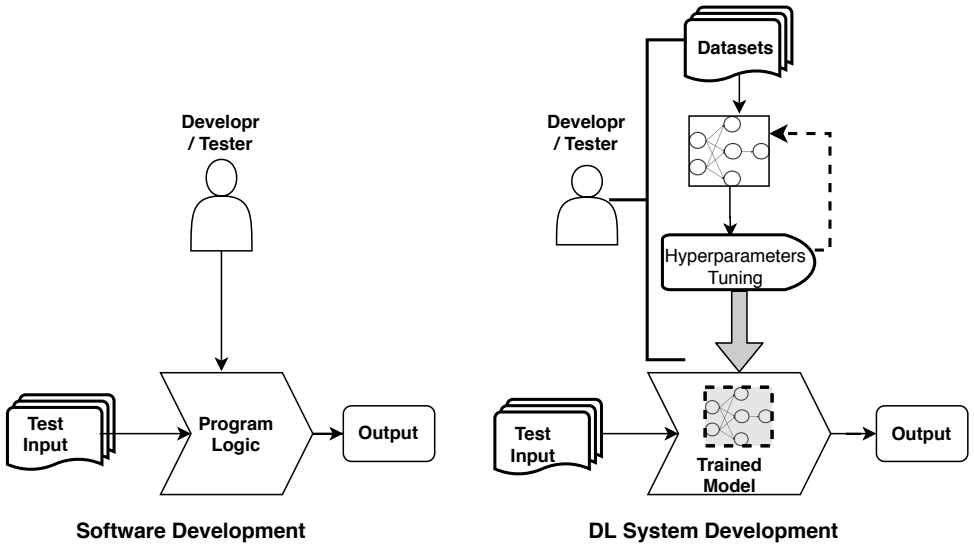

Figure 2.7: Comparison : Traditional Software vs DL System Development

Differential Testing
Differential testing, a form of random testing, is done by giving one or more similar systems the exact same input and use the output as a cross-referencing oracle to identify semantic or logic bugs. The principle behind it, is feeding both systems mechanically generated test cases and if one of the systems shows a difference in behavior or output then there's a candidate for a bug-exposing test case [4]. However if both systems give the same output, even if a bug is present, the bug cannot be detected. Pei et al. [2] successfully adapted the differential testing approach to DL systems through DeepXplore. The problem DeepXplore aimed at resolving with this technique was error detection without manual labeling. Manual labeling on its own is a problem due to being costly, time-consuming, and has a limited coverage due to the aforementioned reasons. To resolve that DeepXplore applies differential testing by using multiple similar DL systems. However, the approach does incur difficulties such as if the systems are too similar, the algorithm may not be able to find the difference inducing inputs which essentially are what causes erroneous behavior, particularly related to the robustness of the system. A different approach to differential testing was proposed by Guo et al. [3] to avoid the implications faced in DeepXplore's case by using one model in the framework DLFuzz. The 'differential' part comes through the use of mutation testing to mutate a set of inputs. The mutated inputs and the original inputs are then run through the DL system and if a difference in output is observed between the two inputs then there is an error. With this, we move on to the next testing technique that is widely utilized in DL testing.

Mutation Testing
Mutation testing is a fault-based testing technique that uses the metric "mutation adequacy score" to measure the effectiveness of a test set [7]. The way this is done is through the use of mutants, deliberately seeded faults that are injected in the original program. The objective here is to affirm the quality of the test set, hence if the test cases fail to detect the mutants then the quality of the tests is under question. The mutation score gives a tangible estimate by calculating the ratio of the detected faults over the total number of seeded faults. In DL testing this technique is utilized in various ways. DLFuzz[3] uses mutation as to mutate inputs and use the output of those mutants in cross-comparison to the original input's output. DeepMutation [14] on the other hand injects mutation faults not only in the input but the training program as well. Afterward further mutation operators are designed and injected directly in the model of the DL system. Other techniques like DeepStellar [13] use mutants for two-fold purposes, if the mutant generated leads to incorrect output then it's an adversarial sample, otherwise, if it improves neuron coverage it is retained and added back as a seed to the test case queue.

Metamorphic Testing
Metamorphic testing at its core tries to solve *the oracle problem* [6] which is built on the assumption that a test oracle is readily available but in practice that may not be the case [5]. The oracle problem is particularly present in applications that are meant to provide the answer to a problem like shortest path algorithms in non-trivial graphs and Deep Learning systems which are to give a prediction using non classi-

fied or immensely large sets of data. The way that metamorphic testing tackles the problem is by evolving the successful test cases used on the system. By firmly believing that there are errors in the system, metamorphic testing seeks to improve the test cases by branching out of the reasoning behind the test case and testing around that reasoning, called a metamorphic relation, i.e. if the test was meant to check the occurrence of a k^{th} element in an unsorted array and the program returned an element from that position, formulate case variations of what errors could possibly occur whilst returning an element from the array [5]. This does not explicitly try to detect all errors in the system or prove that there was an error in the system but it increases the confidence in the system behavior is correct. An example of how metamorphic testing and its reasoning approach is used in DL systems is DeepRoad [17]. Because DeepRoad uses image synthesis to generate input, it suffers from a similar problem as the one mentioned in Section 2.2.2 in regards to retraining. The tool does not know on run-time what kind of input goes in as to be able to tell whether the output that came out is correct. For that purpose, they use the reasoning that regardless of the input (driving scenes) that goes in it should correspond to the driving behavior of the original driving scenes which were used to synthesize the new inputs.

Combinatorial Testing
Combinatorial testing is a testing technique aimed at variable interactions. Large systems often consist of many variables that interact with each other and each of those interactions between two or more variables can lead to failures. Such failures are called interaction failures [19]. Testing all interactions between variables within a large system is not feasible, but research led to the belief that not all interactions need to be tested. In fact, interaction failures happen mainly on configuration variables and input variables. Additionally, the bigger the number of interactions is, the smaller the chance of a failure, i.e. 3-way interaction has a lesser chance than a 2-way interaction, 4-way has an even lesser chance, etc. Combinatorial testing is built on those principles and is therefore cost-efficient and effective. However, overconfidence in this may lead to missed interactions that could possibly induce failures. Therefore, this approach requires experience and good judgment to be effective [20].

Within recent years, an effort has been made to adopt combinatorial testing to DL systems [21]. If the vast run-time space of a DL system, where each neuron is a run time state, is compared to the problem combinatorial testing is trying to resolve, the vast interaction space between variables, the similarities from an abstract perspective can be observed. The testing framework implemented for this purpose, DeepCT [21], adapts combinatorial testing by representing the space of the output values into intervals such that each interval is covered. In the spirit of combinatorial testing, these intervals can be viewed as variables whose interaction can be tested. However, while this way the combinations of intervals are finite, they can still increase exponentially with the number of neurons. Therefore, sampling of neuron interactions is conducted to reduce the number of test inputs that have to be executed.

Fuzz Testing
Fuzz testing is a form of testing where random mutations are applied to the input and the resulting values are checked for whether they are interesting [22]. Due to its success in detecting bugs, many techniques have been developed that made different 'fuzzers' classify as blackbox, whitebox, or gray-box [23]. All of which is due to the fuzzing strategy as there are underline rules that the fuzzers follow. Although only two of the tool/framework related papers that we found focus on showcasing fuzzing [3][23] as a DL testing approach, there are also papers that do not explicitly focus on fuzzing techniques, but fuzzing is interwoven within frameworks [14][27].

2.4 Challenges in DL Testing

DL testing has experienced recent rapid growth. Nevertheless, DL testing remains at an early stage in its development, with many challenges and open questions lying ahead. The key challenges in automated systematic testing of large-scale DL systems are twofold: (1) how to generate inputs that trigger different parts of a DL system's logic and uncover different types of erroneous behaviors?, and (2) how to identify erroneous behaviors of a DL system without manual labeling/checking?

Early Stage of Research
Arguably, the biggest challenge to DL testing is currently the fact that the field is in an early stage. Xie et al. [24] noted that due to the research being at an early stage, there is a lack of comparative studies on the effectiveness of DL testing criteria and strategies. This results in doubt on whether the currently proposed criteria and strategies are indeed useful and can be built upon or whether the application of traditional strategies is still effective.

Test Input Generation
The challenge with testing input is that the *robustness* of the DL system greatly relies on quality testing data. The better the testing data the higher the confidence in the DL system. However, for a DL system to be applicable within safe-critical fields, mistakes cannot be allowed. For that purpose, multiple testing techniques were proposed to improve testing data quality. SADL [9] proposes a test adequacy criterion that test input should be "sufficient but not overly surprising to the testing data". According to this approach to improve the testing of the DL system, the testing data should contain input that is different from the one used for training but not too different. Another approach for test input generation is the synthetic test case generation implemented by DeepTest [16]. DeepTest, a systematic testing tool for DNN-driven vehicles, uses image transformation to apply realistic changes that a car would face to the input, i.e. presence of fog, rain, change in contrast, etc. to generate its synthetic test cases. DeepRoad [17], although being focused on the same area as DeepTest, does not use image transformation, but Generative Adversarial Networks (GANs) to generate input mimicking real-world weather conditions. DeepXplore [2] uses adversarial sample generation and DeepFault [18] uses a suspiciousness-guided algorithm to generate its synthetic inputs. Various tech-

niques are used to generate input to be able to improve training data quality. After all, test inputs that trigger a faulty behavior within a DL system can be added to the training data to resolve a logical bug, or depending on the technique used the input can be retained for other uses, i.e. for a future test case seed [3].

Test Oracle Problem
One of the greater challenges of DL testing is the test oracle problem [6]. DL systems are 'predictive' systems, systems that are meant to provide us with an answer. This implicates automated testing because erroneous behavior cannot be identified without manual labeling or checking [2]. Initial attempts at resolving the oracle problem were done through the use of differential testing[4] that resulted in the automated whitebox testing framework DeepXplore [2]. However, unlike traditional software, acquiring a similar DL system to the one under test is significantly more difficult. After all, DL systems are decision systems that are made by training a model with a, often, a large set of data that determines the weighs between neurons. A later attempt at differential testing was done by Guo et al. [3] which eliminated the need of having at least two similar systems but is based on the assumption that a DL system could potentially fail if the input contains slight perturbations that are indistinguishable to the human eye.

Test Assessment Criteria
For a system that is as complex as a DL system, it is very challenging to have test assessment criteria pre-defined. Deep learning rises from the collaborative functioning of layers and does not belong to any single property of the system and so ultimately, we can never be sure your model produced has the exact properties we'd like. To this end, actually testing the quality of a model requires training, which would traditionally be considered our second tier of testing as integration. In addition, this form of training is computationally expensive and time-consuming.

Complex DNN Model
Calculating tensor multiplications are difficult and rarely can be done by software engineers as "back of the envelope calculations". The maths is complicated for such complex models. Even with fixed seed initialization, the regular Xavier weight initialization uses 32-bit floats, matrix multiplication involves a large series of calculations, and testing modules with batching involves hard-coding tensors with 3 dimensions. This complexity is further reflected in the variations of DNN models, like RNNs (Recurrent Neural Networks) and CNNs (Convolutional Neural Networks), each of which is better suited for specific tasks. This leads to some techniques being less universal due to having to focus on a specific model. Whilst none of these tasks are insurmountable, they massively stifle development time and creating useful unit tests.

DL Testing Components Failures
Even trained DNN Models to fail silently, whilst testing a DNN model behaves correctly with specific input, the inputs of a neural network are rarely a finite set of inputs (with the exception of some limited discrete models). Networks work in

larger orchestration and regularly change their inputs, outputs, and gradients.

2.5 DL Testing Tools

2.5.1 Timeline

Figure 2.8 shows trends in **"Deep Learning System Testing"**, showing several key contributions in the development of DL testing tools and techniques. In 2017, K. Pei et al. [2] published the first white-box testing paper on DL systems. Following this paper, a number of DL testing techniques and tools have emerged, such as DeepTest [16], SADL [9], DeepGauge [11], DeepConcolic [15], DeepRoad [17], etc.

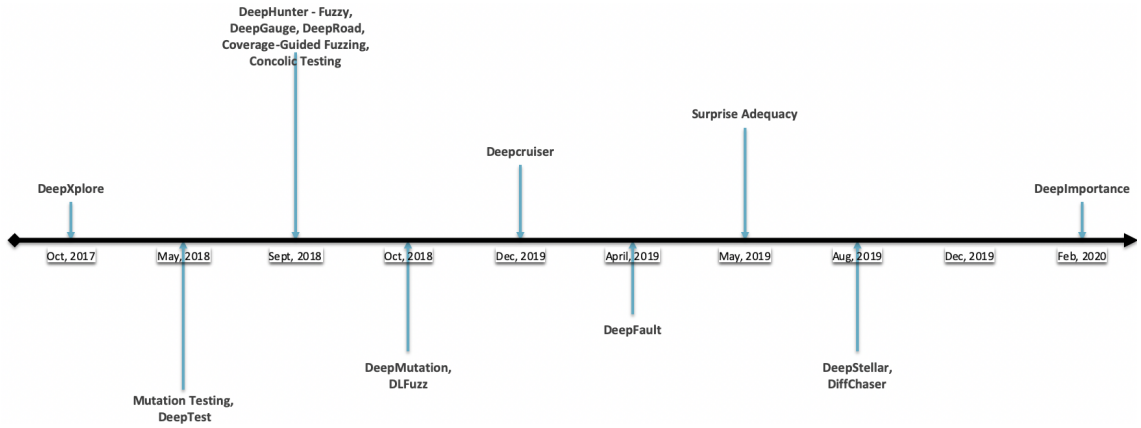

Figure 2.8: Timeline of DL Testing Tools Research, 2020

2.5.2 Research Distribution

Figure 2.9 shows commutative trends in **"Deep Learning System Testing"** and Figure 2.10 shows commutative trends in **"Testing "Deep Learning "**. This shows that there is a trend of moving from testing general machine learning to deep learning testing, as seen from the number of published papers in the field of DL testing and techniques for each year. Before 2017 and 2018, the research papers mostly focused on general machine learning; after 2018, we see a more dedicated focus on DL specific testing notably arise. The majority of publications on DL testing techniques came in 2019. The numbers shown in all three graphs also include survey publications in the field of DL testing.

Comparing to publications for machine learning testing, as early as in 2007, Murphy et al. [30] mentioned the idea of testing machine learning applications, which is one of the first papers about testing Machine learning systems. Figure 2.11 shows the commutative trends in **Testing ""Machine Learning "**. Table 2.2 shows the total number of publications and maximum citations for each category.

All the statistics are taken from Google Scholar as per their search relevance[1], January, 2020. The year displayed for each research paper is the year of publication.

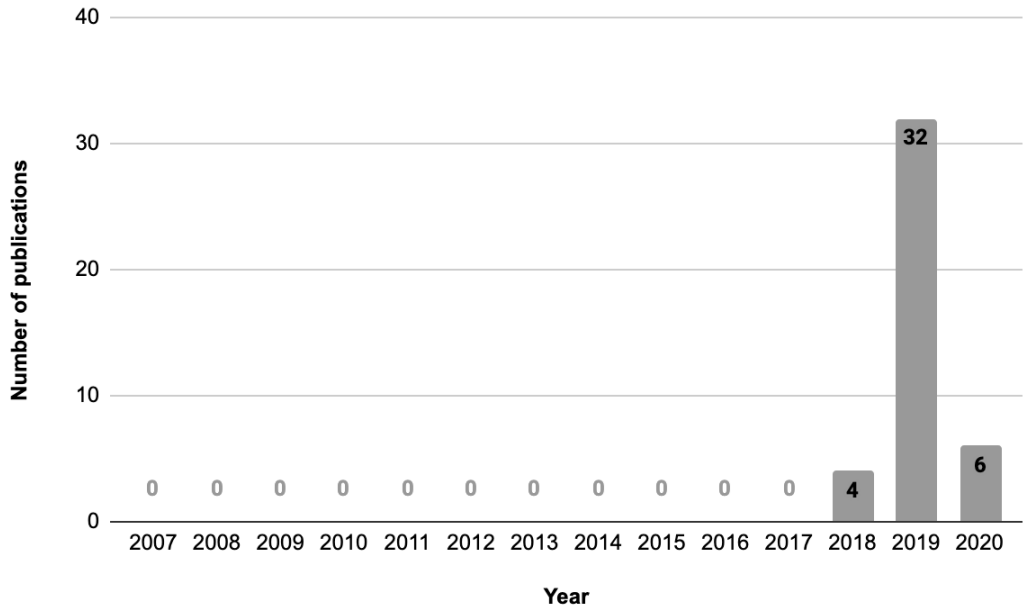

Figure 2.9: "Deep Learning System Testing" Publications

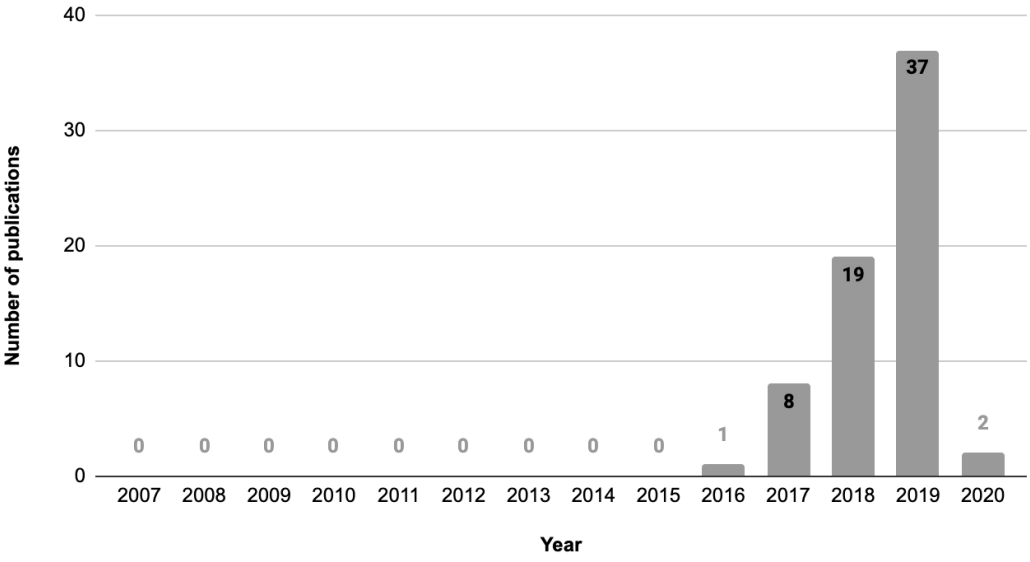

Figure 2.10: "Testing "Deep Learning"" Publications

[1]Google Scholar https://scholar.google.com/

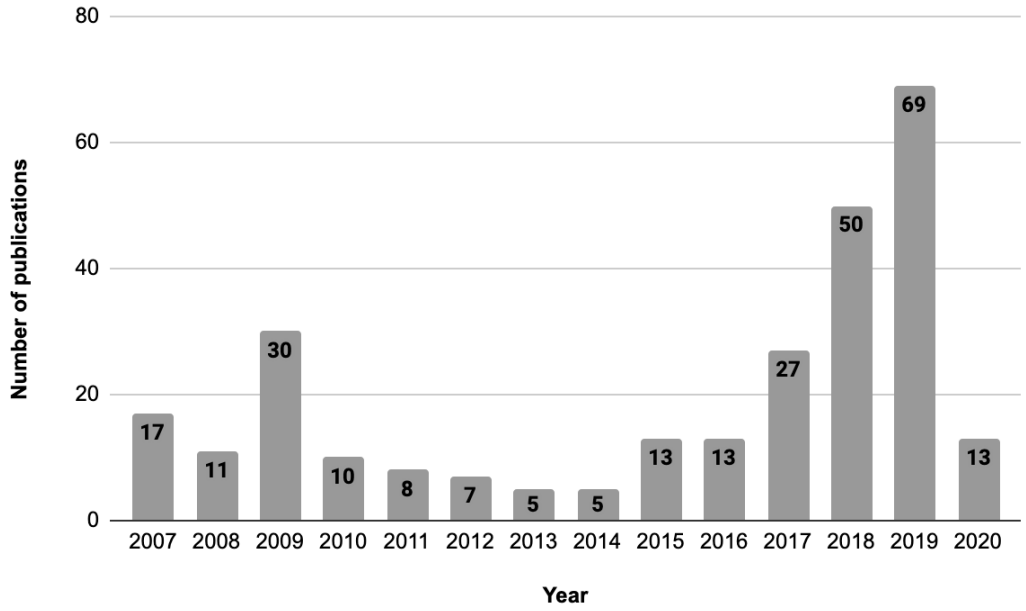

Figure 2.11: "Testing "Machine Learning"" Publications

Table 2.2: Total Research Publications and Citations

	Total Publications	**Maximum Citations**
Deep Learning System Testing:	42	55
Testing "Deep Learning":	67	345
Testing "Machine Learning":	294	152

