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
Advisor: Name, Company or Institute (if applicable)
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

Cover: Description of the picture on the cover page (if applicable)

Typeset in LaTeX
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

Abstract text about the project in Computer Science and Engineering. The abstract will be covered toward the end of the thesis.

Keywords: Deep Learning, DL Testing Tools, Testing, Benchmarking, project, thesis.

Acknowledgements

We would like to thank our supervisor Robert Feldt for continuous input, assistance and counseling throughout the thesis work.

Himanshu Chuphal, Kristiyan Dimitrov, Gothenburg, March 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[3][10]. Some of the most popular examples include self-driving cars, malware detection, and aircraft collision avoidance systems [4]. Due to their nature, safety-critical systems undergo rigorous testing to assure correct and expected software behavior. Therefore testing of DL systems becomes crucial, which has traditionally only relied on manual labeling/checking of data[10]. There are multiple DL testing techniques to validate different parts of a DL system's logic and discover the different types of erroneous behaviors. Some of the more popular DL testing techniques [3][17][4][10][29], which validate deep neural networks using different approaches for fault detection. 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 platform and DL software tools[30] 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[32]. Recently, a number of good DL testing tools have been proposed. 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)[32]. Since, there is no standard

method 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 guidelines proposed by Sim et al.[33] for an academic purpose. 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 which 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 properties of 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 application scenarios? Our goal is to design a methodology that uses experimental tests with a relevant DL system using DL testing tools for diverse datasets and benchmark the results. Check the possibility to get feedback from industry experts and check the applicable improvement in the method?

Hypothesis 1: There are significant qualitative differences in the selected DL testing tools in terms of testing properties.

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 and possibly industry experts 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 addresses threats to validity. Finally, Chapter six concludes the thesis and mentions future work and possible research opportunities.

2
Related Work and Background

This section starts with basic introduction about testing in general. Then it defines DL testing, explaining the very recently emerging research in the field of DL testing tools and techniques. It also depicts the key differences between traditional software testing and DL testing. Moreover, the brief literature review explains DL testing workflow, components and testing properties that will help the reader to get a better understanding of the DL testing. Section 2.5 aims presenting trends concerning DL testing tools, data sets, research trends, open source tools and research focus highlighting research challenges in DL testing.

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[2]. 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 to combat the challenges associated with testing economic 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 pro-gram as a black box. In this approach, test data are derived solely from the specifications (i.e., without taking advantage of knowledge of the internal structure of the program)[2].

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[2].

Traditional software testing is done through the use of test cases against which the software under test is examined [6]. 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 as to ensure program correctness [9]. For that purpose the concept of Test Adequacy Criteria has been introduced. Test adequacy criteria are 'rules' and conditions that the test set needs to comply with as 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

This section gives a definition and analyses of DL testing. It describes the DL testing workflow, DL testing components and DL testing properties. It also gives challenges and research trends in the field of DL testing and techniques.

2.2.1 Definition

A DL system to be 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 a 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 categorised as: Training data (the data used to train the algorithm to perform its task), Validation data (the data used to tune the hyperparameters of a learning algorithm), and Test data (the data used to validate DL model behaviour). 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 audio signals 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 learn the model parameters. The model performance on the validation set is used to tune the hyperparameters. The testing set data is used to evaluate the model performance on unseen data following the same steps as above.

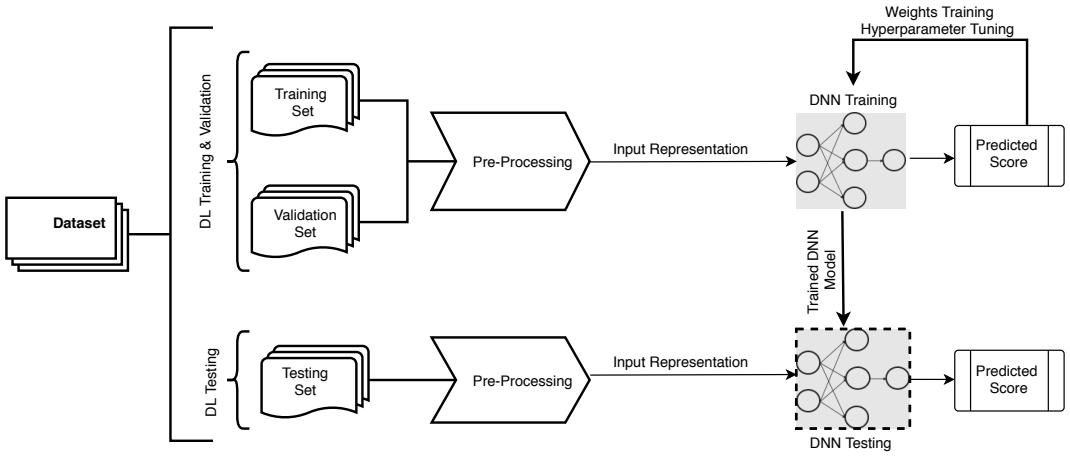

Figure 2.1: DL System Phases

There are two stages in a DL system namely Training Stage and Testing Stage Figure 2.2 shows stages a DL system.

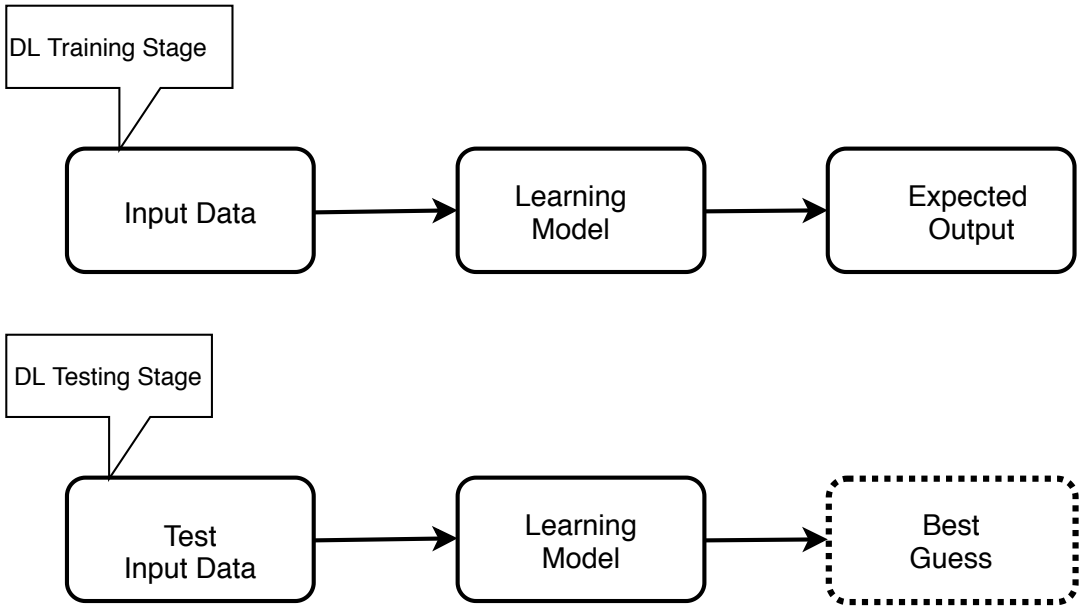

Figure 2.2: DL System Stages

Definition : A DL bug refers to any fault in a deep learning system that causes a discordance between the existing and the required conditions. We define DL testing as any activity aimed to reveal bugs in a DL system. Figure 4.1 depicts the overview DL testing tools process in general.

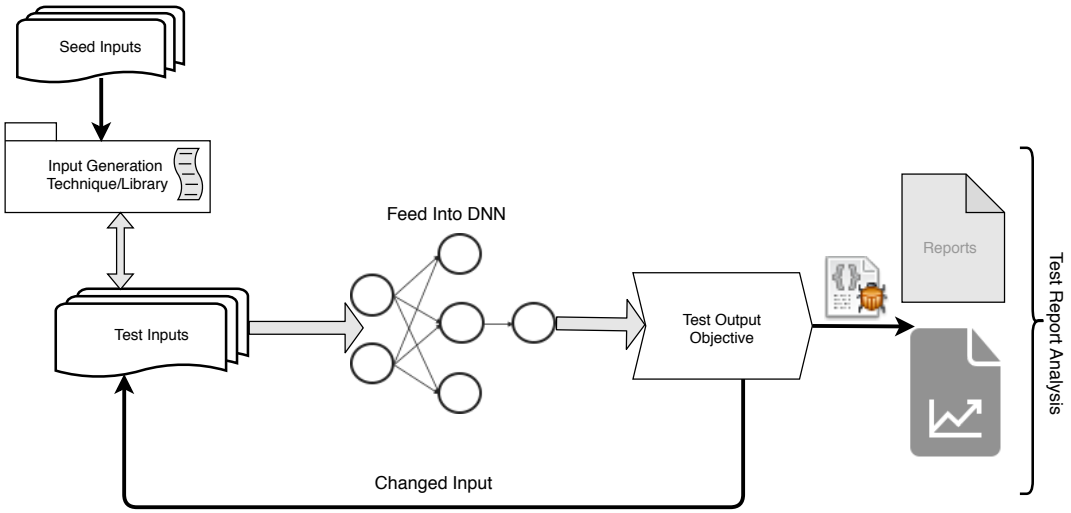

Figure 2.3: Overview of DL testing tools process

After going through twenty of the most popular, or latest, DL testing tools/frameworks, we have identified a generic process which is present within the research papers,

depicted in Figure 4.1. Although, the techniques vary in how each activity is executed, the process generally involves generation of new testing input whether by image-transformation[17], mutation [4][15], gradient descent[3], etc. of the original input or generating entirely new synthetic input based on the original[18]. The test cases are made out of that input and are ran through the DL system as to fulfill a testing objective, i.e. discover robustness related behavioural errors, increase neuron coverage, etc. Finally, either inputs are retained for a new iterations 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 deep learning systems since the deep learning systems are different from conventional software in many aspects. Till now, a lot of researchers have devoted their efforts to enhance the testing of deep learning models. Some of the differences are listed in Section 2.3.