2.5.3 DL Datasets

Datasets play a dominant role in shaping the future of technology. A lot of research papers these days use proprietary datasets that are usually not released to the general public. Table 2.3 to Table 2.8 show some key examples of widely-adopted and openly available datasets used in DL testing research. There are numerous ways how we can use these datasets. We can use them to apply various DL techniques. Some of these datasets are huge in size. In each table, the first column shows the name. The next three columns give dataset information, the size, and the total number of records for each type. The datasets can be divided into following six key categories, which includes:

1. Image Classification
2. Natural Language Processing
3. Audio/Speech Processing.
4. Biometric Recognition
5. Self-driving
6. Others

Table 2.3: DL Dataset: Image Classification

Dataset	Dataset Information	Size	Number Of Records
ImageNet	Images of WordNet phrases (Visual recognition dataset)	∼150GB	∼1,500,000
MNIST	Handwritten digits Images	∼50 MB	70,000 images in 10 classes
CIFAR-10	60,000 images of 10 classes (each class is represented as a row in the above image)	170 MB	60,000 images in 10 classes
MS-COCO	Object detection, segmentation and captioning dataset	∼25 GB	330K images, 80 object categories, 5 captions per image, 250,000 people with key points
Open Images Dataset	Open Images is a dataset of almost 9 million URLs for images	500 GB (Compressed)	9,011,219 images , more than 5k labels
VisualQA	Open-ended questions about images.	25 GB (Compressed)	265,016 images, at least 3 questions per image, 10 ground truth answers per question
The Street View House Numbers (SVHN)	Real-world image dataset for developing object detection algorithms.	2.5 GB	6,30,420 images in 10 classes
Fashion-MNIST	MNIST-like fashion product database	30 MB	70,000 images in 10 classes
LSUN	Large-Scale Scene Understanding to detect and speed up the progress for scene understanding	NA	10,000 images
Youtube-8M	large-scale video dataset that was announced in Sept 2016 by Google group.	1.53 Terabytes.	6.1 million YouTube video IDs, 2.6 billion of audio/visual features with high-quality annotations and 3800+ visual entities.

Table 2.4: DL Dataset: Natural Language Processing

Dataset	Dataset Information	Size	Number Of Records
IMDB Reviews	Dataset for movie lovers	80 MB	25,000 highly polar movie reviews for training, and 25,000 for testing
Twenty Newsgroups	Information about newsgroups 1000 Usenet articles from 20 different newsgroups	20 MB	20,000 messages taken from 20 newsgroups
Sentiment140	Dataset for sentiment analysis	80 MB (Compressed)	1,60,000 tweets
WordNet	Large database of English synsets.	10 MB	117,000 synsets
Yelp Reviews	Dataset by Yelp for learning purposes, consists of millions of user reviews, businesses attributes and over 200,000 pictures from multiple metropolitan areas	2.66 GB JSON, 2.9 GB SQL and 7.5 GB Photos (Compressed)	5,200,000 reviews, 174,000 business attributes, 200,000 pictures and 11 metropolitan areas
The Wikipedia Corpus	Collection of a the full text on Wikipedia.	20 MB	4,400,000 articles containing 1.9 billion words
Machine Translation of Various Languages	Training data for four European languages.	∼15 GB	∼30,000,000 sentences and their translations
The Blog Authorship Corpus	Dataset of blog posts collected from thousands of bloggers and has been gathered from blogger.com.	300 MB	681,288 posts with over 140 million words

Table 2.5: DL Dataset : Audio/Speech Processing

Dataset	Dataset Information	Size	Number Of Records
Free Spoken Digit Dataset	Dataset to identify spoken digits in audio samples	10 MB	1,500 audio samples
Ballroom	Dataset of ballroom dancing audio files	14GB (Compressed)	∼700 audio samples
Free Music Archive (FMA)	Dataset for music analysis.	∼1000 GB	∼100,000 tracks
Million Song Dataset	Freely-available collection of audio features and metadata for a million contemporary popular music tracks	280 GB	A million songs
LibriSpeech	Large-scale corpus of around 1000 hours of English speech	∼60 GB	1000 hours of speech
VoxCeleb	Large-scale speaker identification dataset	150 MB	100,000 utterances by 1,251 celebrities
Google AudioSet	Dataset from YouTube videos and consists of an expanding ontology, Categories cover human and animal sounds, sounds of musical instruments, genres, everyday environmental sounds, etc	2.4 gigabytes Stored in 12,228 TensorFlow record files,	2.1 million annotated videos that include 527 classes and 5.8 thousand hours of audio

Table 2.6: DL Dataset: Bio metric Recognition

Dataset	Dataset Information	Size	Number Of Records
Open Source Bio-metric Recognition Data	Dataset of tools to design and evaluate new bio-metric algorithms and an interface to incorporate bio-metric technology into end-user applications.	16 MB	open source code for facial recognition, age estimation, and gender estimation

Table 2.7: DL Dataset: Self-driving

Dataset	Dataset Information	Size	Number Of Records
Udacity self-driving challenge	self-driving challenge Dataset	∼500 MB to ∼60 GB	A millions of images.
Baidu Apolloscapes	Dataset for self-driving technologies	∼3 GB	25 different semantic items like cars, bicycles, pedestrian, street lights, etc. covered by 5 groups.

Table 2.8: DL Dataset: Others

Dataset	Dataset Information	Size	Number Of Records
Drebin	Applications from different malware families	NA	∼123,500
Waveform	CART book's generated waveform data	NA	5,000
VirusTotal	Malicious PDF files	NA	5,000
Contagio	Clean and malicious files	NA	∼29,000

2.6 Benchmarking Research

This section starts with answering four typical questions as we talk about benchmarking in general, such as *"What is 'benchmarking'?"*, *"Why do we conduct benchmarking activities?"*, *"What benefits does benchmark bring?"* and *"What can we actually benchmark?"*. Finally, it gives insight into benchmarking research work done in the field of Deep Learning.

2.6.1 What is Benchmarking?

Benchmarking is a widely used method in experimental software engineering, in particular, for the comparative evaluation of tools and algorithms. There are two aspects common to many benchmarking studies: [35] (i) Comparison of performance levels to ascertain which organization(s) is achieving superior performance levels. (ii) Identification, adaptation/improvement, and adoption of the practices that lead to these superior levels of performance. A benchmarking method is two-fold [29]. First, for the end-users of DL tools, benchmarking results can serve as a guide to selecting appropriate software tools. Second, for software engineers within the field, the in-depth analysis, and comparative conclusion points out possible future research directions to further optimize the properties of the software tools. Sim at al. [31] extends the meaning of performance. When talking about performance metrics, in the paper it is argued that performance is not just an innate software characteristic but also the interaction between the software and the user. This further expands the possible options when it comes to measuring software.

2.6.2 Benchmarking in DL Systems

Deep learning systems, which are the focus of our research, have been successfully deployed for a variety of tasks, and its popularity results in numerous open-source DL software tools. There is a guide for the selection of appropriate hardware platform and DL software tools [29] but there is no guide available as such to select appropriate DL testing tools and techniques. Additionally, there is an increasing trend in the research work done within the field of DL testing. To get benchmarking results, we need to have a reliable benchmarking method. There exist some challenges to get such a reliable method. Reliable benchmarking: Requirements and solutions by

Dorely [34] explains three major difficulties that we need to consider for benchmarking such as a technical bias for benchmarking framework design, hardware resources selection and the independence of different tool executions. Moreover, there is also a survey paper [30] on 'Machine Learning Testing: Survey, Landscapes, and Horizons' but this only focuses on Machine learning in general and it is just a survey on ML testing tools.

3
Methodology

This chapter starts with a description of the methods used for the three research questions. The following section explains the properties necessary for a benchmarking method. Furthermore, the Requirement-Scenario-Task model is presented, highlighting methods used to elicit DL testing requirements and derive testing scenarios. Finally, DL benchmarking scenarios and tasks are explained.

3.1 Research Questions

In this section, a description of the methods used to answer the three research questions is given.

RQ1: For RQ1 (as stated in section 1.4), we investigated fifteen existing state-of-the-art research papers on DL testing and techniques. All research papers were taken from Google Scholar pages[1] sorted by search relevance. The focus was to understand the DL testing workflows, components, and testing properties of the tools. We also studied the evaluation method, DL datasets, code support, and availability of each DL testing tool. The result of the pre-study has a summary of all fifteen DL testing tools and techniques and is presented in chapter 4.

RQ2: For RQ2, which includes *Hypothesis 1* and *Hypothesis 2*, as stated in section 1.4, we used the results from our literature review of fifteen research papers on DL testing tools and the result of the benchmarking tool on four DL testing tools.

Hypothesis 1: To answer *Hypothesis 1*, the testing properties of each DL testing tool were identified as a part of the literature review. The details of the qualitative differences across these tools are presented in chapter 4.
Hypothesis 2: To answer *Hypothesis 2*, a benchmarking method was designed. The output of the designed benchmarking method was used to find a significant difference in test input generation of four DL testing tools for a given type of dataset. The tools were selected based on their working status and code support. The results and analysis are explained in section 5.2.

RQ3: For RQ3 (as stated in section 1.4), relevant scenarios of DL testing tools were identified, out of which benchmarking tasks were designed to reflect real-world

[1] `Google Scholar https://scholar.google.com/`

scenarios. The method used to ensure the relevance of the scenarios is explained in Section 3.2. The designed benchmarking method was then presented to two industry researchers for validation by conducting a semi-structured interview for feedback on the method. The analysis and results of the design are presented in chapter 5 and section 5.2.

3.2 Benchmarking Method

For the benchmarking method design, we use the benchmark definition and methodology proposed by Sim et al. [31]. The paper contains the necessary benchmark components, properties, and guidelines which to follow to create a successful benchmark in software engineering. We focus on the proposed five relevant properties for creating a successful benchmark within the field of software engineering: relevance, solvability, scalability, clarity, and portability. Each property will be presented by giving the general description defined by Sim et al.[31] and followed by how it relates to a benchmarking method in DL.

- **Relevance**: The task set out in the benchmark must be representative of ones that the system is reasonably expected to handle in a natural (meaning not artificial) setting and the performance measure used must be pertinent to the comparisons being made. The relevance property in the case of the DL testing tool benchmark is related to the task sample but also to the components that are part of the task. Therefore, the benchmarking tasks, and the datasets and DNN models which are part of the tasks, need to be representative of actual data and situations that the system is expected to handle as well.

- **Solvability**: It should be possible to complete the task domain sample and to produce a good solution. Similar to the relevance property, the solvability property is concerned with the task sample. The task set needs to be comprised out of tasks that are not too difficult for the DL testing tools to handle, as well as being not too simple for the tools to show their potential as best as possible.

- **Scalability**: The benchmark tasks should scale to work with tools or techniques at different levels of maturity. This property involves the benchmark being able to work with as wide a range of maturity of the DL testing tools as possible. The initial benchmark will be meant to work with the currently available DL testing tools. The benchmarking method, however, may contain tasks that may seem not as predominant at the moment but show potential for future development. It is important to note that a benchmark is a continuous effort, it will need to be updated as the tools evolve and should be able to accommodate or account for such updates.

- **Clarity**: The benchmark specification should be clear, self-contained, and as short as possible. This clarity should help ensure that there are no loopholes to be exploited. This study serves as the specification of the benchmarking

method. However, the tool in which the method is implemented is accompanied by a self-contained configuration file that is supported by a help file. This property will be tested by presenting the benchmarking method to industry researchers for validation and feedback. Through the feedback, we'll discover if the benchmark's clarity property is fulfilled.

- **Portability**: The benchmark should be specified at a high enough level of abstraction to ensure that it is portable to different tools or techniques and that it does not bias one technology in favor of others. The portability property of the DL benchmark is concerned with the applicability of the benchmark design on different platforms and languages. The benchmark specification, which is the study itself, should give sufficient information for the abstract concept of the benchmark to be applied on any language or platform, if possible.

By conforming to the properties, benchmarking tool was implemented by using the Requirements-Scenario-Task model, which is described in the following subsection.

3.2.1 Requirement-Scenario-Task Model

As pointed out by Sim et al. [31], performance is not an innate characteristic of the software when creating a benchmark but "the relationship between technology and how it is used" and that creativity may be necessary to device meaningful performance metrics. Therefore, we took inspiration from the model presented by Bai et al. [32]. The paper presents a scenario-based modeling technique, which captures system functionality at different abstraction levels and can be used to direct systematic testing. The inspiration that our study took, is from its key point of using requirements to give a functional view of the system, whereas the scenarios give a user's point of view of the system under test. What this means is that by eliciting requirements to get a functional view of the DL tools, we can then synthesize relevant test scenarios to get a user's perspective on the tools. The user, in this case, is a DL tester. By having the user's perspective on what a DL tool should contain in terms of relevant functionality for testing, i.e. selecting a model should be possible, benchmarking tasks can be made that test the tools that are in an early state for whether they have the necessary functionality. Additionally, one requirement could possibly lead to multiple scenarios, hence making scenarios a more fine-grained method for constructing tasks. By establishing requirements based on the investigation done for **RQ1** and creating testing scenarios out of these requirements, we can both ensure task relevance and have a two-dimensional view (functional and user view) of the tools that the benchmark is meant to test.

Figure 3.1 shows the overview of the Requirement-Scenario-Task model for the benchmarking method. The details of each component of the model are explained in the following subsections.

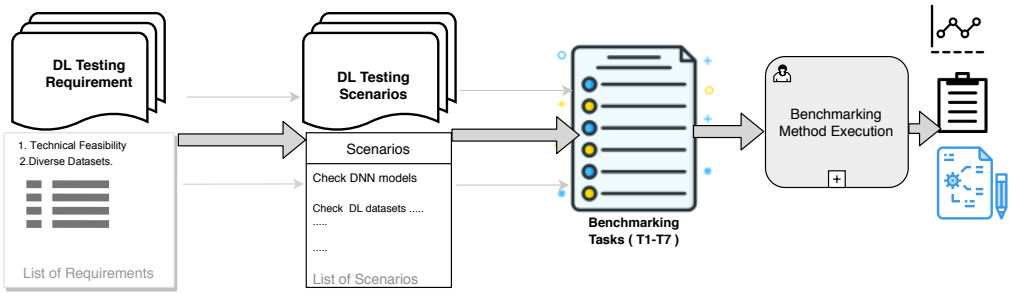

Figure 3.1: Requirement-Scenario-Task Model for the Benchmarking Method

3.2.2 DL Testing Tool Requirements

For the elicitation of the requirements, we conducted a pre-study on fifteen DL testing tool research papers to first get to know the domain extensively by observing the intricacies involved in DL testing, the details of which can be found in Section 2.2. Furthermore, we proceeded with familiarizing ourselves with the differences between traditional and DL software testing techniques in Section 2.3 and the challenges that DL testing imposes, found in Section 2.4. These are necessary pre-conditions to understanding the domain before both requirements can be extracted and meaningful testing scenarios can be made. For the elicitation process, we used elicitation techniques that were reviewed by Sharma et al. [33]. The two main techniques used were *Reading Existing Documents*, complemented with a *Brainstorming* session. For the existing documents, we referred to the DL testing tool papers, which were investigated for **RQ1** as well as a taxonomy [10] about common faults in real-world DL systems. The brainstorming session was conducted between the researchers involved in this study.

3.2.3 DL Testing Scenarios

From the set of requirements, we established benchmark testing scenarios for the DL testing tools. Although a benchmark is used for comparisons, it is a testing tool that consists of testing tasks. Such testing tasks usually are aimed at scenarios that occur within the subject under test. As such, after prioritizing the requirements into what is most relevant to the DL testing tools, we established a set of scenarios. This process is very important for one of the seven properties proposed by Sim et al. [31] for a successful benchmark, **Relevance**. The tasks set of the benchmark must be representative of what the tools are supposed to handle.

Benchmarking Test Scenarios:
The benchmarking test scenarios are the scenarios that are applicable to a benchmarking effort. Before the scenarios can be synthesized into tasks for the benchmark, another important property must be considered at this stage of the design and that

is **Solvability**, "it should be possible to complete the task domain sample and to produce a good solution" [31]. Out of all DL Testing Scenarios, not all the testing scenarios were applicable for benchmarking for two reasons. Firstly, to focus on the scenarios that are solvable by the DL testing tools in their current state, and secondly, to filter out the less relevant benchmarking scenarios. After filtering out the inapplicable DL testing scenarios, benchmarking test scenarios were identified.

3.2.4 Benchmarking Tasks

After the benchmarking scenarios have been filtered out, benchmarking tasks were synthesized which are solvable to the DL testing tools. Ideally, the tasks that a benchmark should give to the tools would strike a balance between difficulty and feasibility. The tasks need to be difficult enough for a tool to show their full potential but not too difficult for it to give any meaningful output [31]. The tasks were aimed at the capabilities that the tool has and was designed to address relevant issues within the DL testing tools. The scenarios provide the groundwork for the individual tasks but the tasks extend the groundwork into meaningful objectives for the tools.

Building Benchmarking Models

The benchmarking tool was complemented by six DNN models for its testing process against DL testing tools. Two models for each type of dataset classifications (Images, Texts, and Self-driving) were included. Fours models were compiled using the Keras Models Directory[2]. Two models for driving datasets were taken from Udacity self-driving dataset. The details of each model and its architecture are presented in Appendix A.8.

[2]`Keras Models Directory https://github.com/keras-team/keras/tree/master/examples`

4
Pre-Study Results

4.1 Results from Pre-Study

The goal of the pre-study is to investigate existing state-of-the-art and real-world applicable DL testing tools research papers. In this section, we present the results of the investigation on the fifteen DL testing tools and techniques.

4.1.1 DL Testing Tools

Table 4.1 shows a summary of fifteen DL testing tools and techniques. The first column shows the name of each tool/technique, followed by the year of publication, type of testing, and DL datasets applied by the testing tool in the research paper. We also investigated the correctness, fairness, efficiency, and robustness testing properties by these DL testing tools.

DL Testing Properties:
A comparison was done among of most common DL testing properties across the selected fifteen research papers on DL testing and techniques to find out which properties are the most commonly tested ones. The information on the properties was extracted directly from their respective research papers. The properties that the tools test on the DL system were either explicitly stated in each paper or implied. Figure 4.1 depicts the Research distribution among different DL testing properties. As seen in the figure, robustness is currently the most prominent property of concern for the DL testing research community, which is common across thirteen out of fifteen research papers. It is followed by Reliability, Correctness, and Generality. The paper by Jahangirova et al. [10] identified many real faults within DL systems, however, they were primarily technical faults related to the implementation, configuration, and training of the models. The interesting fault that is noted within the paper is the selection of the wrong model. This is related to the Relevance property of a model. Different types of DNNs perform better at certain tasks, i.e. recurrent neural networks (RNNs) perform better at sequential input streams[27], however, the relevance of a model, which we marked as Model Relevance, is not a point of concern within the fifteen DL testing tool papers. Similarly, fairness and efficiency were not elaborated upon within the research papers.