We have defined the entire DL testing process in multiple sections. Figure 4.1 depicts the structure of following sections on DL testing.

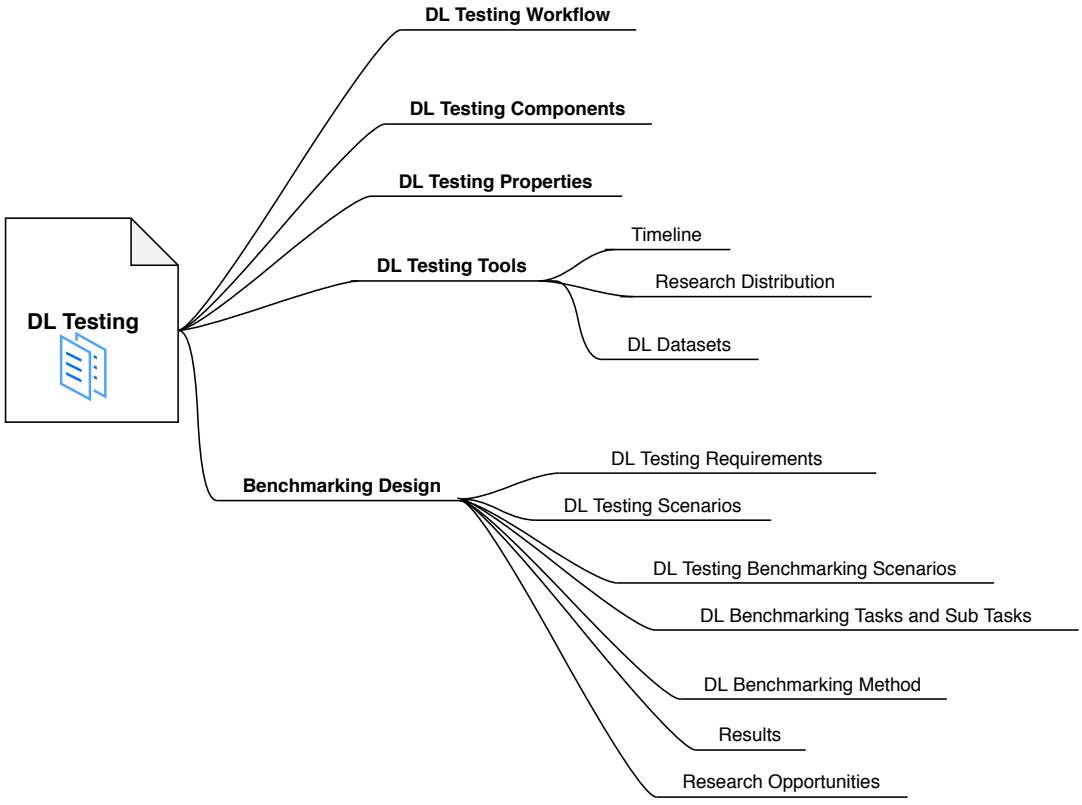

Figure 2.4: DL Benchmarking Structure Overview

2.2.2 DL Testing Workflow

The DL testing workflow is about how to conduct DL testing with different testing activities. In this section, we introduce the five key activities in DL testing and the approaches that are involved with those activities. Figure 2.5 shows different DL testing workflows.

The first activity of testing a DL system as per deduction of various testing technique papers is the Test Input Generation. This activity usually consists of modifying the existing testing input set[4][17] or generating new testing input[18]. 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 behaviour. 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[4][3][14][17][19]. 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 confirm 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[10] 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 behaviour within a DL system has been found, it needs to be retained because that input becomes interesting. In this particular case, interesting is used as to depict the possible wide usability of the error inducing input. DeepStellar[14] uses the error inducing input as a seed for its repeated test generation to improve overall coverage. Whereas other techniques like SADL[10] 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 labelled in most cases as to improve the model and eliminate the detected behavioural errors. Making this process automated would greatly improve the testing process as it would automatically fix behavioural errors per test iteration. Currently, most techniques are unaware on 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.

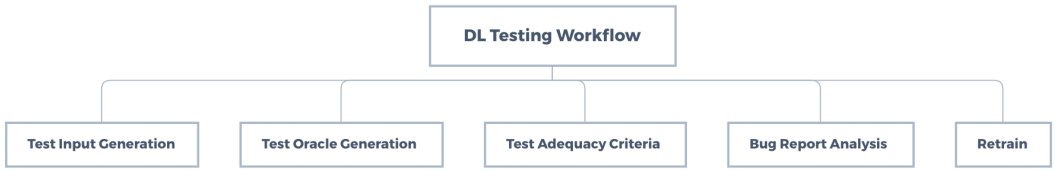

Figure 2.5: DL Testing Workflow

2.2.3 DL Testing Components

The DL testing components is about where to test in DL systems. In this section, we introduce three key components in DL testing. Figure 2.5 shows different key DL testing components.

DL testing component is one of the most important aspect of DL testing which comprises of testings component such as DL data, DL testing learning program/script, or DL testing framework) for which a DL testing tool might find a bug. To develop a DL model, an DL software engineer usually needs to collect data, label the data, design learning program architecture, and implement the proposed architecture based on specific frameworks. 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 [32]. 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. Figure 2.6 shows different DL Testing Components.

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

DL Testing Program:
DL testing program the script/program written by DL software engineer to build and validate the DL system. As shown in Figure 2.2 and Figure 4.1, 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.

DL Testing framework/library: DL Testing framework is the library, or platform being used when building a DL model, for example TensorFlow [39], Keras [32], and Caffe [40], Scikit-learn [41], etc.

.

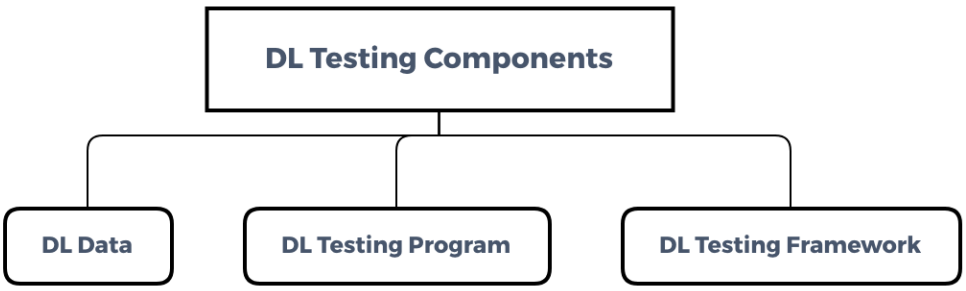

Figure 2.6: DL Testing Components

Some libraries have been around for years while new library like TensorFlow has come to light in recent years. Most of popular ones are open source and are popular in the data scientist community. Tensorflow [39] attracts the largest popularity on GitHub compare to the other deep learning framework, which is Google open source project. It is the most famous deep learning library these days. It was released to the public in late 2015. It is is developed in C++ and has convenient Python API, although C++ APIs are also available. Prominent companies like Airbus, Google, IBM and so on are using TensorFlow to produce deep learning algorithms.

Keras [38] is a Python framework for deep learning. It is a convenient library to construct any deep learning algorithm. The advantage of Keras is that it uses the same Python code to run on CPU or GPU. Besides, the coding environment is pure and allows for training state-of-the-art algorithm for computer vision, text recognition among other. Keras has been developed by François Chollet, a researcher at Google. Keras is used in prominent organizations like CERN, Yelp, Square or Google, Netflix, and Uber. Table 2.1 shows different open source DL Testing frameworks.

Table 2.1: Deep Learning Libraries

Library	Platform	Written in	Cuda support	Parallel Execution	Trained Models?	RNN	CNN
TensorFlow	Linux, MacOS, Windows, Android	C++, Python, CUDA	Yes	Yes	Yes	Yes	Yes
Keras	Linux, MacOS, Windows	Python	Yes	Yes	Yes	Yes	Yes
Caffe	Linux, MacOS, Windows	C++	Yes	Yes	Yes	Yes	Yes
Torch	Linux, MacOS, Windows	Lua	Yes	Yes	Yes	Yes	Yes
Theano	Cross-platform	Python	Yes	Yes	Yes	Yes	Yes
Microsoft Cognitive Toolkit	Linux, Windows, Mac with Docker	C++	Yes	Yes	Yes	Yes	Yes
MXNet	Linux, Windows, MacOs, Android, iOS, Javascript	C++	Yes	Yes	Yes	Yes	Yes
Infer.Net	Linux, MacOS, Windows	Visual Studio	No	No	No	No	No

2.2.4 DL Testing Properties

Testing properties refer to what quality characteristics to test in the DL system: what conditions DL testing needs to guarantee for a trained DNN model. This section lists some typical properties that the literature has considered. We classified them 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 behaviours of a DL system and deserve being treated independently when testing a DL system. Figure 2.7 shows different functional and non-functional properties in DL testing.

Non-functional properties

Currently, the majority of the DL testing techniques research has been focused on a very particular property and that is *robustness*. Before the reasoning behind why research has been focused on this property is explained an understanding on what robustness of a DL system is must be made. 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 less robustness issues. *Generality* as a DL system property has been brought up as well in testing tool research[15], 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. Regrettably, 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 of techniques which serve to evaluate the training data like DeepMutation. Similar to *generality*, *reliability* has been brought up with the DL testing technique research [4] but it was not further built-upon as the main focus was *robustness*.

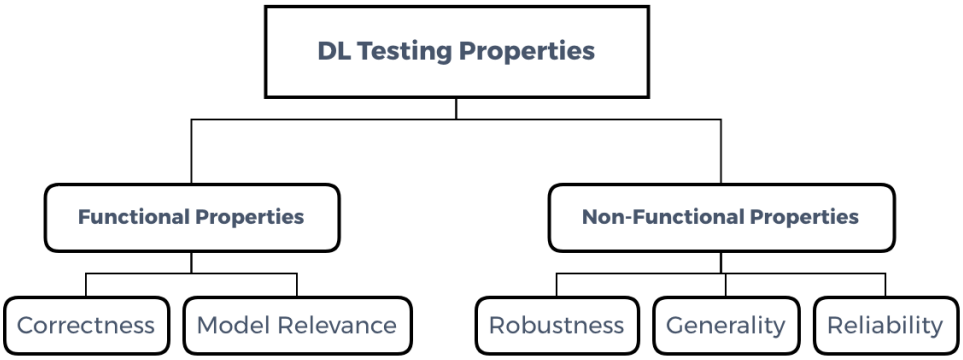

Figure 2.7: DL Testing Properties

Functional properties

The first functional property of importance is system *correctness*. Correctness is essentially the system's prediction accuracy. Similar to *robustness*, DL system *correctness* is heavily reliant on the quality of the training data[13]. 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 difficult 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 [10][12][4] as to improve correctness.

The second functional property that we've defined as important is DNN model relevance. Depending on what type of task the DL system is made for, an appropriate

DNN model will have to be selected and it is not always obvious as to which DNN should be used. Even though, DNN selection for a DL system has been discovered to be a potential issue [11], for the testing community this property is mostly relevant to the coverage of the variety of the DNN models. Techniques that apply for one type of DNN may not for another. DLFuzz[4] focuses its testing technique on a Convolutional Neural Network (CNN), DeepStellar[14] on the other hand is aimed at Recurrent Neural Networks (RNN), etc.

2.3 Software Testing vs. DL Testing

As defined by Guo et. al. [4], 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 2.2 shows the key differences between traditional software testing and DL Testing.

In traditional software development, developers specify the clear logic of the system, whereas 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 2.8 shows the key the comparison between the two systems.

Table 2.2: 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

We have identified several prominent testing techniques which were successfully adapted from traditional software testing, i.e. differential, mutation, metamorphic, combinatorial and fuzz testing. The following sections explain each type of technique in detail. Keep in mind that although some papers focus on highlighting an individual technique, it usually uses more than one technique to fulfill it's purpose.

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 behaviour or output then there's a candidate for a bug-exposing test case [5]. However if both systems give the same output, even if a bug is present, the bug cannot be detected.

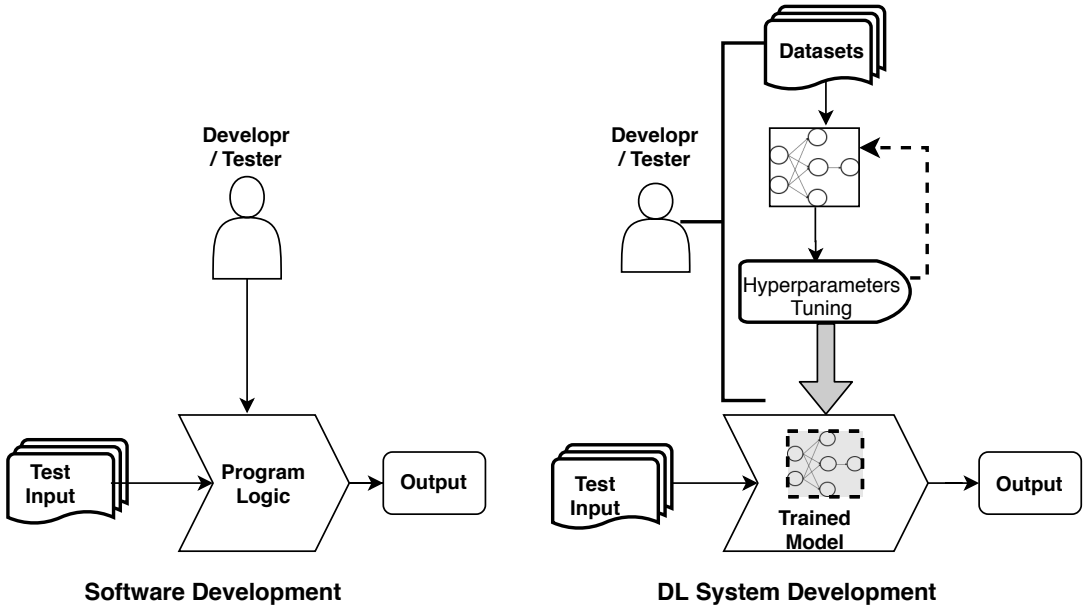

Figure 2.8: Comparison : Traditional Software vs DL system development

Pei et al. [3] 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 behaviour, particularly related to the robustness of the system. A different approach to differential testing was proposed by Guo et al. [4] as to avoid the 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 ran 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 [8]. 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[4] uses mutation as to mutate inputs and use the output of those mutants in cross-comparison to the original input's output. DeepMutation[15] on the other hand injects mutation faults not only in the input but the training program as well. Afterwards further mutation operators are

designed and injected directly in the model of the DL system. Other techniques like DeepStellar[14] 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*[7] which is built on the assumption that a test oracle is readily available but in practice that may not be the case [6]. The oracle problem is particularly present in applications which 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 classified 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 kth 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 [6]. 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 behaviour being correct. An example of how metamorphic testing and its reasoning approach is used in DL systems is DeepRoad[18]. 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 runtime 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 behaviour 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[20]. 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 interaction 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 that 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 judgement to be effective [21].
Within recent years, an effort has been made to adopt combinatorial testing to DL systems[22]. If the vast runtime 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[22], 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 as 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 input and the resulting values are checked for whether they are 'interesting'[23]. Due to it's success in detecting bugs many techniques have been developed that made different 'fuzzers' classify as blackbox, whitebox or gray-box [24]. 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[4][24] as a DL testing approach, there are also papers that do not explicitly focus on fuzzing techniques, but fuzzing is interwoven within frameworks[15][28].

2.4 Challenges in DL Testing

DL testing has experienced rapid recent 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.

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[10] 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[17]. 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[18], 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[3] uses adversarial sample generation and DeepFault[19] uses a supiciousness-guided algorithm to generate it's synthetic inputs. Various techniques are used to generate input as to be able to improve training data quality. After all, test inputs that triggers a faulty behaviour within a DL system can be added to the training data to resolve logical bug, or depending on the technique used the input can be retained for other uses, i.e. for a future test case seed[4].

Test Oracle Problem
One of the greater challenges of DL testing is the test oracle problem[7]. DL systems are 'predictive' systems, systems which are meant to provide us with an answer. This implicates automated testing because erroneous behaviour cannot be identified without manual labelling or checking[3]. Initial attempts at resolving the oracle problem were done through the use of differential testing[5] that resulted in the automated whitebox testing framework DeepXplore[3]. 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, large set of data that determines the weighs between neurons. A later attempt at differential testing was done by Guo et al.[4] 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 which is as complex as 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, what would traditionally be considered our second tier of testing as integration. In addition, this form of training is computationally expensive and time-consuming, usually not possible on local machines.

Complex DNN model
Calculating tensor multiplications are difficult and rarely can be done by an software engineers as "back of the envelope calculations". The maths is complicated for such complex models. Even with fixed seed initialisation, the regular Xavier weight initialisation uses 32-bit floats, matrix multiplication involves a large series of calculations and testing modules with batching involves hard-coding tensors with 3 dimensions. 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 fail silently, Whilst testing a DNN model behaves correctly with a specific input, the inputs of a neural networks 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.

More details to be added in the final report in this section.

2.5 DL Testing Tools

We investigated a total of twenty most popular latest DL testing tools and techniques. Following sections explain the details of fifteen such DL testing tools.

2.5.1 Timeline

Figure 2.9 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. [3] published the first white-box testing paper on deep learning systems. Their work pioneered to propose coverage criteria for DNN. Enlightened by this paper, a number of deep learning testing techniques have emerged, such as DeepTest [17], DeepGauge [12], DeepConcolic [16], and DeepRoad [18].