In the rest of the section, the investigated fifteen tools and techniques are presented. Each tool is presented by first introducing the core concept of the tool, followed by details of its setup and evaluation, supported dataset classifications, type of testing

and finally properties tested by the DL testing tool.

Table 4.1: Summary of the Investigated State-of-the-art DL Testing Tools and Techniques

Tool Name	Publication Year	Type of Testing	DL Datasets
DeepXplore	2016	Whitebox Differential	Images Self Driving Textual
Surprise Adequacy	2019	Test Adequacy Criterion	Images Self Driving
DLFuzz	2018	Differential Fuzzing	Images
DeepConcolic	2018	Concolic	Images
DeepFault	2019	Whitebox	Images
nMutant	2018	Mutation	Images
DeepTest	2018	Metamorphic	Autonomous Driving
DeepHunter - Fuzzy	2018	Coverage Guided Fuzzing	Images
DeepGauge	2018	Multi-granularity test criteria	Images
DeepMutation	2018	Mutation	Images
Deepcruiser	2018	Coverage Guided Metamorphic	Speech-to-text
DeepRoad	2018	Metamorphic	Autonomous Driving
DeepStellar	2018	Coverage Guided	Images Speech-to-text
Coverage-Guided Fuzzing	2018	Coverage-Guided Fuzzing	Images
DiffChaser	2019	Black-box	Images

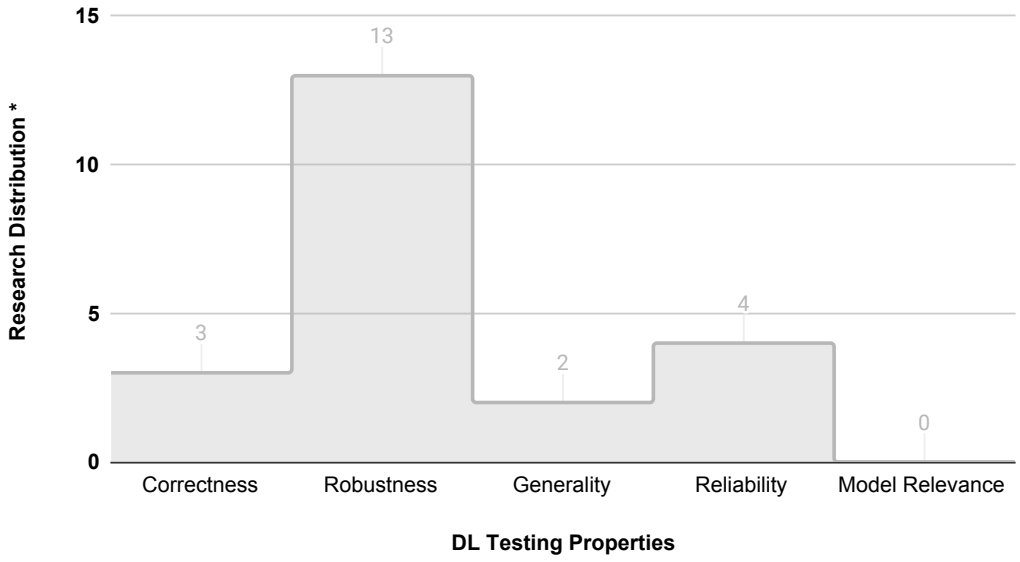

Figure 4.1: DL Testing Properties Research Distribution

1. **DeepXplore**
   DeepXplore [2] is the first white box framework for systematically testing real-world DL systems. It proposes a test effectiveness metric called neuron coverage and develops a neuron-coverage guided differential testing technique that uncovers behavioral inconsistencies between different deep learning models. DeepXplore performs gradient ascent to solve a joint optimization that maximizes both neuron coverage and the number of potentially erroneous behaviors. It covers three important aspects: (i) Systematically test deep neural nets (DNNs), (ii) Differential testing of multiple DNNs without manual labeling, and (iii) Improve test coverage of DNN by means of Neuron Coverage. Neuron Coverage aims to maximize the number of activated neurons during testing. Neuron coverage of a set of test inputs is defined as the ratio of the number of unique activated neurons for all test inputs and the total number of neurons in the DNN. A neuron is considered to be activated if its output is higher than a threshold value (e.g., 0). Note that this is just one way of defining neuron coverage.
   Differential Testing, on the other hand, aims to Systematically test DL systems by levering multiple DL systems with similar functionality as cross-referencing oracles to identify unexpected erroneous corner cases without manual checks.
   - **Setup and Evaluation:**
     DeepXplore is implemented using Python 2.7 on the Keras (2.0.3) DL framework with TensorFlow (1.0.1). Some of the datasets that it evaluates are Udacity self-driving car challenge, image classification from ImageNet and MNIST, Android malware data from Drebin, and PDF malware data from VirusTotal. Its tests measure the Neuron Coverage of a model.
   - **Type of Testing**
     DeepXplore involves the Whitebox framework and Differential Testing. DeepXplore, being the very first white-box testing technique in the field of DL testing, has a few limitations based on its approach. It doesn't fully capture all real-world input distortions, such as simulating shadows from other objects still remains an open problem. It doesn't cover realistic transformations i.e. to emulate different weather conditions (e.g., snow or rain) for testing self-driving vehicles. Finally, a key limitation of the used gradient-based local search is that it does not provide any guarantee about the absence of errors.
   - **Tested Properties**
     Robustness, Correctness, and Reliability.

2. **Surprise Adequacy:**
   Surprise Adequacy for Deep Learning Systems (SADL) [9] recognizes that there is a need to focus on improving the training data used to train the model. According to it, the testing data needs to contain several, but not too many, 'surprising' inputs. What that means is that the DL system should be trained in a way as to be able to respond correctly towards corner-case scenario input, taking images as an example, the DL system should be able to tell

the number “nine” even if there is a weird fluctuation in the image, just like a human would. While the concept of adversarial input is not new, DL systems behaving incorrectly when such input is introduced is still a wide-spread problem due to the difficulties of obtaining quality training data [10]. The general idea behind solving this problem is improving training data quality and SADL adds a new layer to the currently expanding body of knowledge by proposing a novel test adequacy criterion that helps improve training data classification (selection). The problem with training data quality is that the model needs to be trained for corner-case scenarios. Such scenarios need to be identified before a model can be trained for them. To identify such scenarios the testing input needs to be diverse, covering both inputs which were used during training and input that is very different from the training data used. This ensures the model handles familiar input correctly as well as similar to unfamiliar input which may influence its accuracy. Several DL testing techniques [2] [11] focus on treating the input as sets or buckets, in general, bundled together inputs, leaving individual input value out of the equation when presenting the end result of the test. This way, while giving general information about model performance when facing such input, information about individual values which could improve training data quality is lost. SADL tests the model by measuring how ‘surprising’ the input is to the DL system with respect to the training data used [9] while retaining the value of the individual inputs. This way the inputs can be ‘cherry-picked’ and added to the training data, improving the diversity of the data, and overall model performance. However, this process is not yet automated and would require manual labeling when adding the identified data.

- **Setup and Evaluation**
  SADL is implemented using Python on the Keras (2.2.0) framework with TensorFlow (1.9.0). It has been evaluated on four datasets: MNIST and CIFAR-10 for image classification, Dave-2, and Chauffeur for the self-driving car challenge. It uses Likelihood-based Surprise Adequacy and Distance-based Surprise Adequacy and presents the results in a ROC-AUC score and FGSM coverage score.

  LSA (Likelihood-based Surprise Adequacy) uses Kernel Density Estimation to estimate the probability density of each activation value within an activation trace and obtain the surprise of a new input with respect to the estimated density.
  DSA (Distance-based Surprise Adequacy) uses the distance between activation traces as a measurement of surprise. Basically it checks the distance between the activation trace of new input and the activation trace observed during training.

  ROC (Receiver Operating Characteristic) determines how well a model can distinguish between two things. The AUC (Area Under Curve) score, on the other hand, shows how well the model performs. ROC-AUC indicates how well the probabilities from the positive classes are separated from the negative classes. Surprise Adequacy uses the ROC-AUC score to get a % estimate of how well adversarial examples are classified, the higher

the score the better the adversarial example classifiers when computing LSA or DSA. The higher the score the more clear the differentiation between the adversarial examples and actual test data. That means that a high score shows a clear 'surprise' in input, i.e. the input is diverse and different from the data that was used to train the model. On the other hand, lower values show that the input is less surprising to the data that was used to train the model.

- **Type of Testing**
  Surprise Adequacy is a test adequacy criterion. It is aimed at improving training data quality by measuring how surprising an input is and retaining that input's individual value by not grouping neurons into sets, buckets, etc. The word 'adequacy' in this case is aimed at the testing data, how adequate is the training data? Test adequacy criteria, in general, are aimed at specific parts of testing and improving their quality. It also a good Code Support for setup simulation
- **Tested Properties**
  Robustness and Correctness.

3. **DLFuzz**
   DLFuzz [3] is a differential fuzzing testing framework that approaches cross-referencing in a different manner than DeepXplore [2]. Having identified that DL systems give incorrect output on nearly identical images with small perturbations, the framework is based on giving the same system two nearly identical inputs, where the original is used as a cross-reference to the new nearly identical input. This nearly identical input is generated through mutation of the original input by applying a very slight change to it that is nearly indistinguishable to the human eye. Being essentially the same input (in this case images), the system is supposed to give the same prediction as for the original input. If the output is different, the system is clearly misbehaving because it fails the cross-reference, hence the input given to reach this incorrect behavior is an important adversarial input that is retained. As such it can be used to improve neuron coverage and the accuracy of the DL system under test.
   - **Setup and Evaluation**
     DLFuzz is implemented using Python on the Keras (2.1.3) framework with TensorFlow (1.2.1) as backend. It has been evaluated using two datasets: MNIST and ImageNet for image classification on a CNN model. It measures Neuron Coverage and execution time.
   - **Type of Testing**
     DLFuzz is a whitebox framework that uses differential fuzz testing. It mutates the original input using slight perturbations into new input which is imperceivable. The differential part comes from the original input being used as a cross-referencing oracle for the mutated input, hence removing the need for a second DL system. It has has a good Code Support.
   - **Tested Properties**
     Robustness and Reliability.

4. **Concolic Testing for Deep Neural Networks**
   Concolic Testing for Deep Neural Networks [15] was published in 2018. Concolic testing alternates between CONCrete program execution and symbOLIC analysis to explore the execution paths of a software program and to increase code coverage. It explains the first concolic testing approach for Deep Neural Networks (DNNs). More specifically, utilize quantified linear arithmetic over rationals to express test requirements that have been studied in the literature, and then develop a coherent method to perform concolic testing with the aim of better coverage. The experimental results of the tool show the effectiveness of the concolic testing approach in both achieving high coverage and finding adversarial examples.
   - **Setup and Evaluation**
     DeepConcolic is implemented in python using the Keras framework. It evaluates two datasets: MNIST and CIFAR-10 for image classification. It measures Neuron Coverage, SS Coverage, and Neuron Boundary Coverage on a subset of neurons.
   - **Type of Testing**
     DeepConcolic uses concolic testing to demonstrate its effectiveness towards increasing test coverage. It has good code support and allows model selection in its run command –model MODEL but works a bit slow on commodity laptop to get the output.
   - **Tested Properties**
     Robustness.

5. **DeepFault**
   DeepFault [18] (published in April 2019), approach for whitebox testing of DNNs driven by fault localization, to establish the hit spectrum of neurons and identify suspicious neurons whose weights have not been calibrated correctly and thus are considered responsible for inadequate DNN performance. It uses an algorithm for identifying suspicious neurons that adapts suspiciousness measures. It uses a suspiciousness-guided algorithm to synthesize new inputs that achieve high activation values of potentially suspicious neurons. The objectives of DeepFault are twofold: (i) identification of suspicious neurons, i.e., neurons likely to be more responsible for incorrect DNN behavior; and (ii) synthesis of new inputs, using correctly classified inputs, that exercise the identified suspicious neurons. It employs a suspiciousness-guided algorithm to synthesize new inputs, that achieve high activation values for suspicious neurons, by modifying correctly classified inputs.
   - **Setup and Evaluation**
     DeepFault is implemented using python on the Keras (2.2.2) framework with TensorFlow (1.10.1) backend. It is evaluated on two datasets: MNIST and CIFAR-10 for image classification. It measures the suspiciousness of neurons by applying an algorithm that activates these neurons.
   - **Type of Testing**
     DeepFault is a whitebox testing approach.
   - **Tested Properties**

Robustness and Generality.

6. **Adversarial Sample Detection for Deep Neural Network through Model Mutation Testing**
   The paper proposes nMutant [26], an adversarial sample detection algorithm that detects adversarial samples on run-time. The tool is based on an observation that adversarial samples are much more sensitive to random perturbations than normal samples. Being inspired by mutation testing, the method used for adversarial sample detection measures how sensitive a sample is too random perturbations and raises an alarm if it's above a certain threshold. Additionally, the approach does not require any underlying knowledge of the DNN system, thus being applicable to a wide range of DL systems.
   - **Setup and Evaluation**
     nMutant is implemented as a stand-alone Java application, simple and easy download from GitHub. It is evaluated to two datasets: MNIST and CIFAR-10 for image classification. It measures the number of identified adversarial samples.
   - **Type of Testing**
     nMutant is a mutation testing approach towards identifying adversarial samples.
   - **Tested Properties**
     Robustness.

7. **DeepTest**
   DeepTest [16] focuses on automatically detecting erroneous behaviors of DNN-driven vehicles that can potentially lead to fatal crashes. The paper presents thousands of erroneous behaviors under different realistic driving conditions. It aims to explore different parts of the DNN logic by generating test inputs that maximize the numbers of activated neurons.
   - **Setup and Evaluation**
     DeepTest is implemented using python on the Keras framework. We were unable to get specific version information. It is evaluated on three models for autonomous driving based on the Udacity dataset. The Chauffeur and Rambo models were pre-trained, the Epoch model has been trained by the authors. It measures neuron coverage.
   - **Type of Testing**
     DeepTest uses a systematic testing methodology to detect erroneous behavior.
   - **Tested Properties**
     Robustness.

8. **DeepHunter**
   DeepHunter [24] is a coverage-guided fuzzing framework for testing general purpose DNNs. The paper proposes a metamorphic mutation strategy to generate its tests, whilst leveraging multiple coverage criteria as feedback for the fuzzer to guide test generation. Within DeepHunter there are 4 seed

strategies and 5 existing testing criteria incorporated.

- **Setup and Evaluation**
  DeepHunter is implemented using python on the Keras (2.1.3) framework with Tensorflow (1.5.0) backend. It is evaluated on three datasets: MNIST, CIFAR-10, and ImageNet for image classification. It measures coverage using different testing criteria.
- **Type of Testing**
  It uses coverage-guided fuzzing to generate its tests and metamorphic relations for detecting errors.
- **Tested Properties**
  Robustness

9. **DeepGauge: Multi-Granularity Testing Criteria for Deep Learning Systems**
   DeepGauge [11] proposes a set of multi-granularity test criteria that focus on gauging the testing adequacy. It measures the neuron activation on various layers to facilitate the understanding of DNN and test data quality from different angles as to further the knowledge-base on DL systems logic. By giving a multi-angle view of the test-bed DeepGauge hopes to further the progress towards the construction of more robust and generic DL systems.
   - **Setup and Evaluation**
     DeepGauge is implemented using python on the Keras (2.1.3) framework and Tensorflow (1.5.0) backend. Its evaluation is done on two datasets: MNIST and CIFAR-10 for image classification. It uses five different metrics for its evaluation, three of which are on neuron-level coverage and two on layer level coverage.
     KMNC (k-multisection neuron coverage): measures how thorough the given set of test inputs T covers the range of [lown, highn].
     NBC (neuron boundary coverage): Measures how many corner-case regions have been covered (upper and lower case boundary value) by a given test input set.
     SNAC (strong neuron activation coverage): Measures how many corner-case regions have been covered (the upper boundary value) by a given test input set.
     TKNC (top-k neuron coverage): Measures how many neurons have once been the most active k neurons on a layer.
     TKNP (top-k neuron pattern): denotes different kinds of activated scenarios from the top hyperactive neurons of each layer.
   - **Type of Testing**
     DeepGauge is a multi-granularity testing criterion that uses multiple angles of neuron coverage for its basis.
   - **Tested Properties**
     Robustness and Generality.

10. **DeepMutation: Mutation Testing of Deep Learning Systems**
    DeepMutation [14] proposes a mutation testing framework that is specialized for DL systems to measure the quality of test data. It recognizes that the behavior of a DL system is determined by both the DNN structure and the connection weights in the network. The structure of the DNN is determined by code fragments of a training program in a high-level language, whereas the weights are determined by the training data used to train the model. Therefore, DeepMutation focuses on both the training data set and the training program by using mutation operators to inject potential faults in either. Afterward, the training process is re-executed, and mutated DL models are generated. Each mutated model is afterward analyzed against a test set in correspondence to the original DL model.
    - **Setup and Evaluation**
      DeepMutation is implemented using python on the Keras (2.1.3) framework and Tensorflow (1.5.0) backend. Its evaluation is done on two datasets: MNIST and CIFAR-10 for image classification. It is evaluated using the Mutation Score and Average Error Rate as metrics.
    - **Type of Testing**
      As the name implies, it uses mutation testing, hence why Mutation Score is the metric of choice.
    - **Tested Properties**
      Robustness and Generality.

11. **DeepCruiser: Automated Guided Testing for Stateful Deep Learning Systems**
    DeepCruiser [27] is an automated testing framework for RNN (recurrent neural network) based stateful DL systems. The tool systematically generates tests on a large scale based on specialized coverage criteria for stateful DL systems to uncover defects. The effectiveness of the technique used within the tool is demonstrated by showing how the basic state coverage is improved through its usage over a period of 12 hours.
    - **Setup and Evaluation**
      We were unable to uncover data for the setup of DeepCruiser in relation to the framework and backend that it uses. Its evaluation is done using one dataset: DeepSpeech-0.3.0 for automated speech recognition. Its measurement is done using two state-level coverage criteria and three transition-level coverage criteria for the automated test generation. The state-level coverage criteria focus on the internal states of the RNN, whereas the transition-level coverage criteria target the abstract transition activation by various input sequences.
    - **Type of Testing**
      DeepCruiser uses Coverage-Guided testing in which new coverage criteria are proposed for RNN-based stateful systems.
    - **Tested Properties**
      Reliability.

12. **DeepRoad**
    DeepRoad [17] is a GAN-based Metamorphic Autonomous Driving System Testing. It is an unsupervised framework to automatically generate large amounts of accurate driving scenes to test the consistency of DNN-based autonomous driving systems across different scenes. In particular, DeepRoad delivers driving scenes with various weather conditions (including those with rather extreme conditions) by applying the Generative Adversarial Networks (GANs) along with the corresponding real-world weather scenes.
    - **Setup and Evaluation**
      We were unable to uncover data for the setup of DeepRoad in relation to the framework and backend that it uses. It is evaluated on four datasets: Udacity, Autumn Chauffeur, and Rwightman for self-driving. The Autumn dataset is used within a CNN, whereas Chauffeur is used for one RNN and one CNN model. It measures the inconsistent behavior of the model. Consistent behavior is considered to be aligned with the steering angle prediction within certain error bounds.
    - **Type of Testing**
      DeepRoad uses a GAN-based metamorphic testing approach for its tests.
    - **Tested Properties**
      Robustness and Generality.