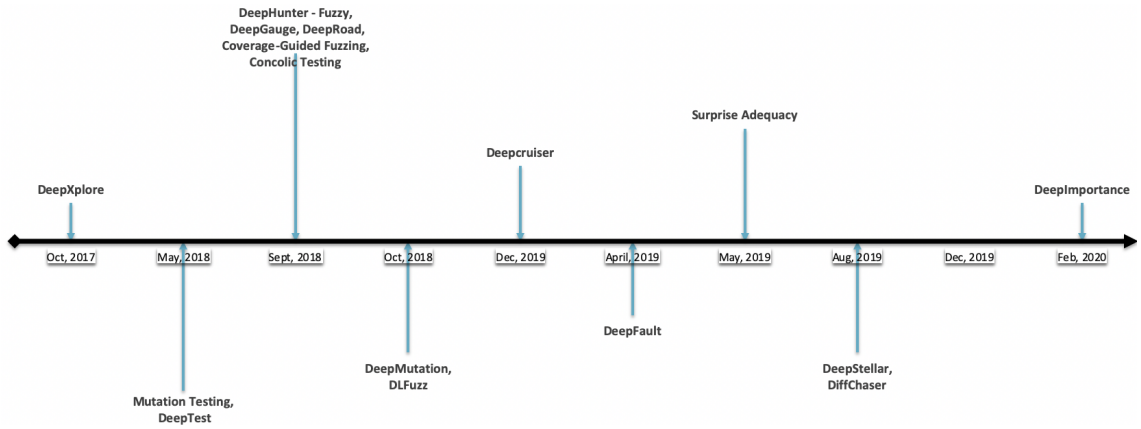

Figure 2.9: Timeline of DL testing tools research, 2020

2.5.2 Research Distribution

We further investigated the number of published papers in the field of DL testing and techniques for each year, to observe whether there is a trend of moving from testing general machine learning to deep learning testing. Figure Figure 2.10 shows commutative trends in **"Deep Learning System Testing"** and Figure 2.11 shows commutative trends in **"Testing "Deep Learning "**. Before 2017 and 2018, papers mostly focus on general machine learning; after 2018, we see a more dedicated focus on deep learning specific testing notably arise. Major publications came only in 2019 and most of the publications are focused on DL testing techniques. Therefore the research field or so to say DL testing techniques and tools are still in the process of maturing. The numbers shown in all the three graphs also include survey publications in the field of deep learning testing.

All the statistics are taken from Google Scholar and are focused around published papers.[1].

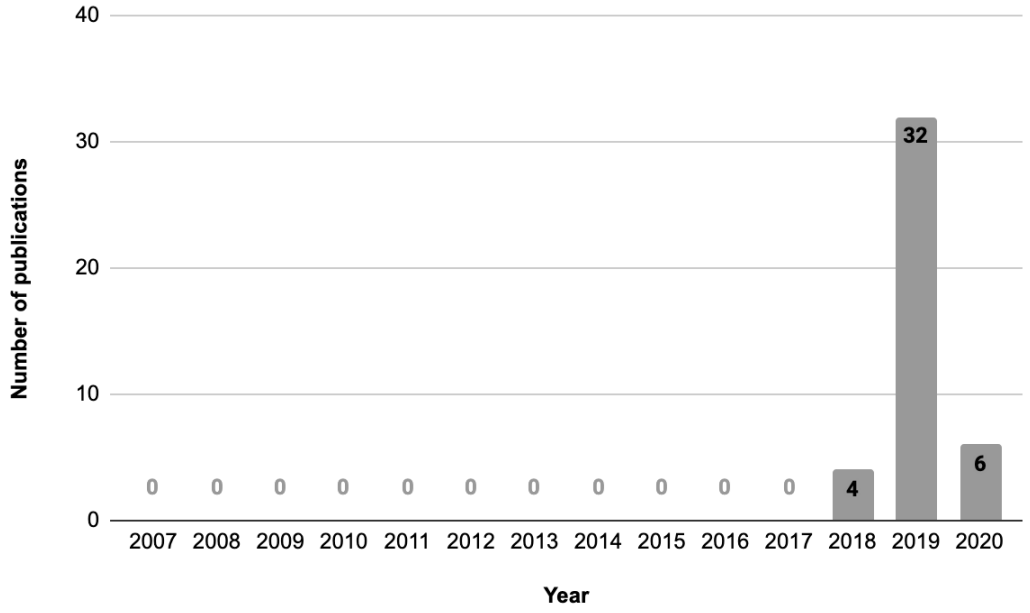

Figure 2.10: "Deep Learning System Testing" Publications

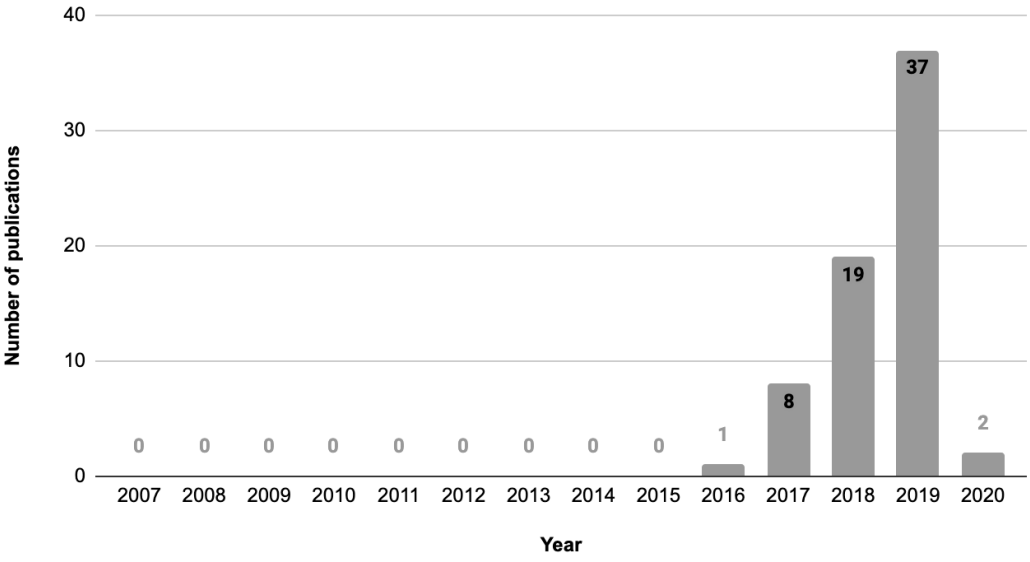

Figure 2.11: "Testing "Deep Learning" Publications

Comparing to publications for machine learning testing, as early as in 2007, Murphy et al. [32] mentioned the idea of testing machine learning applications, which is one of the first papers about testing Machine learning systems. Figure 2.12 shows the

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

commutative trends in **Testing ""Machine Learning "**. Table 2.3 shows total publication for each category.

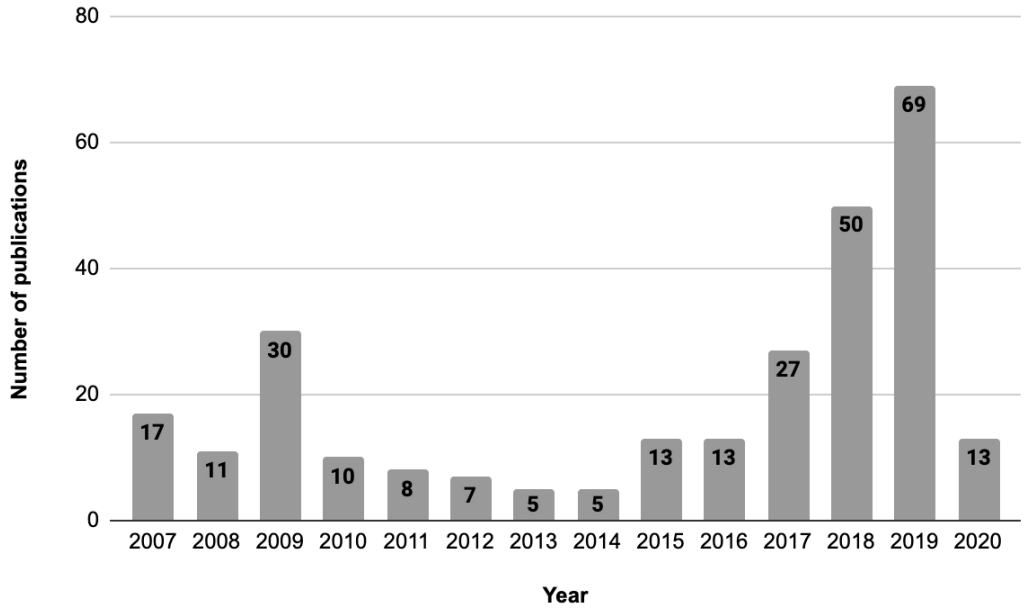

Figure 2.12: "Testing "Machine Learning" Publications

Table 2.3: Total 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

Deep learning being the game changer at the present day scenario, the datasets play a dominant role in shaping the future of the technology. The key to getting better at deep learning (or most fields in life) is practice. Practice on a variety of problems – from image processing to speech recognition. Each of these problem has it's own unique nuance and approach. Learning starts with getting the right data and the best way to mastering in this field is to get hands dirty by practicing with the high-quality datasets. But where can we get this data? A lot of research papers these days use proprietary datasets that are usually not released to the general public. This becomes a problem, if we want to learn and apply our newly acquired skills.

Table 2.4 to Table 2.9 show some 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 Deep Learning techniques. Some

of these datasets are huge in size. In each table, the first column shows the name and link of each dataset. The next three columns give a dataset information, the size and total number of records for each dataset.

The datasets are divided into following six key categories. Figure 2.11 shows DL datasets classification.

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

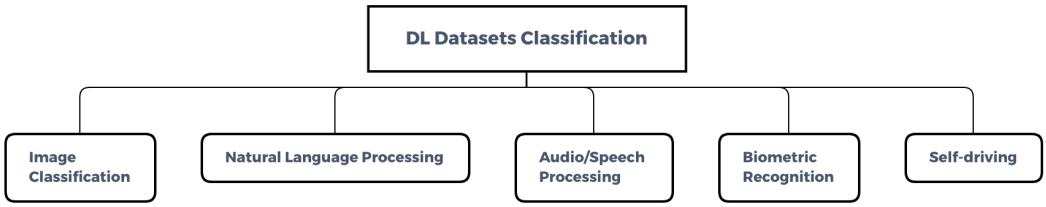

Figure 2.13: DL Datasets Classification

Table 2.4: DL Dataset : Image Classification

Dataset	Data 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 2.5: 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 2.6: 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 2.7: DL Dataset: Biometric Recognition

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

Table 2.8: 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 2.9: 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

For Benchmarking design, we foucsed only on the most popular datasets from Image classification and self driving category. We also discuss research directions of building dataset and benchmarks for DL testing in subsection 3.2.4.

2.6 Benchmarking Research

This sections 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?"*. Then, it explains what benchmarking is NOT and finally gives insight about benchmakring research work done in the field of DL systems and DL testing tools.

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: [36] (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 [30]. First, for the end users of deep learning tools, the 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. However, Sim at al.[33] 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 What Benchmarking is NOT

The Benchmarking Book: A How-to-Guide to Best Practice for Managers and Practitioners by Tim et al.[36] explains what benchmarking is not. In order to clarify what benchmarking is in the field of DL testing tools and techniques, it helps to consider what benchmarking is not in the similar terms but for DL testing tools in particular. Benchmarking:

- **IS NOT** a method to improve DL software tools and DL testing tools or techniques. However, **IS** planned research with an aim to help practitioners to select a most suitable DL testing tool or technique meeting their DL testing requirements and scenarios.
- **IS NOT** a tool to improve DL testing tools and techniques. However, **IS** helpful to identify where and how to improve the DL testing processes.
- **IS NOT** a copy & paste activity [36], copying what someone else has done in their organization would mean exactly same requirements, which may be very

much different. However, **IS** a potential source of ideas, information, methods, practices, etc. that a software tool may be appropriate to adapt, adopt, and implement.

- **IS NOT** one-off event [36], at the most this will lead to achieving a competitive edge today, but that is likely to be eroded as other organizations continue to improve in the future. However, **IS** part of a culture of striving to be the best, or amongst the best, at what the organization does (i.e. to solve specific problems or justify decisions for selecting most suited software tool).
- **IS NOT** a tool that can be applied to any software tool or technique for that matter. However, **IS** a set of benchmarking tasks and sub-tasks, using diverse scenarios to help the end users to select an appropriate software tool for a particular testing scenario.

2.6.3 Benchmarking in DL System

Deep learning systems, which are the focus of our research, have been shown as a successful machine learning method for a variety of tasks, and its popularity results in numerous open-source deep learning software tools. There is a guide for selection of appropriate hardware platform and DL software tools [30] 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 [35] explains three major difficulties that we need to consider for benchmarking such as technical bias for benchmarking framework design, hardware resources selection and the independence of different tool executions.
Moreover, there is also a survey paper [32] on 'Machine Learning Testing: Survey, Landscapes and Horizons' but this only focuses in Machine learning in general and it is just a survey on ML testing tools. For more on testing neural networks, we refer to the work of Zhang et al. [32] that surveys testing of machine-learning systems.

3
Methodology

3.1 Research Methods

This section starts with description of the methods used for the three research questions. Then it explains the DL testing benchmarking method, wherein it highlights key DL Testing Tool Requirements, testing scenarios and the applicable DL benchmarking scenarios. Finally, it explains the benchmarking design method.

3.1.1 Research Questions

3.1.1.1 Research Question 1

RQ1: What existing state-of-the-art and real-world applicable DL testing tools are available? We investigate the correctness, fairness, efficiency, and robustness properties of the existing DL testing tools and decide which testing tools are most suited for a relevant case.
For RQ1, we investigated twenty existing state-of-the-art research papers on DL testing and techniques. The focus of the study was to understand the idea, workflow, compo nets, testing properties setup and evaluation method, used DL datasets, limitations, working status and code support and availability of each DL testing tool, The result of pre-study having summary of all fifteen such DL testing tools and technique is presented in section 4.1.

3.1.1.2 Research Question 2

RQ2:To what extent do different testing techniques and workflows of the DL testing tools perform better towards DL application scenarios? Our goal is to design a formal method using experimental tests with a relevant DL system using DL testing tools for diverse datasets and benchmark the results.Check the possibility to get feedback from industry experts and check the applicable improvement in the formal method? **Hypothesis 1**: There are significant qualitative differences in the selected DL testing tools in terms of testing properties.
Hypothesis 2: There is a significant difference in test input generation of selected DL testing tools for a given type of dataset.
For RQ2 (*Hypothesis 1* and *Hypothesis 2*), we used the results from our literature review of fifteen selected research papers based on DL testing and techniques. Based on our review of each of these papers, we have identified DL testing properties for each techniques. Details for qualitative differences across these tools are presented

in section 4.1. The analysis for Hypothesis 2 to be presented in section 4.3 after half-report presentation. Due to bad situation of COVID-19 and lack of resources, getting feedback from Industry experts, which we have kept as optional in our thesis proposal is very unlikely but if allowed by the companies, we will try to present our benchmarking design to one or two companies in Göteborg.

3.1.1.3 Research Question 3

RQ3: How can the proposed benchmarking methodology of DL testing tools help practitioners and possibly industry experts as a reference for selecting an appropriate testing technique? We aim is to assess the benchmarking methodology using different DL testing tool 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.
For RQ3, we have designed benchmarking design based on identified DL testing tool requirements, test scenarios and the benchmarking tasks. The analysis and results of the design to be presented after half-report presentation in section 4.3 and section 4.2. However, due to bad situation of COVID-19 and lack of resources, getting feedback from Industry experts, which have kept as optional in our thesis proposal is very unlikely but if allowed by the companies, we will try to present our benchmarking design to one or two companies in Göteborg.

3.2 Benchmarking Method

For the benchmarking method design, we focus on the proposed five requirements for creating a successful benchmark within software engineering: relevance, solvability, scalability, clarity and portability. [33]. As such the study will have characteristics of both formal experiments and case studies.
Relevance: The relevance requirement in the case of the DL testing tool benchmark is related to the task sample. The datasets, DNNs models and benchmarking tasks need to be representative of actual data and situations that the system is expected to handle.
Solvability: Similar to the relevance requirement, the solvability requirement 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: This requirement involves the benchmark being able to work with as wide a variety of 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.
Clarity: The clarity requirement involves the understandability and conciseness of the benchmark. This requirement will be tested if an optional industry contact is found. Through the feedback, we'll discover if the benchmark's clarity requirement is fulfilled.

Portability: The portability aspect in the case of the DL benchmark is concerned with the systems and architecture on which the DL system will be run, as well as the system itself. Due to resource restrictions, the benchmark will be constructed to work using the available hardware and software.

The rest of this Section will be primarily covering the reasoning and steps behind the design of the tasks due to their large role and conclude with the final steps of the overall benchmark design.

3.2.1 DL Testing Tool Requirements

Currently research is focused on making progress in improving DL system *robustness*, as elaborated in Section 2.2.4, without having agreed on a common metric to measure the improvement, aside from neuron coverage. And although neuron coverage has been agreed upon as a viable metric for improving system robustness, it usually undergoes a sort of transformation before being shown as output. This leads to tools giving a different type of output all together. This inconsistency is a large challenge to the benchmark's design if performance is thought about in a quantitative way that clearly shows a tool's superiority over another in terms of testing a DL system. However, as pointed out by Sim et al.[33], performance is not a 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 for device meaningful performance metrics. Thus, we took a software engineering approach to the challenge by establishing a set of requirements which a testing tool should comply with based on the research done for **RQ1**. For the elicitation of the requirements we first got 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, hence are not part of the results but the necessary background work. For the elicitation process we used elicitation techniques that were reviewed by Sharma et al.[34]. The two main techniques used were *Reading Existing Documents* and complemented it with *Brainstorming*. For the existing documents we refer to the DL testing tool papers which we investigated in RQ1. The brainstorming session was conducted between the researchers conducting the study. While brainstorming sessions are more effective with various stakeholders, other researchers on DL testing tools and users of such tools, we are unable to get such participants. Due to the same reason other techniques like interviews, meetings, etc. were not applicable.

3.2.2 DL Testing Scenarios

From the set of requirements, we establish benchmark testing scenarios for the 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 which

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 touches on one of the seven properties proposed by Sim et al.[33] for a successful benchmark, **Relevance**. The tasks set of the benchmark must be representative of what the tools are supposed to handle.

3.2.3 Test Scenarios for Benchmarking Design

Before the scenarios can be synthesized into tasks for the benchmark, another important property must be covered at this stage of design and that is **Solvability**, "it should be possible to complete the task domain sample and to produce a good solution"[33]. A task must not be too difficult for a system to give sufficient results for comparison or too trivial to matter. In this case, the same can be said for the scenarios because they are the groundwork we use for the tasks. Additionally, due to the current state of the DL testing tools, some scenarios may also be, while reasonable to consider, unfeasible to be added, i.e. technical feasibility, lack of support from the tools, etc.