13. **DeepStellar: Model-Based Quantitative Analysis of Stateful Deep Learning Systems**
    DeepStellar [13] is a general-purpose quantitative analysis framework for RNN-based DL systems. The tool uses two trace similarity metrics (STS and TTS) to quantify the prediction proximity of different inputs and five coverage criteria (BSC, WSC, n-SBC, BTC, and WTC) to measure the adequacy of the testing input. The tool creates an abstract model which contains a trace in the form of RNN state vectors. These state vectors are abstracted due to the sheer number of states that an RNN model contains beyond analysis capability. This abstract model is afterward applied on two applications (adversarial sample detection and coverage-guided testing) using the defined metrics and criteria to both capture adversarial samples on run-time and improve coverage by uncovering defects for quality assurance.
    - **Setup and Evaluation**
      DeepStellar is implemented using python on the Keras (2.2.4) framework with TensorFlow (1.4, 1.8, and 1.11) backend. It evaluates two types of datasets on RNN-based models: MNIST for image classification and DeepSpeech (0.1.1 and 0.3.0) for automated speech recognition. It measures the coverage % by using the defined coverage criteria.
    - **Type of Testing**
      DeepStellar uses quantitative measures for adversarial sample detection and converge-guided test generation for its tests.
    - **Tested Properties**
      Robustness

14. **Coverage-Guided Fuzzing**
    Coverage-Guided Fuzzing is implemented in DeepHunter [24] was published in 2018 and evaluates images classification datasets.
    - **Setup and Evaluation**
      The paper uses DeepHunter for its evaluation, therefore it shares the same framework information and uses the same types of datasets for its tests.
    - **Type of Testing** It uses coverage-guided fuzzing to generate its tests and metamorphic relations for detecting errors.
    - **Tested Properties**
      Robustness and Reliability.

15. **DiffChaser**
    DiffChaser [25] is based on the idea of Detecting Disagreements for Deep Neural Networks. It uses FDI (first disagreement input), TDI (unique disagreement inputs), Success Rate, Total Time. Typically DL systems undergo an optimization phase before deployment. That optimized version of the DL system may have inconsistencies with the pre-optimized version which bypasses the test cases. Due to the aforementioned, the importance of quality test data increases. As such DiffChaser is aimed at identifying these differences between the version variants of a DNN by automatically generating input near the decision boundary between the two versions by using a mutation testing approach. This improves the quality of the test data which in return provides important information regarding the inconsistencies between the optimized and pre-optimized DL version.
    - **Setup and Evaluation**
      DiffChaser is implemented using python on Keras (2.1.3) framework with TensorFlow (1.5.0) backend. It is evaluated on two datasets: MNIST and CIFAR-10 for image classification. It measures the success rate to generate disagreements and execution time.
    - **Type of Testing**
      DiffChaser uses an automated black-box testing approach for detecting untargeted/targeted disagreements between DNN version variants.
    - **Tested Properties**
      Correctness

4.1.2 DL Testing Tools Availability

We investigated a total of fifteen DL testing tools and techniques, not all tools are publicly available as some of them are under NDA (Non-disclosure agreement). Table 4.2 shows a list of the fifteen investigated DL testing tools and the status of their code availability.

Table 4.2: List of DL Testing Tools and their Availability

DL Testing Tool	Availability
DeepXplore[2]:	Yes
Surprise Adequacy[9]:	Yes
DLFuzz[3]:	Yes
Concolic Testing[15]:	Yes
DeepFault[18]:	Yes
nMutant[26]:	Yes
DeepTest[18]:	Yes
DeepHunter - Fuzzy[24]:	Yes
DeepGauge[11]:	NDA
DeepMutation[14]:	NDA
Deepcruiser[27]:	NA
DeepRoad[17]:	NA
DeepStellar[13]:	NA
Coverage-Guided Fuzzing[23]:	Yes
DiffChaser[25]:	NA

Out of these fifteen testing tools, six tools are either not publicly available or have no information about the code support by the authors (marked as 'NDA' or 'NA' in Table 4.2). Out of the remaining nine tools, we managed to execute only four tools in our local environment by following the guidelines published with the tool. The nine testing tools are also made open-source by the authors but this study involves using the testing tools without changing the functionality of the tools. Only four tools namely DeepXplore [2], Surprise Adequacy [9], DLFuzz [3], and DeepFault[18] are used for the benchmarking evaluation in this study.

5
Benchmarking Method Results

This section covers the overall benchmark construct. This includes the requirements elicited, the scenarios which were synthesized out of the requirements, and how the benchmarking tasks are made to conform to the scenarios. Furthermore, the benchmarking tool is presented with its tasks and steps of execution. Finally, the benchmarking tool results are presented on four DL testing tools.

5.1 Benchmarking Method

The list of DL testing tool requirements and scenarios are presented in Table 5.1. An important observation is that the requirements and scenarios look nearly identical. In the beginning, there was no way of knowing that the scenarios would turn out nearly identical to the requirements, thus it made sense to use the requirements-scenario-task model from a methodological point of view. Furthermore, one requirement might lead to multiple scenarios, i.e. requirement 4. Re-Training Models leads to two scenarios as shown in Table 5.1. Out of the scenarios, benchmarking tasks have been synthesized which can be found in subsection 5.1.3.

Table 5.1: DL Testing Tool Requirements and Scenarios

DL Testing Tool Requirements	DL Testing Tool Scenarios
1. Technical Feasibility	Check DNN models selection capability
2. Diverse Datasets	Check diversity in DL datasets test tasks
3. Execution Time	Check test execution time
4. Re-Training Models	Check retraining of the DNN model under test Check automatic labeling of test input
5. Test Input Generation	Check adversarial input generation
6. Cross-Referencing Model	Check DL system cross-referencing oracle support
7. Output Validation	Check testing output validation and readability
8. Platform Support	Check multiple platform support
9. Code Support	Check DL testing tool code availability

5.1.1 DL Testing Tool Requirements

The proposed requirements are generated according to our methodology presented in Section 3.2.2. These requirements are there to help understand what a DL testing tool should consider according to current research on DL testing and traditional

testing. All requirements are further explained in the remaining of the section.

Technical Feasibility
The technical feasibility is related to the model selection capability of a DL testing tool. During the investigation of the tools, it was discovered that most of the tools do not provide an option to select a model to test. Most of the tools we managed to get working use their own pre-trained fixed models, for instance, DeepXplore [2] and DLFuzz [3] or have a training program that trains a model and then the tool uses that model [9].

Diverse Datasets
No matter how good a testing technique maybe, if it is not made for the type of dataset that the DL system under test uses, then it cannot be used. Examples of the diversity are DLFuzz [3], DeepMutation [14] and DeepGauge [11] that conduct their tests on images classification, DeepStellar [13] and DeepCruiser [27] on speech-to-text, whereas DeepRoad [17] and DeepTest [16] cover autonomous driving. DeepXplore [2] on the other hand, covers several datasets like image classification, autonomous driving, and malware-detection.

Execution Time
Similar to the previous requirement, even if the testing tool is good, if it takes too long to execute, which is deep learning may end up in days or even weeks depending on the hardware, it cannot be applied easily. In additional support to this requirement is the fact that execution time has been used as a metric when comparing tools to showcase improvement, i.e. DLFuzz [3] uses execution time in its comparison to DeepXplore [2].

Re-Training Models
The next important requirement is the re-training of the DL system models. This requirement is important if it is taken into consideration that many testing techniques use adversarial sample detection to discover special points of data which when added to the training data, improve the system robustness. Being able to retrain the system with the discovered adversarial samples opens up different possibilities for testing and is generally helpful. SADL [9] is aimed at finding 'surprising' data points which, however, need to be manually added to the training data to improve the system, thus retraining would be valuable but difficult to apply due to the process of labeling.

Test Input Generation
As previously mentioned, many tools use adversarial sample generation to discover faults in the system, such as DeepXplore [2], Surprise Adequacy [9], etc. It is often integral to discovering robustness faults and as such, we feel that it is important.

Cross-Referencing Model
Cross-referencing oracles are used within differential testing techniques, where the principle is, as stated in section 2.3, to feed two similar systems with the same input

and look for inconsistencies within the results of the two systems.

Output Validation
Once a testing tool completes the tests, it should be capable to generate the output report in any form for analysis of the system under test. Some of the tools presented test output on the console which were either internal to the testing tools or were difficult to interpret without additional support from the authors as there was no available documentation. The results were not intuitive to software engineers without extensive prior knowledge in deep learning.

Platform Support
The platform support requirement is related to the generality of the tool across different platforms. Some of the tools we tried to run did not support the Windows OS. This was due to plugin versions that were not supported by Windows.

Code Support
This requirement is quite peculiar, during our investigation we discovered that many of the proposed techniques were either inaccessible to us or we simply could not get them to work. If the framework made for the technique cannot be used or found, then it's unusable. Moreover, it was also not apparent what arguments should be used to run the tests. Without all of the arguments in place, the tool cannot execute properly or at all. A testing tool should be easy to use.

These are the nine most important requirements based on our research on DL testing tools. There are other possible requirements that are relevant to a testing tool but are either inapplicable to a benchmarking method or simply of low priority for a benchmarking effort. Some of the examples of such requirements are Application Stability, Latest Libraries Support, Cost, Ease of Developing and Maintaining the Scripts, Support to Web, Desktop & Mobile application, and Technical Support. However, due to time constraints and keeping the benchmarking method in mind, we will be focusing on only the most important requirements identified for DL testing tools in this study.

5.1.2 DL Testing Scenarios

With the requirements established and prioritized, the next step is synthesizing scenarios from the requirements. The synthesized ten scenarios are explained below:

Check DNN models selection capability
Does the tool provide the option to select your own model? For the DL testing tools to be applicable to real-world DL systems, model selection should be possible. As such this scenario is important for the flexibility and applicability of the tool. This concern arose as we attempted to use tools like DeepXplore [2] and SADL [9], which did not allow for model selection within their run arguments. Synthesized from Requirement 1: Technical Feasibility.

Check diversity in DL datasets test tasks
How many dataset classifications does the tool support? This scenario is aimed at covering whether the tool can still conduct its testing on a wider range of datasets. Most tools use images classification such as MNIST and CIFAR-10 datasets, but being able to only conduct testing on image datasets, excludes testing of other datasets like speech-to-text and autonomous driving. It is understandable why tools/frameworks focus on specific datasets instead of being generality applicable because the issue is not simply a matter of datasets, it is also a matter of DNNs. Different DNNs perform better in certain tasks that involve specific datasets. This is further supported by the fact that within the industry a known problem is DNN selection [10]. A prime example of such is how Recurrent Neural Networks (RNNs) are good at audio, natural languages, and video processing [13]. Synthesized from Requirement 2: Diverse Datasets.

Check adversarial input generation
Does the tool use adversarial input generation? Adversarial input generation is currently a widely used technique for testing how the system handles input that is similar to the training data but slightly distorted to ensure the *robustness* of the model. As such we feel that including adversarial inputs for testing improves the overall testing quality and should be included in nowadays DL testing tools if they are to be versatile. Additionally, this scenario is needed to answer Hypothesis 2. Synthesized from Requirement 5: Test Input Generation.

Check automatic labeling of test input
Does the tool automatically label test inputs? Several testing tools/frameworks are made for improving the test data and training data quality by identifying and retaining adversarial samples, which when included in the training/testing process would improve model performance. Manual labeling is labor-intensive and we feel that automated labeling of such data is an important step towards the future of data quality improvement. Synthesized from Requirement 4: Re-Training Models.

Check retraining of the DNN model under test
Does the tool support the retraining of the model after its tests? As an example, there are tools that find faults in the DL system and retain data that can be used for improving the model [9]. As such being able to retrain the model with the retained data and run the tests again would lead to better testing and towards an overall DL system improvement as mentioned in Section 2.2.2. Synthesized from Requirement 4: Re-Training Models.

Check DL system cross-referencing oracle support
Does the tool require more than one model to conduct its tests? Some DL testing tools may require a similar DL system like the one under test, by using the second system as a cross-referencing oracle [2]. A cross-referencing oracle is one of the solutions to the oracle problem stated in section 2.4, from which DL systems suffer. However, due to the complexity involved with DL systems, finding a similar system may prove to be difficult [3]. Nevertheless, if there is an available similar DL

system for evaluation, it can be beneficial for testing. For example, DeepXplore [2] was able to find multiple defects using a cross-referencing oracle. Synthesized from Requirement 6: Cross-Referencing Model.

Check test execution time
How long does a testing tool take to execute its tests? While some DL testing tools can show notable benefits when it comes to testing the system if the time the tool takes to conduct its tests outweighs the benefits it provides then that would be a flaw in the tool worth noting. Additionally, while there are cases of cross-comparison between tools to showcase tool efficiency in one form or another [3][15], the comparison is usually between the tool in question and at most, one other tool and execution time is sometimes involved. Synthesized from Requirement 3: Execution Time.

Check DL testing tool code availability
Currently, several tools, while possibly better than others, are either not publicly available or under an NDA (non-disclosure agreement). Synthesized from Requirement 9: Code Support.

Check testing output validation and readability
Does the tool save the results? Another integral part of a testing tool is saving the results for future references. While it may seem to be common sense to have such a functionality, many tools are still on a prototype level made for demonstrating a technique or framework where functionality as the saving of the results may not be present. Synthesized from Requirement 7: Output Validation.

Check platform support
Which platforms does the tool support? Although a tool may be excellent at testing a DL system, if the user cannot use the tool on that user's available platform, the applicability of the tool would lower. As such this scenario has an impact on the tool's generality. Synthesized from Requirement 8: Platform Support.

5.1.3 Benchmarking Tasks

Not all identified scenarios are feasible for a benchmarking effort. Table 5.2 shows the test scenarios that are technically feasible for benchmarking. The three scenarios are omitted for the following reasons:

- **Check adversarial input generation**: The test datasets generated by the testing tool is an internal implementation of the tool, hence cannot be evaluated using a benchmarking task.

- **Check automatic labeling of test input**: Similar to tests input generation, the labeling of test input is an internal implementation of the tool. It is, therefore, difficult for a benchmarking task to figure out whether a testing tool automatically labels individual inputs after it has conducted its tests.

- **Check DL testing tool code availability**: This scenario has been omitted because, for a tool to be benchmarked, the user of the benchmark must have the code already at hand.

Table 5.2: Test Scenarios for Benchmarking Design

Test Scenarios	Feasible for Benchmarking
Check DNN models selection flexibility	**Yes**
Check diversity in DL datasets test tasks	**Yes**
Check adversarial input generation	**NA**
Check automatic labeling of test input	**NA**
Check retraining of the DNN model under test	**Yes**
Check DL system cross-referencing oracle support	**Yes**
Check test execution time	**Yes**
Check DL testing tool code availability	**NA**
Check testing output validation and readability	**Yes**
Check platform support	**Yes**

Table 5.3 shows the list of benchmarking tasks that were made to address the test scenarios. The first column shows the benchmarking tasks, whereas the second column states the key questions to be answered for each task. Figure 5.1 shows the division of the seven benchmarking tasks. The detailed description of each task is done in the remaining of the section.

Table 5.3: Benchmarking Tasks and related key question to answer

Benchmarking Task	Key Questions
T1: Model Selection	Does the testing tool provide an option to test user-defined models? If yes, then the six models provided in the benchmark are used
T2: Diverse Dataset i) T2_1: Images Classification ii) T2_2: Self-Driving iii) T2_3: Text Classification	How many types of dataset classifications does the testing tool support? Does it support Images, Self-Driving, and Texts Classification?
T3: Retraining	Does the testing tool support the retraining of the model after its tests?
T4: Cross-referencing Oracle	Does the tool require more than one model to conduct its tests?
T5: Execution Time	How long does a testing tool take to execute its tests?
T6: Output Capabilities	Does the testing tool save the test results?
T7: Platform Support	Which platforms does the testing tool support?

T1: Model Selection

The model selection task is the most important task in the benchmark because the validity of other tasks within the benchmark depends on its evaluation. If the DL testing tool supports model selection, the six models (see subsection 3.2.4) provided in the benchmark will be used. This way measurements like execution time can be correctly measured. If the tool does not support model selection then the model provided with the tool is used and the tool is not ideal for benchmarking.

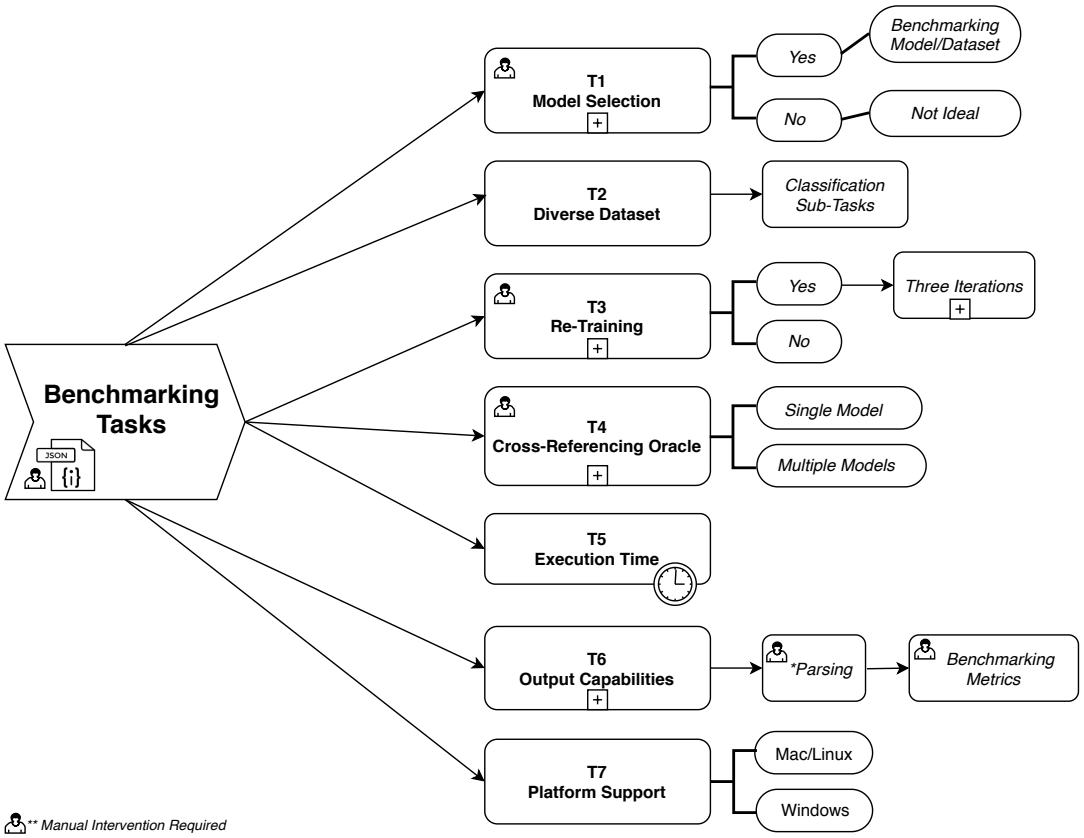

Figure 5.1: Benchmarking Method Tasks

T2: Diverse Dataset

DL systems use a variety of datasets classifications, i.e. MNIST and CIFAR-10 for image classification, Udacity for autonomous driving, Drebin for malware and IMDb, and Babi for texts, etc. As such, being able to handle tests of such a diverse variety of datasets is important for a DL testing tool. This task focuses on checking which datasets are supported by the tool to give a sense of its capabilities. The task is based on three types of dataset classifications as Images Classification (T2_1), Self-Driving (T2_2), and Text Classification (T2_3). As explained in Section 3.2.4, six different DNN models were defined to test each type of dataset classification.

T3: Retraining

Some DL testing tools are capable of retaining data that can be used for retraining the model and improving the model accuracy. If a tool supports retraining, the tool can be run any number of iterations for the user to see how the model has improved over time.

T4: Cross-referencing Oracle

The task provides information for whether the testing tool requires a single model or multiple models (differential testing) for running the tests.

T5: Execution Time
This task provides the total execution time taken by the tool under test, which is important for measuring the tool's *efficiency*.

T6: Output Capabilities
Every DL tool has a unique way of generation of output and it cannot be said in advance what measurement metric is displayed by a DL testing tool due to a lack of obvious performance measure standardization across the DL testing tools. DeepXplore [2] shows neuron coverage, SADL [9] on the other hand provides a ROC-AUC score. Also, not every tool is capable of generating output due to a lack of code support. Therefore, the benchmarking method would only check for the following two things as a part of this task:

- Are tools capable of giving any output in any format?
- Does it have a parser to give any measurement metrics? If yes, the benchmarking tool will just display it as an output.

This would give a metric that can not be compared with existing tools but would definitely help to differentiate with those tools which are not capable of saving any output for a system under test.

T7: Platform Support
This task provides information about the tool's support for different operating systems (Windows, macOS, or Linux). Many of the tools use plugin combinations like TensorFlow 1.5 with python 2.7, which is a combination that is currently not available for the Windows operating system. The results of this task have an important informative value for a practitioner. During our investigation, we came across tools that gave a wide variety of errors during compilation. This led to a loss of time until it was discovered that the framework/language version combination did not work on the operating system.

5.2 Benchmarking Design Results

The sections cover the design of the benchmarking tool and the results on four DL testing tools namely DeepXplore [2], SADL [9], DLFuzz [3] and DeepFault [18].

5.2.1 Benchmarking Tasks Automation

Out of the seven benchmarking tasks identified for the benchmarking method, not all of them can be fully automated. The main reason is that the DL testing tools and techniques presented in this paper are still maturing and don't have a stable working tool yet. Moreover, it would require a significant amount of time and effort to find an alternative solution to automate those tasks. The decision for manual and automation checks for each task is decided based on our experimental simulation of the selected fifteen tools in this paper. Table 5.4 shows the automation and manual check for each task.

Automated Tasks
Diverse dataset tasks can be automated based on provided execution command and task status of each sub-tasks (T2_1, T2_2, and T2_3). Tasks such as Execution Time and Platform Support can be fully automated. Output Capabilities task is just based on the criteria that all the benchmarking tasks are successfully executed and is able to save the results in any format for further analysis. The second aspect of this task is optional, i.e. parsing saved logs can also be automated based on the input command shared by a DL testing tool.

Manual Tasks
All three manual tasks are based on the capabilities of the DL testing tools and have a simple verdict as Yes/No, with default values as No. Such capabilities can be obtained either from the user manual or the code support web page of the DL testing tool. The verdict of each task is located in the last column of Table 5.4. For Model Selection, it is going to be manual as there is no straight forward way as of today for us to know if a tool is providing an option to test user-defined models. For instance, DeepXplore [2] uses its own fixed model. For the Retraining task, it cannot be evaluated automatically whether a testing tool is capable of retraining the model or not. Likewise, it can be known automatically if a tool requires more than one model to do its tests. Therefore, the Cross-referencing Oracle task is also manual.

The seven tasks presented in Table 5.4 are further categorized by priority as Very High, High, or low. The priority is based on their importance to the benchmarking method. Tools that do not pass the Very High priority tasks are not ideal for benchmarking.

Table 5.4: Benchmarking Tasks Automation Check

Task	**Automation/ Manual Check**	**Task Priority**	**Verdict**
Model Selection	Manual	Very High	Yes/No
Diverse Dataset	Automated	High	Yes/No
Retraining	Manual	Low	Yes/No
Cross-referencing Oracle	Manual	Low	Single Model Multiple Models
Execution Time	Automated	High	Time in seconds
Output Capabilities	Automated	High	Yes/No
Platform Support	Automated	Low	Windows Mac/Linux*

5.2.2 Benchmarking Tool

Figure 5.2 shows steps of the implemented benchmarking tool, involving all the seven tasks and sub-tasks as defined in the Figure 5.1. The six steps of the benchmarking process include:

1. Download benchmarking tool from Github
2. Fill out tool run and output configuration file
3. Generate DL testing tool run commands
4. Benchmarking tasks execution
5. Parsing (Optional)
6. Publish benchmarking tasks report

Implementation Details
The benchmarking tool is developed in Python (version 2.7) and the dependencies required to evaluate the four DL testing tools can be installed using the tool's execution options. The details of the benchmarking tool's usage and necessary requirements are described in the user manual page of the tool on GitHub [1].

Benchmarking Configuration File
Benchmarking run and output input is a *JSON* structured configuration file and is the most important step of the entire benchmarking method. It has all the required input fields needed by the benchmarking tool and is filled out based on the capabilities of a DL testing tool under test. The help file and an example of how to fill out the configuration file are available in Appendix section A.1 and subsection A.1.2.

Sub-Tasks and Benchmarking Models
The task T2 has three sub-tasks to test for support of three dataset classifications (Images, Self-driving, and Texts). DL testing tools are tested with pre-defined

[1] `DLTestBenchmark (https://github.com/hchuphal/DLTestBenchmark)`

benchmarking models for each type of dataset if the tool provides the option to select a model. There are six compiled models included in the benchmarking tool. The method used to build these DNN models is explained in methodology subsection 3.2.4. Details of a similar configuration file for DeepFault [18] are available in Appendix subsection A.1.2. The details of each model and its architecture are presented in Appendix A.8.

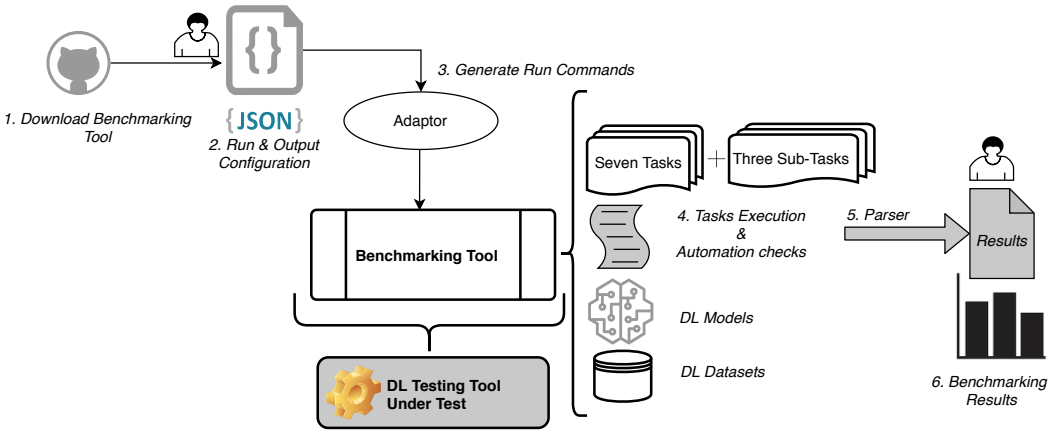

Figure 5.2: Benchmarking Method Process

5.2.3 Benchmarking Results

The results of the benchmarking method on four DL testing tools are presented in Table 5.5. The four testing tools used to demonstrate the benchmark are DeepXplore [2], SADL [9], DLFuzz [3] and DeepFault [18]. All of the 'Passed' tasks are marked with (✓), whereas 'Failed' tasks are marked with (✗). Out of the four, only DeepFault is ideal for benchmarking as it supports the model selection task. The benchmarking tasks report of every tool is also available in Appendix section A.4, section A.5, section A.6 and section A.7. All benchmarking tasks were executed on a Darwin Macintosh Laptop (an x86_64 machine and i386 processor) with a total of four physical cores (see Appendix section A.3). The source code of benchmarking tool is available on Github [2].

The comparison of the 'passed' and 'failed' tasks across the four tools is presented in Figure 5.3. The execution time (in seconds) taken by each tool is presented in Figure 5.4.

[2] `DLTestBenchmark (https://github.com/hchuphal/DLTestBenchmark)`

Table 5.5: Benchmarking Tool Results on four DL testing tool

		DeepXplore	SADL	DLFuzz	DeepFault
[1/9] task_1	Model Selection	✗	✗	✗	✓
[2/9] task_2_1	Image Classification	✓	✓	✓	✓
[3/9] task_2_2	Driving Classification	✓	✗	✗	✗
[4/9] task_2_3	Texts Classification	✓	✗	✗	✗
[5/9] task_3	Re-training	✓	✗	✓	✓
[6/9] task_4	Cross-referencing Oracle	✓	✗	✗	✗
[7/9] task_5	Execution Time (Seconds)	864.43	1108.52	579.86	177.96
[8/9] task_6	Output Capabilities	✗	✓	✓	✓
[9/9] task_7	Cross Platform Support	✓	✓	✓	✓
Benchmarking Verdict		Not Ideal	Not Ideal	Not Ideal	Ideal

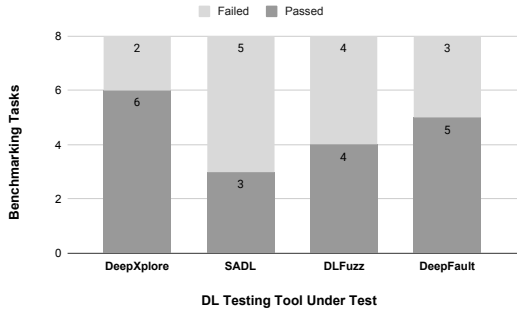

Figure 5.3: Tasks Status Comparison across four DL Testing Tools

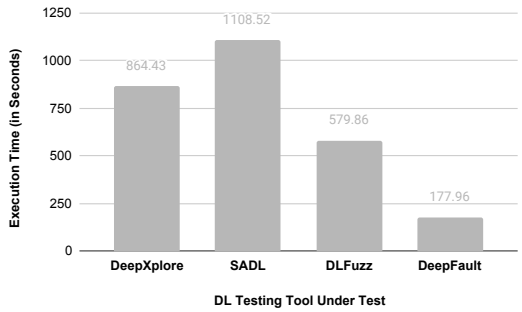

Figure 5.4: Execution Time Comparison across four DL Testing Tools

The output performance metrics of the four DL testing tools are presented in Table 5.6. As shown in the figure, some tools only output internal metrics and some output both external and internal metrics.

Table 5.6: Output Performance Metrics and type by DL Testing Tools

	DeepXplore	SADL	DLFuzz	DeepFault
Output Metric	Neuron Coverage	ROC-AUC Score & FGSM Coverage	Neuron Coverage & Time Consumption	Suspicious Neurons
Metric Type	Internal	External & Internal	Internal & External	Internal

6
Validation

The objective of this section is to confirm the validity of the benchmarking tool. This is done by conforming to the defined properties, which a successful benchmark needs to have, and by validating the necessary components of a benchmark as defined by Sim et al. [31]. Firstly, the feedback from the interview conducted with two industry contacts on the benchmarking tool is presented. Afterward, Section section 6.2 covers the arguments on the benchmark properties, which touches on the most questionable requirements/tasks and how they indeed conform to the benchmark properties. It followed by an argumentative validation on the high-level benchmarking components and validation arguments on manual task objectivity.

6.1 Interview Feedback

To validate the clarity property and objectivity of the manual tasks of the benchmarking method, we conducted a semi-structured interview session with two representatives of Chalmers IT-department.

Interview Setup:
The interview was conducted online and was recorded with the consent of the interviewees for future references. The interview session was divided into four parts: introduction, presentation, open-session with questions, and conclusions. During the introduction session, we introduced ourselves and got to know about the interviewees, as well as giving an overview of the interview. Afterward, a presentation was carried out to showcase the benchmarking method and the intention behind it. The presentation was followed by an open-session with questions aimed at both confirming the clarity and simplicity of the benchmarking method, and to find out whether the method could be helpful to practitioners to narrow down the selection of a DL testing tool. Finally, in the concluding session, we confirmed the answer we received during the open session, which was later used along with the recording of the interview session to summarize the answers to the questions.

Feedback:
To find out whether the manual tasks are sufficiently explained as to ensure objectivity and support the answer to **RQ3**, we asked the following three main questions:

- Are the steps of the benchmarking method simple to follow?
- Is the configuration file simple to understand and use?
- Would the benchmark method help the practitioners in selecting a DL testing tool?

The interviewees have 20 years of experience in the Chalmers IT department, with varied experience in software development, business process, AI, and Deep Learning. Both of the interviewees have also worked on different IT tools and software development cycles. We presented the benchmarking method, including a demo using the implemented benchmarking tool on the DeepFault [18] testing tool and got the following feedback:

- The interviewees found the steps very simple and straightforward to follow from the help page on GitHub [1], where the code of the benchmarking tool is hosted. However, there was confusion on the sequence of the steps while running the benchmark tool.

- Both of the interviewees found it very straightforward to fill out the output run configuration file in JSON format, particularly the manual tasks (having a *"Yes/No"* verdict). However, they provided suggestions to improve the overall format of the benchmark command section, which is part of the automated tasks.

- There were mixed answers regarding the usefulness of the benchmarking method to the practitioners. One of the interviewees said that the tool could definitely help in selecting or narrowing down the selection of a DL testing tool. However, the other interviewee was skeptical about the number of tasks that the benchmark tool should have. Because investing in a testing tool after selection is an expensive activity, there need to be more tasks, which can give further information on the tool before taking a decision on selecting an appropriate DL testing tool. They did not mention any exact number of 'more tasks' that should be included, but they said that the tasks should be more because selecting a testing tool is an investment.

They also liked the help section of the codebase on the GitHub page but suggested for more information about the tasks within the benchmarking tool. Both of the interviewees misunderstood what manual tasks are. They thought that manual tasks involved human judgment to evaluate the benchmarking tool's output. At the moment, only one tool runs at a time, they suggested running more than one tool in parallel or save the benchmarking results in a database for doing a good comparison across the DL testing tools under test. Finally, both of the interviewees gave positive feedback towards the output format of the benchmark results.

Based on the feedback, the information of the tasks was updated in the benchmarking tool and also in the GitHub help page. The important feedback for the execution

[1] `DLTestBenchmark (https://github.com/hchuphal/DLTestBenchmark)`

of tasks in parallel and saving the output in a database are considered in Future Work (see section 9.2).

6.2 Benchmarking Properties

There are seven properties of a successful benchmarking method identified by Sim et al. [31]. However, only five are relevant to this study: Relevance, Solvability, Scalability, Portability, and Clarity. The excluded two are Affordability and Accessibility. Affordability is not applicable due to the benchmark tool is publicly available, hence no cost is incurred in using the benchmark. Accessibility feels redundant because of a similar reason, the project is open-source and found on GitHub, whereas the abstract concept of the benchmark method is found in Chapter 5.

- **Relevance**: The task sample of the benchmark should consist of tasks that the system is expected to reasonably handle in a natural setting. Due to the requirements being elicited from the DL testing tools that the benchmark is meant to evaluate, we can ensure their relevance to the tools. Subsequently, the scenarios synthesized from those requirements become representative of real-world situations.

- **Scalability**: Scalability is another key benchmark property necessary for a good benchmark design [31]. Currently, with the quantifiable performance metric execution time and limited model selection capabilities of the tools, running the tool with different datasets is not as impactful as it will be in the future once other performance metrics which are influenced by the dataset type are added. A benchmark is a continuous effort and it should be treated as such during design. If it is not constantly improved upon, workarounds and exploits of the tasks can be designed or tasks may become outdated [31]. Currently, the method targets tools in a prototype and early state. However, the benchmarking method has six different DL models based on three datasets and seven tasks, which can easily be extended to include more models or tasks using the configuration file, if such a need arises.

- **Solvability**: In terms of task solvability, the main concern lies in the fact that many of the tools do not allow for model selection and use fixed models or models made through their own training program. This is due to the challenge of the field being in an early state. With the application of traditional techniques to DL, the tools developed are mainly in an early framework state designed to primarily showcase the technique. If the tool cannot use the models included in the benchmark, the benchmark would be compromised because performance measurements like execution time cannot be compared between the tools. This came to light early on as the first requirement due to being one of the biggest obstacles and the task to check whether the model selection is possible was made. While not as valuable for the industry, the benchmark information on tools with fixed models may still be valuable to the academic community. Additionally, this may inspire improvements in the tools to allow

model selection for them to be evaluated through the benchmark in the future.

- **Portability**: The next property to consider is Portability. The benchmarking tool is implemented in python, but the benchmarking method (as listed in Chapter 5) can be re-created in any programming language. The key challenge here is the platform. Although the benchmark design can be implemented for any platform, there is no guarantee that the tools can run on that platform. This is one of the reasons why Platform Support is a task within the benchmark, to provide information on what platform the tool can be used. Throughout the initial stages, while attempts were made to get the tools to work, many of them used plugin combinations that are not supported on Windows operating systems. This problem would inherently impact the benchmark as some of the tools would be unable to execute.

- **Clarity**: The final benchmark property to be covered is Clarity. It states that the benchmark specification should be clear to avoid mistakes or finding loopholes in the benchmark to change the results [31]. This property is particularly important for the study, due to the benchmark containing manual tasks on which mistakes could happen and would influence the results. Manual tasks are present within the benchmarking method because it simply is not worth the time and effort to automate tasks which can be done manually with no repercussions if the benchmark specification is done correctly in a clear and concise manner. This study itself serves as a specification in which the tasks and the idea behind them are covered. The manual tasks are encapsulated in a single and simple configuration file. Moreover, a help file was made to guide the user through the process of filling out the configuration file. The content of the configuration file can be found in Appendix A.1.2, whereas the content of the reference help file can be found in Appendix A.1. An interview was also conducted to validate the clarity property of the benchmark and the objectivity of the manual tasks. The interviewees found the configuration file simple and straightforward to fill out with the sole exception of the format of the commands which are necessary to run the tools. However, the commands to run the tools are dependent on the tools and not on the benchmark itself. Additionally, the commands are mainly concerned with automatic tasks rather than manual tasks. In regards to the sequence of the benchmark steps, there were no misunderstandings on what the steps are used for but the interviewees were unsure of whether the steps should be followed explicitly one after the other while running the benchmark.

6.3 High-level Benchmark Components

According to Sim et al. [31] a benchmark in software engineering consists of three components: a Motivation Comparison, Task Sample, and Performance Measures. Due to this study following his definition of a benchmark, our benchmarking method must conform to the three components. The Motivating Comparison component touches on the need for the benchmark, or more precisely the motivation behind

the comparison, which is covered in Section 1.2. The Task Sample is related to the relevance of the tasks of the benchmark. The tasks need to be representative of the tasks that the tool or technique is meant to handle in practice. This concern is handled by the method through the usage of real-world representative testing scenarios to formulate benchmarking tasks. The scenarios are constructed from tool requirements which were derived from different DL testing tools/techniques and testing research papers using established requirements elicitation techniques.

The final component is the Performance Measures component and it states that a benchmark should have quantitative or qualitative measurements of the tool's fitness for purpose, i.e. can it handle what it is expected to handle. Performance measures in regards to the improvement of the tool on a testing property were not feasible, thus leading to the decision to use user-orientated techniques, i.e. the requirements-scenarios-tasks model. As stated by Bai et al. [32] requirements provide a functional view of the system, whereas the scenarios provide a user's point of view. As pointed out by Sim et al. [31], a benchmark can have user interaction based tasks and the measurement can be qualitative. User interaction, in this case, does not mean the user interacting with the tasks of the benchmark, but how the software under test interacts with the user in the form of a task that has a qualitative result, i.e. a Yes/No verdict. In terms of quantitative metrics, at the moment only execution time is present within the benchmark.