3.2.4 Benchmarking Tasks Design

After the scenarios have been filtered out, they need to be synthesized into tasks which are solvable to the tools. Ideally, the tasks that a benchmark would give to the tools would strike a balance between difficulty and feasibility as they need to be difficult enough for a tool to show its full potential but not too difficult for it to give any meaningful output. In the case of DL testing tools the matter is even more complicated as the benchmark cannot evaluate the output of a tool directly due to the reasons stated in Section 3.2.1. Therefore the tasks are aimed at the capabilities that the tool has. If the capabilities are too specific then no meaningful for comparison data would be gathered, hence the tasks would need to be sufficiently general, targeting the commonalities between the tools, yet need to address relevant issues within the tools. The scenarios provide the groundwork for the individual tasks but the tasks extend the groundwork into meaningful objectives for the tools.

3.2.5 Benchmarking Design

Once the tasks have been defined the construction of the actual benchmark will have to be implemented. Here the final three properties will be covered: scalability, clarity and portability. Scalability and Portability are heavily dependent on the script design and possibly available hardware and will be done as best to our abilities within the restricted time frame as possible. For the clarity property we plan on possibly having industry feedback as to refine the benchmark. Due to the pandemic outbreak of COVID-19, however, that is very unlikely.

4
Results

4.1 Results from Pre-Study

The goal of the pre-study is to study research papers on DL testing tools and techniques that could be used for the designing benchmarking method. In this section, we present the results of the pre-study explaining all the investigated fifteen DL testing tools and techniques.

4.1.1 DL Testing Tools

For **RQ1**, we have investigated fifteen existing state-of-the-art research papers on DL testing and techniques. Table 4.1 shows summary of all fifteen such DL testing tools and technique. The 1st column shows the name of each tool/techniques, followed by year of publication, type of testing, use-cases, diverse datasets, Code support, and open source. The investigation tasks also involved identifying relevant use-case(see 'Use-Case' column, incomplete).

Table 4.1: RQ 1: Summary of state-of-the-art DL Testing Tools and Techniques

Tool Name	**Year**	**Type of Testing**	**Test Properties**	**Diverse Datasets**	**Code Support**	**Open Source**
DeepXplore	2016	Whitebox Differential	Robustness Correctness Reliability	Images Self Driving Textual	Yes	Yes
Surprise Adequacy	2019	Test Adequacy Criterion	Robustness Correctness	Images Self Driving	Yes	Yes
DLFuzz	2018	Differential Fuzzing	Robustness Reliability	Images	Yes	Yes
DeepConcolic	2018	Concolic	Robustness	Images	Yes	Yes
DeepFault	2019	Whitebox	Robustness Generality	Images	Yes	Yes
nMutant	2018	Mutation	Robustness	Images	Yes	Yes
DeepTest	2018	Metamorphic	Robustness	Autonomous Driving	Yes	Yes
DeepHunter - Fuzzy	2018	Coverage Guided Fuzzing	Robustness	Images	Yes	Yes
DeepGauge	2018	Multi-granularity test criteria	Robustness Generality	Images	No	NDA
DeepMutation	2018	Mutation	Robustness Generality	Images	No	NDA
Deepcruiser	2018	Coverage Guided Metamorphic	Reliability	Speech-to-text	NA	NA
DeepRoad	2018	Metamorphic	Robustness Generality	Autonomous Driving	NA	NA
DeepStellar	2018	Coverage Guided	Robustness	Images Speech to Text	NA	NA
Coverage-Guided Fuzzing	2018	Coverage-Guided Fuzzing	Robustness Reliability	Images	Yes	Yes
DiffChaser	2019	Black-box	Correctness	Images	NA	NA

We also did a comparison of most common DL testing properties across the selected latest fifteen research papers on DL testing and techniques.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 papers. It is followed by Reliability, Correctness and generality. The testing property 'Model Relevance' that was shown as an issue in the paper by Jahangirova et al.[11], does not appear as a point of concern within the DL testing tool/technique research papers. This is due to the papers being focused on specific types of DNN and how to improve the non-functional properties of those DNNs. Currently, to the best of our knowledge, there seems to be no testing tool that checks whether the correct type of DNN is used for the task that the system is meant to solve. subsection 2.2.4 explains various DL testing properties across all the research papers.

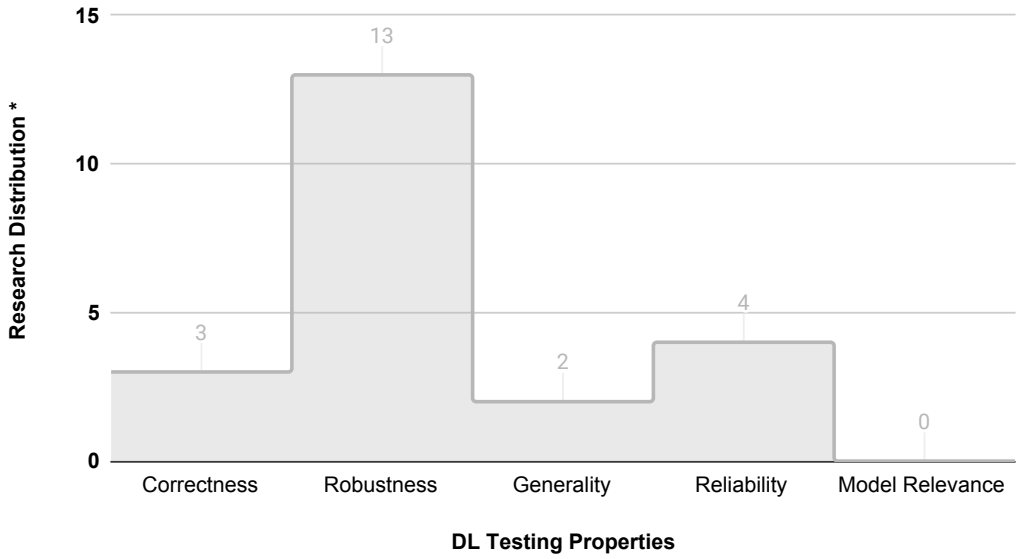

Figure 4.1: DL Testing Properties Research distribution

In the rest of the section we present the fifteen tools we investigated. Each tool is presented by first explaining the concept that the tool is undertaking, followed by details of it's setup and evaluation, supported data classification and finally known limitations.

DLfuzz:
DLFuzz[4] is a differential fuzzing testing framework that approaches cross-referencing in a different manner than DeepXplore[3]. 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
It include datasets such as MNIST and ImageNet and the corresponding CNNs. The empirical evaluations on two well-known datasets to demonstrate its efficiency. Compared with DeepXplore, the state-of-the-art DL Whitebox testing framework, DLFuzz does not require extra efforts to find similar functional DL systems for cross-referencing check, but could generate 338.59% more adversarial inputs with 89.82% smaller perturbations, averagely obtain 2.86% higher neuron coverage, and save 20.11% time consumption

Type of Testing
Whitebox framework. DLFuzz 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.

Limitations
To be added.

Surprise Adequacy:
The idea of surprise adequacy comes through the problem that it tries to solve. Surprise Adequacy for Deep Learning Systems (SADL) [10] 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[11]. 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 input which was 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. Many DL testing techniques [3][12] focus on treating 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 in respect to the training data used [10] 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 labelling when adding the identified data.

Setup and Evaluation

Easy setup guide, particularly easy to get it to run if you're running an isolated virtual environment. Uses relatively new technology: runs with python 3.6, hence it's usable on Windows. Uses a large data set making it hard to generate a model, 60 000 for MNIST, 50 000 for CIFAR-10. CIFAR-10 is far heavier to compute. The testing procedure itself on the model is quite fast in comparison.It involves Datasets such as MNIST, CIFAR-10, Dave-2, Chauffeur, involving measurements such as ROC-AUC score, FGSM coverage.

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 a 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 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. (will write a better explanation when I can actually explain it better). 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. Basically 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.

Limitations

To be added.

DeepXplore:

DeepXplore [3] is the 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. It introduced a new metric, neuron coverage, for measuring how many rules in a DNN are exercised by a set of inputs. 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 as (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 Test on the other hand, aims to Systematically testing 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 testing framework was evaluated on 15 state-of-the-art DL models which are trained using 5 real-world popular data-set with a total of 132,057 neurons trained on very popular datasets containing around 162 GB of data. Some of the examples of data-set are Udacity self-driving car challenge data, image data from ImageNet and MNIST, Android malware data from Drebin, and PDF malware data from VirusTotal. All the experiments were run on a Linux laptop running Ubuntu 16.04 (one Intel i7-6700HQ 2.60GHz processor with 4 cores, 16GB of memory, and an NVIDIA GTX 1070 GPU).

Type of Testing:
DeepXplore involves Whitebox framework and Differential Testing and has a good Code Support but on a old version of Tensorflow and Python 2.7.

Limitations:
DeepXplore being the the very first white-box testing technique in the filed DL testing, It 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.

DeepFault:
DeepFault [19] (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 suspiciousness-guided algorithm to synthesize new inputs that achieve high activation values of potentially suspicious neurons.