6.4 Manual Tasks Objectivity

Manual task objectivity is about whether the tasks that need to be filled in manually in the configuration file are objective, i.e. measures have been taken to ensure that the tasks are filled correctly. The interview revealed that manual tasks can be misunderstood as a form of manual judgment of the output, however, that is not the case in this benchmarking method. The manual tasks are tasks that require information regarding the tool under test that cannot be known automatically by the benchmark. Therefore, to ensure manual task objectivity, measures were taken to ensure clarity and simplicity of the configuration file. First, the configuration file was filled out separately by both researchers on two DL testing tools that were not used in the testing of the benchmark, DeepHunter [24] and DeepConcolic [15]. This was done by referring to the configuration help file from section A.1 and the respective user manual document of the testing tool. Any differences in the filling out of the file were noted and discussed for what ambiguity caused them and a correction was applied in the context of the configuration help file. Finally, the benchmarking tool was executed using these two independently generated configuration files to check the verdict of all the manual tasks. The benchmarking results gave the same output for manual tasks. Moreover, all the manual tasks have a simple verdict as *"Yes/No"* (with default values as *'No'*) to be added in the configuration, which is pretty straight forward to follow. Such a snippet of the configuration file is shown in Figure 6.1.

Figure 6.1: An example snippet of the configuration file generated for the DeepFault testing tool with manual tasks marked as *"Yes"* or *"No"*, based on the capabilities of the testing tool.

```
{
   "toolName": "DeepFault",
   "description": "DeepFault: Fault Localization for Deep Neural
   ↪  Networks",
   "authors": "Hasan Ferit Eniser, Simos Gerasimou, Alper Sen",
   "language": "python",
   "publication": "2019",
   "..........."
   "..........."
   "manual_check": {
      "model_selection" : "Yes",
      "retraining" : "Yes",
      "differential_testing" : "Yes"
   },
   "datasets_classification": {
      "images" : "Yes",
      "self_driving" : "No",
      "texts" : "No"
   }
}
```

7
Discussion

In this section, the previously defined hypothesis and research questions will be discussed and answered. This will be done by using the results collected during the pre-study, benchmarking tool design, and execution process. Furthermore, benchmarking results are discussed.

7.1 Findings of the Study

Based on the pre-study results published in Table 5.5 and the benchmarking results presented in chapter 5, we can answer the research questions:

RQ1: Our investigation on DL testing tools/techniques resulted in fifteen testing tool research papers, out of which only four tools were in a good working state in our setup environment by following the user manual documents of the testing tools. This shows that although there is a growth in the number of open-source DL testing tools, many of them are not applicable for practical use. The reason for the failure of the DL testing tools is mainly because of failures in the interaction with the DL platform, i.e. TensorFlow, rather than in the execution of code logic. This is mostly due to the discrepancies between local and platform execution environments. Moreover, DL specific failures are also caused by inappropriate model parameters and obsolete framework API. The findings suggest that this is due to the early stage of the field, where the primary focus is improving DL systems by applying traditional and novel testing techniques and showcasing them within a tool that's in an early state. Efficiency was not a primary concern for the majority of the research paper, but there were cases where different DL testing tools were compared for the purpose of elevating one tool over another /citedlfuzz /citeconcolic in terms of efficiency. Moreover, the comparisons were made only with DeepXplore /citedeepxplore. Due to efficiency not being stated as a primary concern but used for comparison, it has not been added to the pre-study results. However, it is a part of the tasks of the benchmark method in the form of execution time measurement.

RQ2: The results of running the selected four DL testing tools on the designed benchmarking method suggest that the majority of the tools are currently not ideal for a benchmarking effort. The main reason for that is the lack of model selection support, without which the tools cannot be evaluated by using the models defined within the benchmarking tool. Furthermore, by addressing *Hypothesis 1*, it has been revealed that benchmarking a testing property is a major challenge. The

investigation done for **RQ1** unveiled that *robustness* is the most commonly tested dominant property. However, attempts to measure the improvement of the testing property were unsuccessful due to a lack of common performance metrics among the DL testing tools. As mentioned by Xie et al. [24], due to the early state of the field there is a lack of comparative studies on the effectiveness of the different techniques. This shows that it is difficult to agree upon a measurement that is effective. Without proof of effectiveness from comparative studies and the early state of the field, a surge of different metrics is caused to surface on the landscape of DL testing, i.e. Neuron Coverage, Mutation Score, Layer Coverage, etc. As a result, what is measured as a *property*, i.e robustness, reliability, correctness, etc., stays the same but the metrics used to quantify an improvement within the property differs. This results in a great challenge to benchmarking the effectiveness of the tools towards a given property, due to the obvious inability to be compared in terms of output. Currently, the most wide-spread measurement is the one proposed by DeepXplore [2] that is Neuron Coverage. However, Neuron Coverage is an internal metric to DeepXplore, similarly to the way FGSM coverage is internal to SADL [9]. It cannot be said that the tools calculate Neuron Coverage the same way.

Using the results of our benchmarking process to answer *Hypothesis 2* shows that we cannot know the input generated by a DL testing tool, therefore we cannot accept or reject the hypothesis. The reason for that is that every tool uses its own technique or library to generate the test inputs and are internal to the tool. The tools use a different sample of inputs for each test. Due to this, the calculations of measurements like an AUC-ROC score (See item 2 under pre-study results) cannot be made accurately for comparison, resulting in a similar problem of the inability to select a model.
It is important to note that these findings are based on the benchmarking results on four tools. The remaining tools that we did not manage to execute or other emerging DL testing tools, which were not covered in this paper, could potentially give different results. It is also not outside of expectations that these tools do not suffer from the issues defined above, which could potentially lead to different results.

RQ3: Our findings suggest that the proposed benchmarking method can be used by practitioners to narrow down the selection of a DL testing tool for a given dataset. Our benchmarking results show that only one out of four tools are suitable for benchmarking. The reason behind that is the tool's inability to select a user-defined model, which makes it also not suitable for practitioners. The results also reveal that only one tool supports more than one dataset classification, i.e. image classification. The benchmarking result will be helpful if the practitioner is looking for a tool that supports more that one dataset classification or a specific type of classification. Furthermore, tools like DeepXplore [2] use differential testing, which suffers from scalability and the difficulty of finding a similar system [3]. The remainder of the benchmarking tasks further helps in the selection process by narrowing down the scope through information on the tool's capabilities. The results are further supported by an interview with two industry researchers who, when asked, showed positive feedback towards the method, saying that it

could definitely help. However, there were also doubts about whether there are a sufficient number of tasks. According to the interviewees, selecting a testing tool is an endeavor that involves a significant investment, therefore having more tasks can further help to narrow down the selection of a DL testing tool.
The proposed benchmarking methodology can be used as an initial stepping stone in advancing the field of DL testing. Currently, diverse approaches are emerging within the field, which is an indicator that the bounds of the field are being established [31]. Therefore, a benchmarking methodology is necessary for the different approaches to be compared for a new research paradigm to be established. The second benefit is the results of the benchmark themselves, four of the tools have been compared and their differences have been noted. Although not as comprehensive of comparison as a comparative study would provide and rather limited in its evaluation of the tools, the data itself still holds value by providing information regarding the tools and how they compare against the others.

Although the benchmark can give guidance in tool selection, it cannot show which tool is better at improving a certain DL test property, i.e. robustness, correctness, reliability, etc. Additionally, due to three out of four tools not supporting model selection, and therefore not using a common model, a direct comparison of the execution time of each tool cannot be made. However, execution time is a valuable metric for the future of the benchmark, having been used in previous DL benchmark studies [29]. Once the limitation of model selection is resolved, future tools can be compared in terms of execution time and further metrics can be added. The final limitation in regards to **RQ3** is that the benchmarking method was not presented to industry experts who have experience with DL testing tools. A future study could present the method to experts with such experience and enhance the benchmarking method in accordance to their feedback. Nevertheless, valuable feedback for future improvements was gathered from the interview. A valid point was made, that if benchmarking suite needs to be executed on multiple DL testing tools, it would require a significant amount of time and effort to run them all through the benchmark one by one. A possible future improvement for the benchmarking tool can be running multiple tools in parallel and saving the benchmark results for each tool. An additional suggestion was to be able to upload the results to a database.

In conclusion, although the robustness is the predominant property of concern, a testing tool's robustness improvement of a DL system cannot be measured. This is a result of the four testing tools used within the study, however other tools could potentially lead to a different outcome. Nevertheless, by aiming at benchmarking the capabilities of the tools and tailoring the tasks to the current state of the tools, the results suggest that it could be used to narrow down the selection of a DL testing tool. However, there are suggestions to further improve the benchmarking method, for instance having more performance measurements and tasks, due to the costs involved in the selection of a DL testing tool.

7.2 Regarding DL Testing Tools

Out of the fifteen tools published in Table 4.2, we managed to get the code running only for four tools. As stated in the discussion for **RQ1**, the reasons for the failure of the DL testing tools execution is major because of failures in the interaction with the platform. For instance, reaching the code of DeepXplore [2] was fairly simple as the code is openly available and the support for the tool was good. However, DeepXplore runs on an old Tensorflow framework version (1.0) and Python 2.7. Using an older version of Tensorflow other than 2.0 with Python 2.7, which is now officially obsolete makes it incompatible to execute. DeepHunter [24] suffers from a similar fate. While running a higher version of Tensorflow (1.5), it still requires Python 2.7 and is incompatible with the Windows OS. The Coverage-guided fuzzing technique proposed in the paper by Xie et al. [23] is implemented in DeepHunter. DeepMutation [14] and DeepGauge [11] are under an NDA, therefore the code is currently unavailable. However, both of the tools use Keras (2.1.3) and Tensorflow (1.5). We were unable to get the code for DeepCruiser [27], DeepRoad [17], DeepStellar [13] and DiffChaser [25]. The mutation-inspired testing algorithm proposed by Wang et al. [26], nMutant, is implemented in its own tool that is unlike all of the other tools, an executable Java file. Although starting the application was simple, we were unable to test its effectiveness due to being unable to load a model to undergo the test. This shows that many of the DL testing tools are not in a complete working state but rather in a prototype state to showcase the proposed technique.

7.3 Regarding Benchmarking Results

As per benchmarking results published in Table 5.5 for four DL testing tools, the DeepFault [18] testing tool performed the best and passed all the Very High and High tasks and thus is considered as an ideal tool for benchmarking. Benchmarking results are purely based on the completion of both manual and automated nine tasks and sub-tasks. Accuracy of benchmarking tasks depends a lot on how accurately the tool's run and output configuration is filled out. If there are mistakes done in filling out the tool's configuration, which is used to execute a DL testing tool, mistakes would be reflected in the benchmarking results. Better benchmarking result doesn't necessarily mean that a DL testing tool is better than the other tools. As stated by Spendolini et al. [35], results do not come without effort and investment, and benchmarking is not a panacea for all issues. Benchmarking has been used as an improvement tool for many years, and the fundamental idea behind its use is simple. DL benchmarking results can only be regarded as a reference for selecting a more suited testing technique or tool for a given application scenario.

7.4 DL Testing Tools Recommendations

As mentioned in section 7.1, many of the tools are currently not ideal for benchmarking, due to a lack of essential or important functionality which is necessary

to be applicable to real-world scenarios and perform better in a benchmark. To address this issue, we propose a list of recommendations, based on our results, for the DL testing tool researchers/developers that develop DL testing tools. The list also provides recommendations on how to avoid obstacles and get the full use out of the benchmarking method.
Table 7.1 shows our recommendations for what DL tool researchers/developers should consider and what practitioners should do to narrow down the selection of a DL testing tool. Each recommendation is afterward explained as to what it means to the researchers/developers and what it means for general practitioners that are looking for a testing tool.

Table 7.1: List of recommendations to DL Tool Researchers and Practitioners

Recommendation	**DL Testing Tool Researchers**	**DL Practitioners**
Model Selection Option	✓	✓
Standardize Performance Metric	✓	✗
Output Readability	✓	✓
Tool Support	✓	✓
Diverse Datasets Classifications	✓	✗
Cross-platform Support	✓	✓
Support for Latest Libraries	✓	✗

1. **Model Selection Option:**
   Model Selection is a core feature for the applicability of the tool towards real-world scenarios. If practitioners cannot test their own models, then they cannot use the tool. Furthermore, the benchmarking method cannot compare the execution time of the tool with other tools, if a common model is not used.
   - **To DL testing tool researchers/developers**
     We recommend for DL testing tool researchers/developers to add a model selection capability to the tool to both be able to evaluate their own work with that of others using the benchmark and make the tool applicable for practitioners that want to test their own DL models.

   - **To DL practitioners**
     We recommend practitioners to try to understand the tool's options and if possible to change the code of the tool to make it accept an external model to test. The benchmark will then be able to use its compiled models and evaluate the testing tool.

2. **Standardize Performance Metric:**
   Currently, it is difficult to compare the tools when talking about an improvement of a DL model because the tools do not show the inputs that they use for the tests.
   - **To DL testing tool researchers/developers**

We recommend the inputs used during the tests to be saved and made available to the users to be able to make more accurate estimations and use common performance metrics.

- **To DL practitioners**
  Not applicable, as it touches the internal implementation of the DL testing tool.

3. **Output Readability:**
   A testing tool should be able to generate a readable output. Often, we had issues understanding the output of the tools. If practitioners cannot understand the output generated by the tool, then it is of little value to them.
   - **To DL testing tool researchers/researchers**
     We recommend researchers/developers to avoid using internal metrics in the output of the testing tools and put efforts to only generate output that can be easily understood.

   - **To DL practitioners**
     We recommend practitioners to contact the authors of the tools for further information if they cannot understand the generated output of the tool.

4. **Tool Support:** A testing tool is easy to use if it has good code support and an updated user manual.
   - **To DL testing tool researchers/researchers**
     We recommend researchers/developers to put required efforts to provide a good user manual of the testing tool including its execution steps and information about the dependencies needed to run the tool.

   - **To DL practitioners**
     Similar to Output Capabilities, we recommend practitioners to contact the authors if they need support with tool execution.

5. **Diverse Datasets Classifications:**
   Support for more than one DL dataset classifications.
   - **To DL testing tool researchers/developers**
     We recommend researchers/developers to include testing options for more than one DL dataset classifications, if applicable.

   - **To DL practitioners**
     Practitioners can refer the user manual of the tool to know supported DL dataset classifications, but nothing much can be done to include the support for an additional dataset classification as it touches the internal implementation of the DL testing tool.

6. **Cross-platform Support:**
   The tool should be executable on different operating systems.
   - **To DL testing tool researchers/developers**
     We recommend researchers/developers to make the testing tool available on different operating systems.

   - **To DL practitioners**
     There is no way for practitioners to make the tools available on every platform due to the platform unavailability being caused by old framework versions. However, most of the tools work on Linux and macOS operating systems, therefore we recommend practitioners to use those systems when benchmarking or using the tools.

7. **Support for Latest Libraries:**
   The tools should not use obsolete libraries, should be updated
   - **To DL testing tool researchers/developers**
     We recommend researchers/developers to keep the testing tool updated with latest supported libraries and get rid off obsolete libraries used by the tools.

   - **To DL practitioners**
     Not applicable, as it touches the internal implementation of the DL testing tool.

8

Threats to Validity

In this section, the threats to construct, internal, external, and reliability validity are discussed. In the following sections, each of these threats to validity for both the pre-study and the design of the benchmarking method, that was conducted in order to answer the research questions are explained.

8.1 Threats to Construct Validity

Construct validity focuses on whether the theoretical constructs are interpreted and measured correctly [36]. Threats to the construct validity can be benchmarking design errors, which could lead to the wrong phenomena being studied.

- Possible misunderstandings could happen if *'performance measures'* are interpreted in a different way. To mitigate this threat, this study used performance measures as defined by Sim et al. [31].

- Another threat is the term "DL testing properties". A possible misunderstanding may come with the term on whether it is the property of the DL system under test or the property of the testing tool itself. Throughout the study, we have used the term 'testing properties' as the property of the DL system under test. To mitigate this issue, clarifications have been made that with testing properties, we mean the property of a DL system, i.e. robustness, correctness, etc., that the tool tests and attempts to improve by finding defects related to the property.

- Having only one researcher reviewing the benchmarking method might have caused bias. To try and mitigate this risk, the issues regarding the method were also discussed among us and an interview was conducted with two experienced researchers from Chalmers IT department.

- The execution time measured for each DL testing tool is not on the same model. Hence, the time measurement in the results has little informative value. While unable to mitigate this issue due to only one tool allowing for model selection, the execution time task is still included for its future value to the benchmarking method.

- Manual tasks can be misunderstood with some tedious tasks which are not

objective. However, such tasks are just an input to the benchmark's configuration file with a verdict as *'Yes/No'* (see Appendix subsection A.1.2). Although this was not predicted as a potential problem, the interviews revealed that the word *'manual task'* can be misunderstood. To mitigate this issue, the help configuration file (see Appendix section A.1) and the user manual of the benchmarking tool was updated.

8.2 Threats to Internal Validity

Internal threats to validity involve factors, such as the design of the study, that could have affected the outcome [36].

- The models included in the current version of the benchmarking tool are trained on only three dataset classifications (i.e. images, self-driving, and texts). Other types of datasets classifications might be important to some researchers. However, we believe that the tests for the most popular datasets are included in the benchmark.

- The number of benchmarking tasks and their priority are entirely based on our study on fifteen DL testing tools. Some of the tasks might not make much of a sense to emerging testing tools in the near future. Nevertheless, we believe that based on our research, the seven tasks, identified in this study, can be further improved and extended in the future.

- The interviewees could not get hands-on experience with the configuration and help file due to a distant online interview and lack of time. However, they were able to answer all the questions without raising that as an issue.

8.3 Threats to External Validity

External validity focuses on whether claims for the generality of the results are justified [36]. The benchmarking of the DL testing tool should be both representative and generalizable.

- The interviewees have experience with Deep Learning and software tools in general but no experience per se with Deep Learning testing tools. Due to a lack of time and lack of industry contacts in the same field, we were unable to present the benchmarking tool to industry experts.

- The results of the benchmarking tasks heavily depend on the four tools used for our benchmarking evaluation. Different tools could potentially not suffer from the model selection issue or provide the inputs that they use during the tests. Due to a lack of code support, we were unable to make the remaining tools work in our local environment setup.

d

8.4 Threats to Reliability

Reliability focuses on whether the study yields the same results if other researchers replicate it [36].

- If the study is replicated in the future and the benchmarking tool is executed on the same DL testing tools, different results may be given if the tools have matured. For instance, the testing tools may provide an option for model selection in the near future.

- The research methodologies and the benchmarking steps are thoroughly described in this paper. By doing so, we believe this study can be reproduced by other researchers.

9
Conclusion and Future Work

This chapter contains two sections; the first section contains the conclusion, which includes the contribution of this thesis. The second section highlights potential future work for this research.

9.1 Conclusion

This thesis aims to explore the benchmarking methodology for DL Testing tools. In order to fulfill the goals of this thesis, we conducted pre-study research on fifteen DL testing tools followed by the design of the benchmarking methodology. During our pre-study, we studied existing state-of-the-art DL testing tools and techniques. Our pre-study aimed to explore DL testing workflow, components, and properties. We investigated the correctness, fairness, efficiency, and robustness properties of the existing DL testing tools to decide which testing tools are most suited for a relevant scenario. The pre-study results suggest that currently, the majority of the DL testing tools are focused on improving the robustness property of the DL systems. A requirement-scenario-task model was used to design the benchmarking method tasks for DL testing tools. After running the benchmarking method on selected four DL testing tools it became apparent that only one testing tool is ideal for benchmarking. This suggests that, even though there is an increase in DL testing tool research papers, the field is still in an early state that is not developed enough to run a full benchmarking suite. Although the fifteen DL testing techniques studied in this paper explain promising prospects to test DL systems, they are not sufficiently developed to be considered a complete testing tool yet. Due to this limitation, quantitative performance metrics that measure a tool's robustness testing ability on a DL system could not be established. However, the performance measures aimed at the capabilities of the tools, defined within the benchmark, are helpful for narrowing down the selection of a DL testing tool for practitioners. Moreover, a list of recommendations has been on what practitioners can do to further narrow down the DL testing tool selection.

9.2 Future Work

A benchmarking method is a continuous effort, hence scalability is an important aspect of its design. For future studies, we recommend a collaborative effort with the DL testing tool researchers to standardize the testing tools and improve the benchmark method accordingly. The importance of collaboration when conducting

a benchmark has already been pointed out by Sim et al. [31], and became even more evident through the data presented in this study. For instance, many of the DL testing tools do not support the model selection or output the test inputs used during the test. Such challenges can be overcome if the DL testing tool researchers are involved in the benchmarking effort. By standardizing such necessary components needed to perform a benchmark, the testing tools can be better cross-compared and we believe that it could further advance the field. Moreover, a list of recommendations has been presented in this study to DL testing tool researchers/developers for future research that develop DL testing tools. As the DL testing techniques are in the early prototype stage, we also recommend further investigation of the DL testing tools as they evolve towards a complete testing tool in the future. A possible future improvement that will need to be made is the addition of more, common quantifiable performance metrics, which evaluate the tools in terms of improvement towards DL testing properties (see Section 2.2.4). Regarding the formation of the benchmarking tasks, it is fine to have manual tasks in benchmarking as long as they are objective. However, a future investigation can be made to avoid manual interventions in the benchmarking process. Finally, we recommend investigations to further enhance the benchmarking method by considering the execution of multiple DL testing tools in parallel and saving the results in a database for future reference.

Bibliography

[1] Myers, G.J., Sandler, C. and Badgett, T., 2011. *The art of software testing.* John Wiley Sons.

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

[3] Guo, J., Jiang, Y., Zhao, Y., Chen, Q. and Sun, J., 2018, October. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering.* (pp. 739-743).

[4] McKeeman, W.M., 1998. Differential testing for software *Digital Technical Journal, 10*(1), pp.100-107.

[5] Chen, T.Y., Cheung, S.C. and Yiu, S.M., 2020. Metamorphic testing: a new approach for generating next test cases. *arXiv preprint arXiv:2002.12543.*

[6] Barr, E.T., Harman, M., McMinn, P., Shahbaz, M. and Yoo, S., 2014. The oracle problem in software testing: A survey. *IEEE transactions on software engineering, 41*(5), pp.507-525.

[7] Jia, Y. and Harman, M., 2010. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering, 37*(5), pp.649-678.

[8] Zhu, H., Hall, P.A. and May, J.H., 1997. Software unit test coverage and adequacy. *Acm computing surveys (csur), 29*(4), pp.366-427.

[9] Kim, J., Feldt, R. and Yoo, S., 2019, May. Guiding deep learning system testing using surprise adequacy. In *IEEE/ACM 41st International Conference on Software Engineering (ICSE)* (pp. 1039-1049). IEEE.

[10] Jahangirova, G., Humbatova, N., Bavota, G., Riccio, V.,Stocco, A. and Tonella, P., 2019. Taxonomy of Real Faults in Deep Learning Systems. *arXiv preprint arXiv:1910.11015.*

[11] Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., Xue, M., Li, B., Chen, C., Su, T., Li, L., Liu, Y. and Zhao, J., 2018, September. Deepgauge: Multi-granularity testing criteria for deep learning systems. *In Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (pp. 120-131).

[12] Zhang, T., Gao, C., Ma, L., Lyu, M. and Kim, M., 2019, October. An empirical study of common challenges in developing deep learning applications. In *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)* (pp. 104-115). IEEE.

[13] Du, X., Xie, X., Li, Y., Ma, L., Liu, Y. and Zhao, J., 2019, August. Deepstellar: model-based quantitative analysis of stateful deep learning systems. In *Proceed-*

ings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (pp. 477-487).

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

[15] Sun, Y., Wu, M., Ruan, W., Huang, X., Kwiatkowska, M. and Kroening, D., 2018, September. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (pp. 109-119), https://github.com/TrustAI/DeepConcolic.

[16] Tian, Y., Pei, K., Jana, S. and Ray, B., 2018, May. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering* (pp. 303-314).

[17] Zhang, M., Zhang, Y., Zhang, L., Liu, C. and Khurshid, S., 2018, September. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (pp. 132-142).

[18] Eniser, H.F., Gerasimou, S. and Sen, A., 2019, April. Deepfault: Fault localization for deep neural networks. In *International Conference on Fundamental Approaches to Software Engineering* (pp. 171-191). Springer, Cham, https://github.com/hasanferit/DeepFault.

[19] Kuhn, D.R., Kacker, R.N. and Lei, Y., 2010. Practical combinatorial testing. *NIST special Publication, 800*(142), p.142.

[20] Nie, C. and Leung, H., 2011. A survey of combinatorial testing. *ACM Computing Surveys (CSUR), 43*(2), pp.1-29.

[21] Ma, L., Zhang, F., Xue, M., Li, B., Liu, Y., Zhao, J. and Wang, Y., 2018. Combinatorial testing for deep learning systems. *arXiv preprint arXiv:1806.07723.*

[22] Klees, G., Ruef, A., Cooper, B., Wei, S. and Hicks, M., 2018, January. Evaluating fuzz testing. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security* (pp. 2123-2138).

[23] Xie, X., Ma, L., Juefei-Xu, F., Chen, H., Xue, M., Li, B., Liu, Y., Zhao, J., Yin, J. and See, S., 2018. Coverage-guided fuzzing for deep neural networks. *arXiv preprint arXiv:1809.01266, 3.*

[24] Xie, X., Ma, L., Juefei-Xu, F., Xue, M., Chen, H., Liu, Y., Zhao, J., Li, B., Yin, J. and See, S., 2019, July. DeepHunter: a coverage-guided fuzz testing framework for deep neural networks. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (pp. 146-157).

[25] Xie, X., Ma, L., Wang, H., Li, Y., Liu, Y. and Li, X., 2019, August. Diffchaser: Detecting disagreements for deep neural networks. In *Proceedings of the 28th International Joint Conference on Artificial Intelligence* (pp. 5772-5778). AAAI Press.

[26] Wang, J., Sun, J., Zhang, P. and Wang, X., 2018. Detecting adversarial samples for deep neural networks through mutation testing. *arXiv preprint arXiv:1805.05010.*

[27] Du, X., Xie, X., Li, Y., Ma, L., Zhao, J. and Liu, Y., 2018. Deepcruiser:

Automated guided testing for stateful deep learning systems. *arXiv preprint arXiv:1812.05339.*

[28] Sun, Y., Huang, X., Kroening, D., Sharp, J., Hill, M. and Ashmore, R., 2018. Testing deep neural networks. *arXiv preprint arXiv:1803.04792.*

[29] Shi, S., Wang, Q., Xu, P. and Chu, X., 2016, November. Benchmarking state-of-the-art deep learning software tools. In *2016 7th International Conference on Cloud Computing and Big Data (CCBD)* (pp. 99-104). IEEE.

[30] Zhang, J.M., Harman, M., Ma, L. and Liu, Y., 2020. Machine learning testing: Survey, landscapes and horizons. *IEEE Transactions on Software Engineering.*

[31] Sim, S.E., Easterbrook, S. and Holt, R.C., 2003, May. Using benchmarking to advance research: A challenge to software engineering. In *25th International Conference on Software Engineering, 2003. Proceedings.* (pp. 74-83). IEEE.

[32] Bai, X., Tsai, W.T., Paul, R., Feng, K. and Yu, L., 2002, January. Scenario-based modeling and its applications. In *Proceedings of the Seventh IEEE International Workshop on Object-Oriented Real-Time Dependable Systems.(WORDS 2002)* (pp. 253-260). IEEE.

[33] Sharma, S. and Pandey, S.K., 2013. Revisiting requirements elicitation techniques. *International Journal of Computer Applications*, 75(12).

[34] Beyer, D., Löwe, S. and Wendler, P., 2019. Reliable benchmarking: Requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21(1), pp.1-29.

[35] Stapenhurst, T., 2009. *The benchmarking book*. Routledge.

[36] Easterbrook, S., Singer, J., Storey, M.A. and Damian, D., 2008. Selecting empirical methods for software engineering research. In *Guide to advanced empirical software engineering* (pp. 285-311). Springer, London.

[37] Knuth: Computers and Typesetting,
`http://www-cs-faculty.stanford.edu/~uno/abcde.html`

[38] Keras: The Python Deep Learning library, 2020
`https://keras.io`

[39] An end-to-end open source machine learning platform, 2020
`https://www.tensorflow.org`

[40] Caffe : Deep learning framework, 2020
`https://caffe.berkeleyvision.org/`

[41] scikit-learn : Machine Learning in Python, 2020
`https://scikit-learn.org`

[42] Python : Programming language, 2020
`https://www.python.org/`

A

Appendix 1

A.1 Benchmarking Run Configuration JSON File Structure

A.1.1 Run and Output Configuration with 'help' Text

```
// Please follow // comments for each json parameter to fill in the run and output config

{
   "toolName": "",               // DL Testing Tool name
   "description": "",            // description of the tool
   "authors": "",                // Authors name
   "language": "pass",               //**Tools run and implementation programming lanaguage used
   "publication": "",
   "path_to_script" : "",
   "commands": [                 // Total commands in list, can be extented if more than 7 run commmadns are required
      { "path_1":"pass",         // path of the locations of the script /tool to run the command 1
      "command_1": "pass",       // single command for same type of dataset or a different dataset that tool should run for benchmakring
      "dataset_type": "pass"},   // type of dataset for command 1 ( images, self_driving, texts)
      {"path_2":"pass",          // and so on .. as done for command 1 else leave it witgh "pass"
      "command_2": "pass",
      "dataset_type": "pass"},
      {"path_3":"pass",
      "command_3": "pass",
      "dataset_type": "pass"},
      {"path_4":"pass",
      "command_4": "pass",
      "dataset_type": "pass"},
      {"path_5":"pass",
      "command_5": "pass",
      "dataset_type": "pass"},
```

```
26        {"path_6":"pass",
27        "command_6": "pass",
28        "dataset_type": "pass"},
29        {"path_7":"pass",
30        "command_7": "pass",
31        "dataset_type": "pass"}      // extend list if more commands are
          ↪  supported
32    ],
33    "manual_check": {                // manual check based in Tools support
      ↪  and capabilities
34        "model_selection" : "No",     // Default Value is No, put as yes if
          ↪  model selection is suuported
35        "retraining" : "No",          // Default Value is No, put as yes if
          ↪  retraining is suuported
36        "differential_testing" : "No"// Default Value is No, put as yes if
          ↪  differential testing (testing with more than 1 model) is
          ↪  suuported
37    },
38    // if model_selection is possible (Yes) then fill out all run command
      ↪  as in previous commands but with benechmarking models based on type
      ↪  pf datasets
39    // moded 1 (images dataset) :
      ↪  ./model/Images_cifar10_1_512_leaky_relu_model1.h5
40    // model 2 (self_driving dataset): ./model/Self_Driving_CNN_model1.h5
41    // model 3 (texts dataset):   ./model/Text_babi_RNN_model1.h5
42    "benchmarking_commands": [       // commands to run benchmakring models
      ↪  by replacing the path of one of models 1, 2, 3 based on dataset
43        { "path_1":"pass",            // follow the same sequence as before
          ↪  but with benchmakring models
44        "command_1": "pass",          // command using benchmarking model
45        "dataset_type": "pass"},      // dataset type supported by
          ↪  benchmarking model
46        {"path_2":"pass",             // and so on .. as done for command 1
          ↪  else leave it witgh "pass"
47        "command_2": "pass",
48        "dataset_type": "pass"},
49        { "path_3":"pass",
50        "command_3": "pass",
51        "dataset_type": "pass"},
52        {"path_4":"pass",
53        "command_4": "pass",
54        "dataset_type": "pass"}       // can be extented if more variaions
          ↪  are needed to run
55    ],
56
57    "datasets_classification": {  // manual check based in Tools support
      ↪  and capabilities for different dataset classification
58        "images" : "No",           // Default Value is No, put as yes if
          ↪  images classifications is suuported
```

```
59        "self_driving" : "No",      // Default Value is No, put as yes if
          ↪  self driving  classifications is suuported
60        "texts" : "No"              // Default Value is No, put as yes if
          ↪  texts classifications is suuported
61     },
62     "output_config" : {              // manual check based in Tools support
       ↪  and capabilities for different dataset classification
63        "output_saved" : "No",        // Default Value is No, put as yes if
          ↪  tool generates any form of output either on console or saved in
          ↪  a file
64        "output_default_path" : "./output/",
65        "postProcessingCommand" : "None",   // Default Value is None, put as
          ↪  yes if postprocessing of the generated output is supported
66        "parser_path" : "_path_"            // If postProcessingCommand is
          ↪  yes, path to the parser tool or script
67     }
68  }
```

A.1.2 Example of Configuration file of a DL Testing Tool

Run Configration file filled out for DeepFault DL Testing tool which was used for benchmakring tests.

```
1  {
2     "toolName": "DeepFault",
3     "description": "DeepFault: Fault Localization for Deep Neural
      ↪  Networks",
4     "authors": "Hasan Ferit Eniser, Simos Gerasimou, Alper Sen",
5     "language": "python",
6     "publication": "2019",
7     "path_to_script" :
      ↪  "/Users/hchuphal/Desktop/github/thesis2020/Code/DeepFault-master",
8     "commands": [
9        {
         ↪  "path_1":"/Users/hchuphal/Desktop/github/thesis2020/Code/DeepFault-master",
10       "command_1": "python2.7 run.py --model mnist_test_model_1_100
         ↪  --dataset mnist -C 9 --approach tarantula --suspicious_num 10",
11       "dataset_type": "images"},
12
         ↪  {"path_2":"/Users/hchuphal/Desktop/github/thesis2020/Code/DeepFault-master",
13       "command_2": "python2.7 run.py --model
         ↪  cifar10_test_model_1_512_leaky_relu --dataset cifar10 -C 9
         ↪  --approach tarantula --suspicious_num 10",
14       "dataset_type": "images"},
15       {"path_3":"pass",
16       "command_3": "pass",
17       "dataset_type": "pass"},
18       {"path_4":"pass",
19       "command_4": "pass",
20       "dataset_type": "pass"},
```

```
        {"path_5":"pass",
        "command_5": "pass",
        "dataset_type": "pass"},
        {"path_6":"pass",
        "command_6": "pass",
        "dataset_type": "pass"},
        {"path_7":"pass",
        "command_7": "pass",
        "dataset_type": "pass"}
    ],
        "benchmarking_commands": [
        {
         ↪  "path_1":"/Users/hchuphal/Desktop/github/thesis2020/Code/DeepFault-master",
        "command_1": "python2.7 run.py --model
         ↪  Images_cifar10_1_512_leaky_relu_model1 --dataset cifar10 -C 9
         ↪  --approach tarantula --suspicious_num 10",
        "dataset_type": "images"},

         ↪  {"path_2":"/Users/hchuphal/Desktop/github/thesis2020/Code/DeepFault-master",
        "command_2": "python2.7 run.py --model Self_Driving_CNN_model1
         ↪  --dataset nvidia -C 9 --approach tarantula --suspicious_num 10",
        "dataset_type": "self_driving"},
        {
         ↪  "path_3":"/Users/hchuphal/Desktop/github/thesis2020/Code/DeepFault-master",
        "command_3": "python2.7 run.py --model Text_imdb_CNN_model2
         ↪  --dataset imdb -C 9 --approach tarantula --suspicious_num 10",
        "dataset_type": "texts"},
        {"path_4":"pass",
        "command_4": "pass",
        "dataset_type": "images"}
    ],
    "manual_check": {
        "model_selection" : "Yes",
        "retraining" : "Yes",
        "differential_testing" : "Yes"
    },
    "datasets_classification": {
        "images" : "Yes",
        "self_driving" : "No",
        "texts" : "No"
    },
    "output_config" : {
        "output_saved" : "Yes",
        "output_default_path" : "./output/",
        "postProcessingCommand" : "None",
        "parser_path" : "_path_"
    }
}
```

A.2 Python Script component used for executing Benchmarking Tasks

```
# Script Component of benchmarking tasks
class BenchmarkingTasks(unittest.TestCase):
    """ Results of 6 Tasks and 3 SubTasks"""
    with open(_TEMP_CONFIG, 'r') as myfile:
        json_data=myfile.read()
    parsed_json = (json.loads(json_data))
    obj = json.loads(json_data)
    language = str(obj['language'])
    commmands_list = obj['commands']
    manual_check = obj['manual_check']
    output_config = obj['output_config']
    language = obj['language']
    datasets = obj['datasets_classification']

    def test_Model_Selection(self):
        time.sleep(1)
        assert self.manual_check['model_selection']  in self._pass, "Model
        ↪  Selection is not possible"

    def test_Image_Classifications_Support(self):
        time.sleep(1)
        assert self.datasets['images']  in self._pass, "Image
        ↪  Classifications are not possible"

    def test_SelfDriving_Classifications_Support(self):
        time.sleep(1)
        assert self.datasets['self_driving']  in self._pass, "Self_driving
        ↪  datasets are not possible"

    def test_Texts_Classifications_Support(self):
        time.sleep(1)
        assert self.datasets['texts']  in self._pass, "Texts/Malware
        ↪  datasets are not possible"

    def test_Retraining(self):
        time.sleep(1)
        assert self.manual_check['retraining']  in self._pass,"Retraining
        ↪  is not possible"

    def test_Differential_Testing(self):
        time.sleep(1)
        assert self.manual_check['differential_testing'] in
        ↪  self._pass,"Differential Testing is not possible"

    def test_Execution_Time(self):
```

```
40          time.sleep(1)
41          assert self._time > 1.0 ,"Execution time of Testing is less than 1
            ↪  second"
42          #logger.info("\n Total time taken in ms : " + str(self._time))
43
44      def test_Output_Capabilities(self):
45          time.sleep(1)
46          assert self.output_config['output_saved'] in self._pass,"Output
            ↪  Saving is NOT possible"
47          #logger.info(self._command_status)
48          for i, status in enumerate(self._command_status):
49              if self._command_status[i] != 0:
50                  assert self._command_status[i] == 0, "DL Testing Tool
                    ↪  command failed to execute!"
51
52  def _write_output(_buffer):
53      logger.info(_buffer)
54      try:
55          # log file to write to
56          logFile =
            ↪  _OUTPUT+'Benchmarking_dl_testing_tool_'+time.strftime("%Y%m%d-%H%M%S")+'_._log
57          report = open(logFile, 'a')
58          report.write(_buffer)
59
60      except Exception as e:
61          # get line number and error message
62          report.write('En error message while Executing DL Testing command'
            ↪  + e+ logFile)
63
64  if __name__ == '__main__':
65      # 1. get run config
66      _make_runconfig()
67      try:
68          shutil.rmtree(_OUTPUT)
69          os.mkdir(_OUTPUT)
70      except OSError as e:
71          logger.warning("Error: No Previous Output found! %s - %s." %
            ↪  (e.filename, e.strerror))
72      #sys.stdout =
        ↪  open('Benchmarking_logs_'+time.strftime("%Y%m%d-%H%M%S"+'_.log'),
        ↪  'w')
73      # 2. Read the run config
74      if _TEMP_CONFIG:
75          with open(_TEMP_CONFIG, 'r') as myfile:
76              json_data=myfile.read()
77      parsed_json = (json.loads(json_data))
78      #print(json.dumps(parsed_json, indent=4, sort_keys=True))
79      obj = json.loads(json_data)
80
```

```
81      language = str(obj['language'])
82      commmands_list = obj['commands']
83      manual_check = obj['manual_check']
84      output_config = obj['output_config']
85      language = obj['language']
86      datasets = obj['datasets_classification']
87      # 3. Run each command specified in the run configuration of the DL
        ↪  testing tool
88      spath = str(obj['path_to_script'])
89      _total_commands = [command for command in commmands_list if
        ↪  command["dataset_type"] != 'pass']
90      logger.info("\n")
91      logger.info ("\n********* Tasks Execution Started ..... *********")
92      logger.info('Total ' + str(len(_total_commands))+ ' commands to
        ↪  execute for Benchmarking!\n\n')
93      #_buffer = []
94      bar = progressbar.ProgressBar(maxval=9, \
95      widgets=[progressbar.Bar('#', 'Progress ....[', ']'), ' ',
        ↪  progressbar.Percentage()])
96      bar.start()
97      start = time.time()
98      for i, commands in enumerate(commmands_list):
99          for command, argument  in commands.items():
100             if argument != 'pass':
101                 logger.info('Executing DL Testing tool ' +'run ' + command
                    ↪  +' : '+ argument)
102                 if 'dataset_type' in ['images', 'texts', 'self_driving']:
103                     logger.info('Dataset Classifications'+ dataset_type)
104                     # change to working directory of the script
105                 os.chdir(commands["path_"+str(i+1)])
106                 if "python" in argument:
107                     with CodeTimer(' Time to run the  testing command :'):
108                         try:
109                             _status = os.system(argument)
110                             _buffer.append(_status)
111                             #_buffer = subprocess.check_output(argument)
112                             #_write_output(_buffer)
113                         except Exception as e:
114                             logger.error("Benchmarking DL Testing Tool
                                ↪  command failed!")
115
116                     #returned_output = subprocess.check_output('python
                        ↪  gen_diff.py light 1 0.1 10 20 50 0')
117     final_time = (time.time() - start)
118     bar.finish()
119     print("\n")
120     logger.info("Total Execution Time taken to run all the commands :"
        ↪  +str(final_time) +' Seconds')
121     parser = argparse.ArgumentParser()
```

```
    final_time, _buffer]
    test_suite = unittest.TestSuite()

    repetitions = 1  # how many times to we want to repeat the tasks (7)
    tasks = get_tests()
    for __ in xrange(0, repetitions):
        test_suite.addTests(tasks)
    logger.info("\nExecuting Benchmakring Tasks one by one....")
    time.sleep(1)
    TestRunner.main()
    runner.run(test_suite)
    final_time_2 = (time.time() - start)
    logger.info("Total Execution Time taken by Benchmakring Tool :"
    ↪  +str(final_time_2) +' Seconds')
```