Setup and Evaluation
Empirical evaluation of several DNN instances trained on MNIST and CIFAR-10 datasets shows that DeepFault is effective in identifying suspicious neurons. Also, the inputs synthesized by DeepFault closely resemble the original inputs, exercise the identified suspicious neurons and are highly adversarial
Type of Testing
DeepFault used Whitebox framework and has a good Code Support but uses an old version of Python for its scripts and simulation.
Limitations
DeepFault doesn't fully capture all real-world input distortions and also lack the implementation details for simulation the setup for evaluation.

We also covered eleven more DL testing tools such as Concolic Testing, nMutant, DeepTest, DeepHunter - Fuzzy, DeepGauge, DeepMutation, Deepcruiser, DeepRoad, DeepStellar, Coverage-Guided Fuzzing and DiffChaser. Details of all the remaining tools will be added in the final report.

4.1.2 Open-source Tools

We investigated a total of most popular latest fifteen DL testing tools and techniques. Not all tools have good code support and some of them are not open source. Table 4.2 shows the code support and type of license for each DL testing tool. Some of the tools have code available but due to lack of code support and library, the scripts don't work.

Table 4.2: DL Testing Tools Availability

	Year	Code Support	Open Source
DeepXplore[3]:	2016	Yes	Yes
Surprise Adequacy[10]:	2019	Yes	Yes
DLFuzz[4]:	2018	Yes	Yes
Concolic Testing[16]:	2018	Yes	Yes
DeepFault[19]:	2019	Yes	Yes
nMutant[27]:	2018	Yes	Yes
DeepTest[19]:	2018	Yes	Yes
DeepHunter - Fuzzy[25]:	2018	Yes	Yes
DeepGauge[12]:	2018	No	NDA
DeepMutation[15]:	2018	No	NDA
Deepcruiser[28]:	2018	NA	NA
DeepRoad[18]:	2018	NA	NA
DeepStellar[14]:	2019	NA	NA
Coverage-Guided Fuzzing[24]:	2018	Yes	Yes
DiffChaser[26]:	2019	NA	NA

Out of the fifteen tools we investigated, we managed to get the code running only on

three tools. It is important to note that many of the tools are still in an early state and are designed to showcase a particular technique rather than being industrially available. Reaching the code of DeepXplore was fairly simple as the code is openly available and the support for the tool was good. However, DeepXplore runs on an old Tesorflow framework version (1.0) and Python 2.7. Using an older version of Tensorflow than 2 with Python 2.7 ,which is now officially obsolete and also not a default version on the operating system, making it incompatible on the windows operating system. DeepHunter suffers from a similar fate. While running a higher version of Tensorflow (1.5) it still requires Python 2.7 and is incompatible on the Windows OS. The Coverage-guided fuzzing technique proposed in the paper by Xie et al.[24] is implemented in DeepHunter. DeepMutation and DeepGauge are under an NDA, therefore the code is currently unavailable. However, both of the frameworks use Keras 2.1.3 and Tensorflow 1.5. We were unable to get the code for DeepCruiser, DeepRoad, DeepStellar and DiffChaser. The mutation-inspired testing algorithm proposed by Wang et al.[27], 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. DeepFault [19] has a good support for code base but has many issues with the libraries, which made it difficult for us to get any good output.
This section will be filled up based with more details on the final benchmarking work after half report presentation.

4.2 Benchmarking Method Results

4.2.1 DL Testing Tool Requirements

These requirements are there to help understand what a DL testing tool should consider according to primarily current research on DL testing and traditional testing.

1. Technical Feasibility
   - Fixed Model
   - Flexible Model Selection
2. Diverse Datasets
3. Test Input Generation
4. Execution Time
5. Open source/Code Support
6. Re-Training Models
7. Intuitive Test Report/Readability
8. Ease of Test Execution
9. Output validation
10. Platform
11. Application Stability
12. Latest Libraries
13. Cost

14. Ease of Developing and Maintaining the Scripts
15. Support to Web, Desktop & Mobile application
16. Technical Support and Assistance
17. Language Support

Requirement description and reasoning will be added for the final report.

4.2.2 DL Testing Scenarios

Introductory statement regarding the scenarios will be added at the final report.

1. Check DNN models selection flexibility
2. Check diversity in DL datasets test tasks
3. Check adversarial input generation
4. Check automatic labeling of test input
5. Check retraining of the DNN model under test
6. Check DL system cross-referencing oracle support
7. Check test execution time
8. Check DL testing tool code availability
9. Check testing output validation and readability
10. Check platform support

Check DNN models selection flexibility
Does the tool provide the option to select your own model? For the DL testing tools to be applicable to real-life DL systems, model selection should be possible. As such this scenario is important for the flexibility and applicability of the tool. This concern rose as we attempted to use tools like DeepXplore[3] and SADL[10] which did not allow for model selection within their run arguments.

Check diversity in DL datasets test tasks
How many datasets 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 MNIST and CIFAR-10, image datasets, but being able to only conduct testing on image datasets leaves many areas like speech-to-text and autonomous driving left out. 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 which involve specific datasets. This is further supported by the fact that within the industry a known problem is DNN selection[11]. A prime example of such is how Recurrent Neural Networks (RNNs) are good at audio, natural languages and video processing[14].

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.

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.

Check retraining of the DNN model under test
Does the tool support the retraining of the model after its tests? As an example, several tools find faults in the DL system and retain data that can be used for improving the model[10]. 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.

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 as the one under test by using the second system as a cross-referencing oracle[3]. Due to the complexity involved with DL systems, finding a similar system may prove to be difficult and unsuitable. As such we have identified this scenario as a part that provides integral information necessary for a tool's *generality*.

Check test execution time
How long does a 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 from or another[4][18], the comparison is usually between the tool in question and at most one other tool and execution time is rarely involved.

Check DL testing tool code availability
Currently, several tools, while possibly better than others, are either not open-source or under an NDA.

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 reference. 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 saving of the results may not be present.

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 tools generality.

4.2.3 Benchmarking Tasks

Table 4.3: 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**

We have identified seven DL tests task Ids mapped to DL testing scenarios. Figure 3 shows different tasks and their flows. Figure 4.2 shows division of all benchmarking Tasks Design.

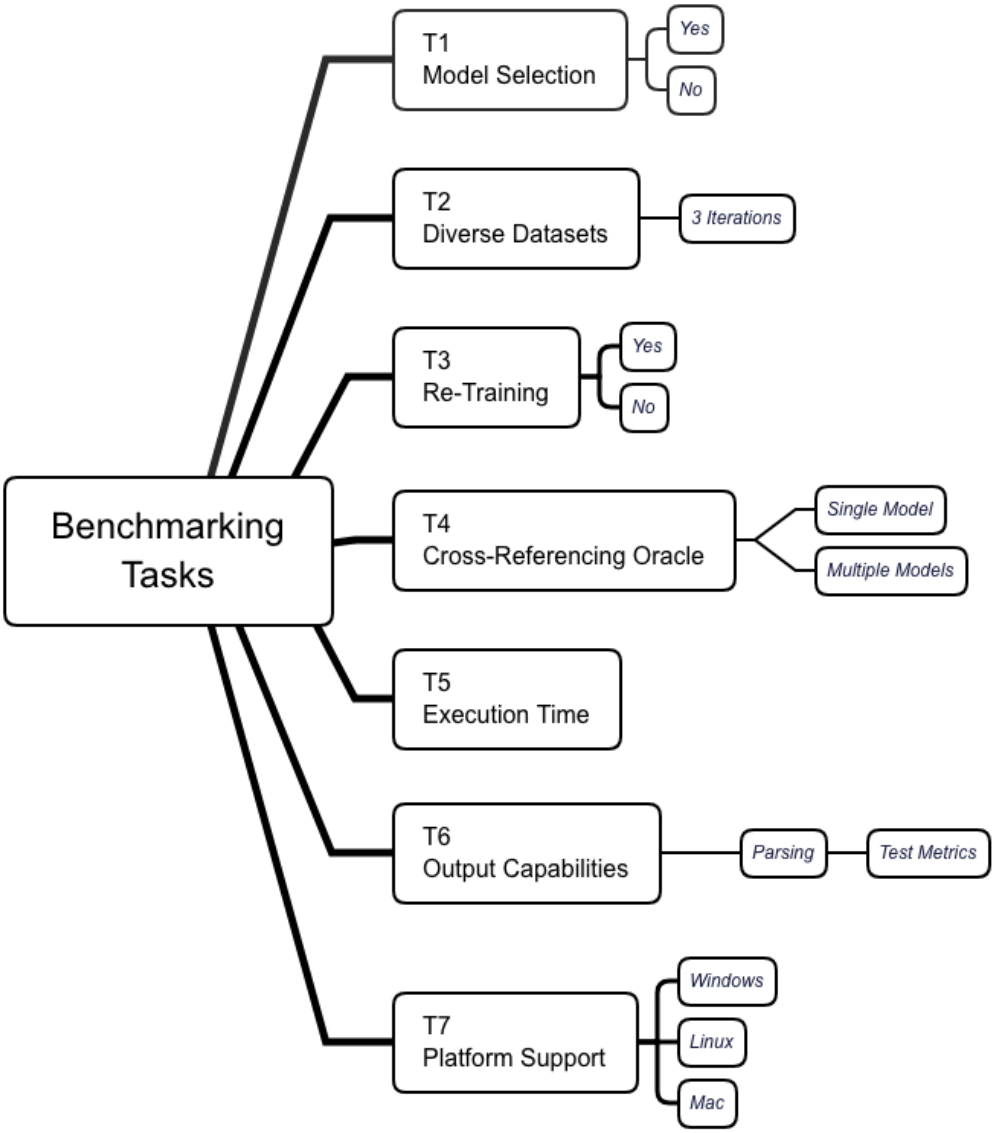

Figure 4.2: Benchmarking Tasks Design

- T1: Model Selection
- T2: Diverse Dataset
- T3: Retraining
- T4: Cross-referencing Oracle
- T5: Execution Time
- T6: Output Capabilities
- T7: Platform Support

T1: Model Selection
Many DL testing tools are currently prototypes or frameworks that demonstrate certain techniques and use fixed models and should be distinguished from the tools that allow the option of selecting your own model to be tested. While fixed model tools are inapplicable to real-world DL systems, the tools provide merits to the field

of DL testing. Due to this, we believe it is important for the benchmark to currently evaluate this initial state of the tool.