A.3 System Configuration for DL Testing Tools

```
# System Information
System: Darwin and Release: 19.3.0
Version: Darwin Kernel Version 19.3.0: Thu Jan  9 20:58:23 PST 2020;
↪  root:xnu-6153.81.5~1/RELEASE_X86_64
Machine: x86_64 and Processor: i386

# CPU and Memory Information
Physical cores: 2 and Total cores: 4
Max Frequency: 3100.00Mhz
Min Frequency: 3100.00Mhz
Current Frequency: 3100.00Mhz
svmem(total=8589934592, available=2073575424, percent=75.9,
↪  used=4944916480, free=143818752, active=1930104832,
↪  inactive=1841102848, wired=3014811648)
System Memory % used: 75.9
```

A.4 Benchmarking Tasks Results for DeepXplore Tool

```
2020-04-25 02:43:42,515 - root - INFO - Total Execution Time taken to run all the commands :864.438571215 Seconds
2020-04-25 02:43:42,520 - root - INFO -
Executing Benchmakring Tasks one by one....

 ********** DL Testing Tool Benchmarking Report **********

Status:
    Tasks Passed: 7
    Tasks Failed: 2

Summary of Benchmarking Tasks Execution:
    +-----------------------------------------+-------+------+------+-------+
    |          Task group/Sub-Task            | Count | Pass | Fail | Error |
    +-----------------------------------------+-------+------+------+-------+
    | BenchmarkingTasks:  Results of 7 Tasks | 9     | 7    | 2    | 0     |
    | Total                                   | 9     | 7    | 2    | 0     |
    +-----------------------------------------+-------+------+------+-------+
    BenchmarkingTasks
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------+-------------+
    |                Task name                |                                                Failure Message/Info                                                   | Task Status |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------+-------------+
    | test_Differential_Testing               |                                                                                                                       | pass        |
    | test_Execution_Time                     | 864.44 Seconds                                                                                                        | pass        |
    | test_Image_Classifications_Support      |                                                                                                                       | pass        |
    | test_Model_Selection                    | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 148, in test_Model_Selection                                                     |             |
    |                                         |     assert self.manual_check['model_selection']  in self._pass, "Model Selection is not possible"                    |             |
    |                                         | AssertionError: Model Selection is not possible                                                                       |             |
    | test_OS_Support                         | Darwin-19.3.0-x86_64-i386-64bit                                                                                       | pass        |
    | test_Output_Capabilities                | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 201, in test_Output_Capabilities                                                 |             |
    |                                         |     assert self._command_status[i] == 0, "DL Testing Tool command failed to execute!"                                 |             |
    |                                         | AssertionError: DL Testing Tool command failed to execute!                                                            |             |
    | test_Retraining                         |                                                                                                                       | pass        |
    | test_SelfDriving_Classifications_Support |                                                                                                                       | pass        |
    | test_Texts_Classifications_Support      |                                                                                                                       | pass        |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------+-------------+
```

Figure A.1: DeepXplore : Benchmarking Tasks Results

A.5 Benchmarking Tasks Results for SADL Tool

```
2020-04-25 03:08:41,180 - root - INFO - Total Execution Time taken to run all the commands :1108.52183914 Seconds
2020-04-25 03:08:41,186 - root - INFO -
Executing Benchmakring Tasks one by one....

 ********** DL Testing Tool Benchmarking Report **********

Status:
    Tasks Passed: 4
    Tasks Failed: 5

Summary of Benchmarking Tasks Execution:
    +-----------------------------------------+-------+------+------+-------+
    |          Task group/Sub-Task            | Count | Pass | Fail | Error |
    +-----------------------------------------+-------+------+------+-------+
    | BenchmarkingTasks:  Results of 7 Tasks | 9     | 4    | 5    | 0     |
    | Total                                   | 9     | 4    | 5    | 0     |
    +-----------------------------------------+-------+------+------+-------+
    BenchmarkingTasks
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------+-------------+
    |                Task name                |                                                Failure Message/Info                                                   | Task Status |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------+-------------+
    | test_Differential_Testing               | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 186, in test_Differential_Testing                                                |             |
    |                                         |     assert self.manual_check['differential_testing'] in self._pass,"Differential Testing is not possible"            |             |
    |                                         | AssertionError: Differential Testing is not possible                                                                  |             |
    | test_Execution_Time                     | 1108.52 Seconds                                                                                                       | pass        |
    | test_Image_Classifications_Support      |                                                                                                                       | pass        |
    | test_Model_Selection                    | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 148, in test_Model_Selection                                                     |             |
    |                                         |     assert self.manual_check['model_selection']  in self._pass, "Model Selection is not possible"                    |             |
    |                                         | AssertionError: Model Selection is not possible                                                                       |             |
    | test_OS_Support                         | Darwin-19.3.0-x86_64-i386-64bit                                                                                       | pass        |
    | test_Output_Capabilities                |                                                                                                                       | pass        |
    | test_Retraining                         | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 182, in test_Retraining                                                          |             |
    |                                         |     assert self.manual_check['retraining']  in self._pass,"Retraining is not possible"                                |             |
    |                                         | AssertionError: Retraining is not possible                                                                            |             |
    | test_SelfDriving_Classifications_Support | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 162, in test_SelfDriving_Classifications_Support                                 |             |
    |                                         |     assert self.datasets['self_driving']  in self._pass, "Self_driving datasets are not possible"                    |             |
    |                                         | AssertionError: Self_driving datasets are not possible                                                                |             |
    | test_Texts_Classifications_Support      | Traceback (most recent call last):                                                                                    | fail        |
    |                                         |   File "benchmarking_tasks.py", line 172, in test_Texts_Classifications_Support                                       |             |
    |                                         |     assert self.datasets['texts']  in self._pass, "Texts/Malware datasets are not possible"                           |             |
    |                                         | AssertionError: Texts/Malware datasets are not possible                                                               |             |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------+-------------+
```

Figure A.2: SADL : Benchmarking Tasks Results

A.6 Benchmarking Tasks Results for DLFuzz Tool

```
2020-04-26 12:14:01,403 - root - INFO - Total Execution Time taken to run all the commands :579.860475063 Seconds
2020-04-26 12:14:01,407 - root - INFO -
Executing Benchmakring Tasks one by one....

 ********** DL Testing Tool Benchmarking Report **********

Status:
    Tasks Passed: 5
    Tasks Failed: 4

Summary of Benchmarking Tasks Execution:
    +-----------------------------------------+-------+------+------+-------+
    |          Task group/Sub-Task            | Count | Pass | Fail | Error |
    +-----------------------------------------+-------+------+------+-------+
    | BenchmarkingTasks:  Results of 7 Tasks | 9     | 5    | 4    | 0     |
    | Total                                   | 9     | 5    | 4    | 0     |
    +-----------------------------------------+-------+------+------+-------+
    BenchmarkingTasks
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+-------------+
    |                Task name                |                                                                  Failure Message/Info                                                                     | Task Status |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+-------------+
    | test_Differential_Testing               | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 186, in test_Differential_Testing                                                                                    |             |
    |                                         |     assert self.manual_check['differential_testing'] in self._pass,"Differential Testing is not possible"                                                |             |
    |                                         | AssertionError: Differential Testing is not possible                                                                                                      |             |
    | test_Execution_Time                     | 579.86 Seconds                                                                                                                                            | pass        |
    | test_Image_Classifications_Support      |                                                                                                                                                           | pass        |
    | test_Model_Selection                    | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 148, in test_Model_Selection                                                                                         |             |
    |                                         |     assert self.manual_check['model_selection']  in self._pass, "Model Selection is not possible"                                                        |             |
    |                                         | AssertionError: Model Selection is not possible                                                                                                           |             |
    | test_OS_Support                         | Darwin-19.3.0-x86_64-i386-64bit                                                                                                                           | pass        |
    | test_Output_Capabilities                |                                                                                                                                                           | pass        |
    | test_Retraining                         |                                                                                                                                                           | pass        |
    | test_SelfDriving_Classifications_Support | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 162, in test_SelfDriving_Classifications_Support                                                                     |             |
    |                                         |     assert self.datasets['self_driving']  in self._pass, "Self_driving datasets are not possible"                                                        |             |
    |                                         | AssertionError: Self_driving datasets are not possible                                                                                                    |             |
    | test_Texts_Classifications_Support      | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 172, in test_Texts_Classifications_Support                                                                           |             |
    |                                         |     assert self.datasets['texts']  in self._pass, "Texts/Malware datasets are not possible"                                                               |             |
    |                                         | AssertionError: Texts/Malware datasets are not possible                                                                                                   |             |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+-------------+
```

Figure A.3: DLFuzz : Benchmarking Tasks Results

A.7 Benchmarking Tasks Results for DeepFault Tool

```
2020-04-26 11:51:51,073 - root - INFO - Total Execution Time taken to run all the commands :177.959795952 Seconds
2020-04-26 11:51:51,075 - root - INFO -
Executing Benchmakring Tasks one by one....

 ********** DL Testing Tool Benchmarking Report **********

Status:
    Tasks Passed: 6
    Tasks Failed: 3

Summary of Benchmarking Tasks Execution:
    +-----------------------------------------+-------+------+------+-------+
    |          Task group/Sub-Task            | Count | Pass | Fail | Error |
    +-----------------------------------------+-------+------+------+-------+
    | BenchmarkingTasks:  Results of 7 Tasks | 9     | 6    | 3    | 0     |
    | Total                                   | 9     | 6    | 3    | 0     |
    +-----------------------------------------+-------+------+------+-------+
    BenchmarkingTasks
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+-------------+
    |                Task name                |                                                                  Failure Message/Info                                                                     | Task Status |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+-------------+
    | test_Differential_Testing               | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 186, in test_Differential_Testing                                                                                    |             |
    |                                         |     assert self.manual_check['differential_testing'] in self._pass,"Differential Testing is not possible"                                                |             |
    |                                         | AssertionError: Differential Testing is not possible                                                                                                      |             |
    | test_Execution_Time                     | 177.96 Seconds                                                                                                                                            | pass        |
    | test_Image_Classifications_Support      | With Image Classifications Model:Images_cifar10_1_512_leaky_relu_model1                                                                                   | pass        |
    | test_Model_Selection                    |                                                                                                                                                           | pass        |
    | test_OS_Support                         | Darwin-19.3.0-x86_64-i386-64bit                                                                                                                           | pass        |
    | test_Output_Capabilities                |                                                                                                                                                           | pass        |
    | test_Retraining                         |                                                                                                                                                           | pass        |
    | test_SelfDriving_Classifications_Support | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 162, in test_SelfDriving_Classifications_Support                                                                     |             |
    |                                         |     assert self.datasets['self_driving']  in self._pass, "Self_driving datasets are not possible"                                                        |             |
    |                                         | AssertionError: Self_driving datasets are not possible                                                                                                    |             |
    | test_Texts_Classifications_Support      | Traceback (most recent call last):                                                                                                                        | fail        |
    |                                         |   File "benchmarking_tasks.py", line 172, in test_Texts_Classifications_Support                                                                           |             |
    |                                         |     assert self.datasets['texts']  in self._pass, "Texts/Malware datasets are not possible"                                                               |             |
    |                                         | AssertionError: Texts/Malware datasets are not possible                                                                                                   |             |
    +-----------------------------------------+-----------------------------------------------------------------------------------------------------------------------------------------------------------+-------------+
```

Figure A.4: DeepFault : Benchmarking Tasks Results

A.8 Benchmarking Pre-Trained Model's Architecture

A.8.1 Image Classification

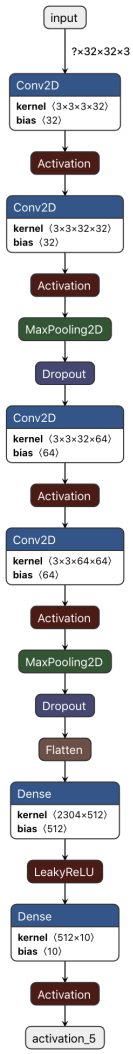

Figure A.5: Keras Cifar-10 CNN Model

Figure A.6: Keras MNIST CNN Model

A.8.2 Self-Driving Classification

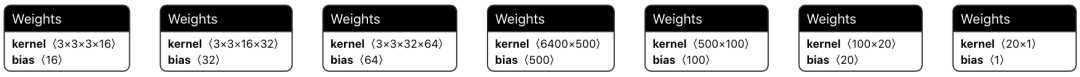

Figure A.7: Nvidia Dave Self-Driving Model 1

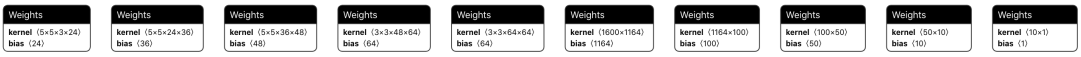

Figure A.8: Nvidia Dave Self-Driving Model 2

A.8.3 Texts Classification

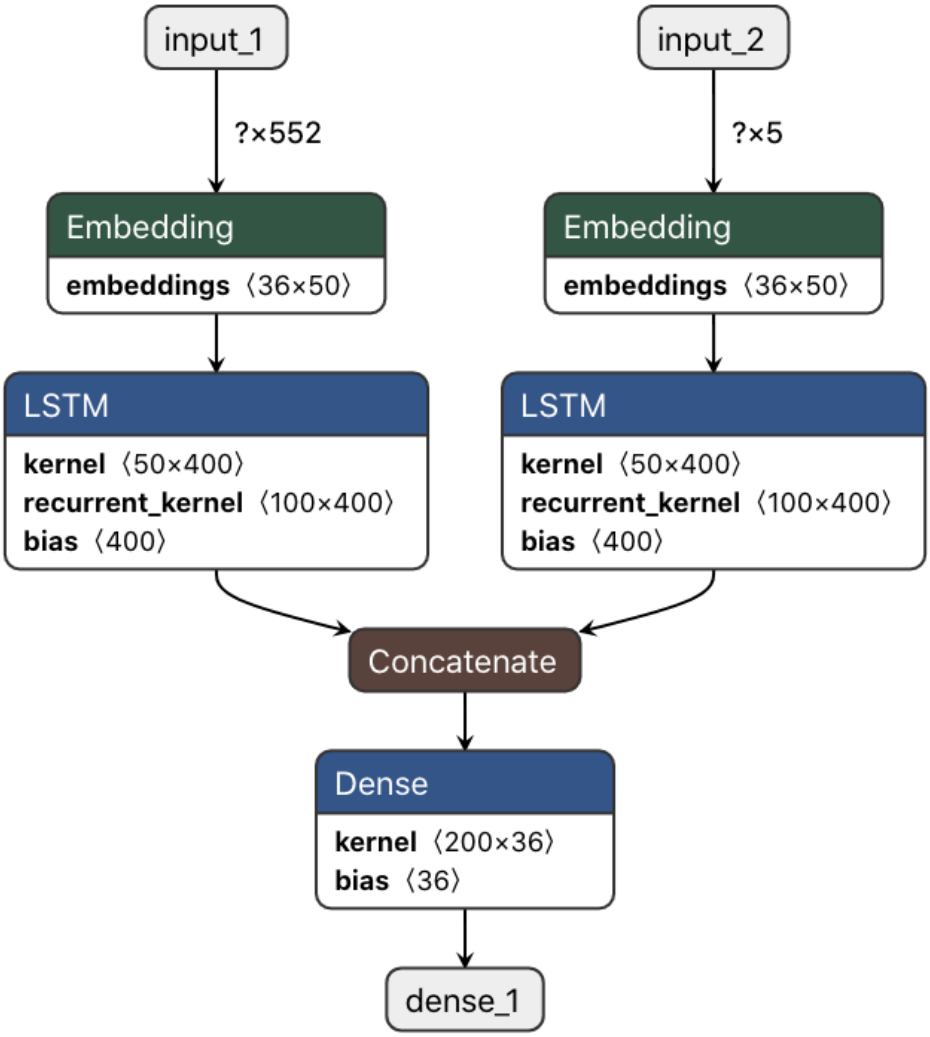

Figure A.9: Text Babi RNN based Model

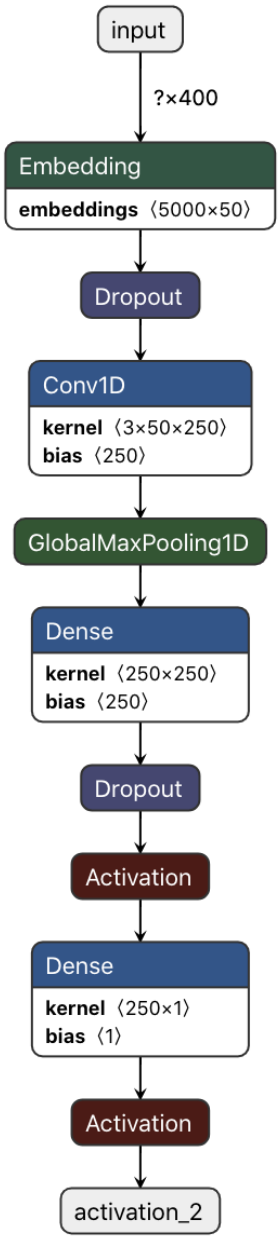

Figure A.10: Text Imdb RNN based Model