T2: Diverse Dataset
Deep Learning systems use a variety of datasets like MNIST and CIFAR-10 for images, Udacity for Autonomous driving, Drebin for malware and text, 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 as to give a sense of its capabilities and focus. Due to the size of the task it was divided into sub-tasks as to provide more informative value. The importance of the information is noted on the scenario description in Section 3.2.2. Table 4.4 shows the division of all T2: Sub-Tasks.

T3: Retraining
As mentioned in Section 3.2.2, some DL testing tools are capable of retaining data that can be used for retraining of the model and improving the model accuracy. Being able retrain the model with the tool and see the model accuracy improvements after retraining can be used as a valuable metric for the tool's performance.

T4: Cross-referencing Oracle
The task will provide information for whether the tool requires a single model or multiple models (differential testing). As mentioned in Section 3.2.2, several DL testing tools require a similar system to be used as a cross-referencing oracle which, while providing benefits to the testing quality, can be difficult to apply. Therefore, the result of this task on its own provides important information for a tool's possible applicability towards certain circumstances, i. e. having no access to a similar second DL system. This task can be seen as trivial to current research as most tools use metamorphic relations rather than cross-referencing. Nevertheless, the research direction of this field is still unknown and the number of publications is growing as can be seen on Figure 2.11. Due to this differential testing approaches may yet flourish and we consider this task important for the benchmark's **Scalability**.

T5: Execution Time
Although it is possible for some tools to find more faulty behavior within a DL model than others, the tools execution time may well be longer than acceptable. As such this metric is an important part of the tool evaluation. Although, due to the immaturity of the field, there is no defined standardized execution time for a DL testing tool, this task would simply provide the total execution time taken by the tool under test, which can be easily compared across the DL testing tools, executed using the same type of resources. 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[3] shows neuron coverage, SADL[10] on the other hand provides a ROC-

AUC score. Also, not every tool is capable of generating output due to a lack of code base and support. Therefore, the benchmarking method would only check for the following 2 things as a part of this task based on execution steps shared by the tools:

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

This would only 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
Many of the tools use plugin combinations like TensorFlow 1.5 with python 2 which is a combination that is currently not available for the Windows operating system. Why it may seem like an obvious task, the results of this task have an important informative value for a practitioner that wishes to use the tools. 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.

Table 4.4: T2: Sub-Tasks

	DataSets Classification	**Model Characteristic (DNN + Architecture)**
T2_1 Images/video	CIFAR MNIST ImageNet	VGG LeNet ResNet
T2_2 Autonomous Driving	Udacity	Nvidia Dave
T2_3 Malware/text	Drebin	Fully Connected

4.3 Benchmarking Design Results

4.3.1 Benchmarking Tasks Automation

Out of all the seven benchmarking tasks identified for benchmarking method, not all of them can be fully automated. The main reason for that is not because of lack of knowledge in developing such a script, but is mainly because of the reason that DL testing tools and techniques presented in this paper are still maturing and don't have a stable working tool yet. All the three working tools namely DeepXplore [3], SADL [10], and DLFuzz [4], which we will be using for demonstrating the benchmarking method have their own way of excuting their scripts to simulate the respective testing techniques. Table 4.5 shows the automation and manual check for each tasks.

Table 4.5: Benchmarking tasks Automation Check

Task	**Automation/ Manual Check**	**Verdict**
Model Selection	Manual * semi-automation possible based on tool's execution command	Yes/No
Diverse Dataset	Automated *Based on status of T2_1, T2_2 and T2_3	Yes/No
Retraining	Manual *semi-automation possible in 3 Iterations	Yes/No
Cross-referencing Oracle	Manual *semi-automation possible based on tool's execution command	Single Model Multiple Models
Execution Time	Automted	Time in seconds
Output Capabilities	Output Save : Automated* Parsing : Manual	Yes/No
Platform Support	Automated	Windows Mac Linux*

The decision for manual and automation check for each task is decided based on our experimental simulation of all the selected fifteen tools in this paper. Due to lack of code and support for execution steps, we could only make there tools working. 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 tools is allowed to use our own model. For instance, we DeepXplore [3] uses it's own fixed models. There is however, one way to detect model selection from the run execution command on the tools if it is allowing to enter our own model but clearly this solution can not be applied for all the tools. Likewise for task such as Retraining and cross-referencing oracle, we can say that we can do semi-automation purely based on the prediction based on the run execution command. But as that is not a generic solution, we would use them as a 'manual' checks for our benchmarking design.

Diverse Dataset task can be automated based on task status of each sub tasks (T2_1, T2_2, and T2_3). Output Capabilities task is just based on the assumption that all the benchmarking tasks are successfully completed and there is a valid output in any format and we are able to save it. The second aspect of this task which is parsing saved logs can be automated based on the input command shared by a DL testing tool for parsing script support for the output of the testing tool. Tasks such as Execution Time and Platform Support can be fully automated.

4.3.2 Benchmarking Algorithm and Script

Algorithm 1 shows steps of the proposed algorithm of the Benchmarking method involving all the seven tasks and sub-tasks as defined in the Figure 4.2 and Table 4.4

Algorithm 1: Procedure of DL Benchmarking method

```
1  initialization
2  Adaptor(postProcessingCommand=FALSE)
   /* Adaptor function to take run command for the DL Testing tool */
3  Configurations()
   /* setup initial Configurations                                 */
4  TasksConfigurations()
   /* setup tasks Configurations                                   */
5  RunConfiguration()
   /* setup run Configurations, import Library and datasets        */
6  ResultConfiguration()
   /* setup result Configurations                                  */
7  while ValidRunCommand do
8      initialization
       /* Prepare Datasets for the DL testing tool                 */
9      if success then
10         task1
11         task2
           /* Task 2 is big and is divided into 3 sub-tasks        */
12         if subtask then
13             subtask1
14             subtask2
15             subtask3
16         task3
17         task4
18         task5
19         task6
20         task7
21         tasksDone = TRUE
22     else
23         exit()
   /* If tasks are successfully done, save the logs                */
24 if tasksDone then
25     SaveLogs()                    // save the test report for analysis
26     if success then
27         DisplayReportPath()
28 else
       /* Check which tasks failed                                 */
29     for tasks do
30         checkTasksStatus();
   /* Now this is a While loop                                     */
31 while postProcessingCommand==TRUE do
32     parseBugReport();                          // Parse the saved logs
```

We will add more details (Flow chart of how json structure will work for benchmarking design scripts etc.) in this section after benchmarking design is finalized.

4.3.3 Benchmarking Results

The results of Benchmarking Results will be filled out after half-report presentation.

5

Threats to Validity

The development of the design method for selection of a DL testing tool and benchmarking the performance measure is a shared effort in the research collaboration. The thesis is intended to be limited to the assessment of three DL testing tool on appropriate datasets.

5.0.1 Threats to internal validity

: Benchmarks have been used in computer science to compare the performance of computer systems, information retrieval algorithms, databases, and many other technologies. The creation and widespread use of a benchmark within a research area is frequently accompanied by rapid technical progress and community building. The identification of relevant performance measures for benchmark is a crucial phase, which means if the measures established in the design step 2 don't cover relevant testing properties (e.g., correctness, robustness, and fairness), benchmarking results won't be of any significance when it comes to selection of appropriate DL testing tools. An early careful identification of relevant measures at the current time would mitigate the issues of wrong measurements before the control experiment. All the tests will be conducted on a common available hardware platform (laptop, SNIC or Chalmers IT setup). The results would risk the portability, which is local to benchmarking designs, and introduce biases. A common hardware (having a fixed number of CPUs and GUPs, subjected to availability of resources) and architecture platform common for all the DL testing techniques would set the common testing ground for benchmarking results.

5.0.2 Threats to external validity:

The benchmarking of the DL testing tool should be both representative and generalizable. Due to a large number of public DL models and diverse datasets, there will be limited control over selection of DL software testing tools and public datasets. To make it generalizable, at least a minimum of three DNN models and three datasets will be selected as input for the tests. Performance measures are also a concern in generalizability. Having insufficient number of performance measures would reduce the applicability of the benchmark. To prevent this, as per stated in the internal validity, a significant amount of time is dedicated towards research and identification of the current measurements. Additionally, we do not take into account predictions of

the direction that measurements of DL testing tools will take due to the expectancy of the research community to iterate and maintain the benchmark in line with the new standards

6

Conclusion and Future Plan

This section will be filled up based on the final benchmarking work after half report presentation.

6.1 Discussion

1. Finish up the remaining work in benchmarking design work.
2. Finalise the json structure of benchmarking tasks based on the proposed algorithm.
3. Script and execution of benchmarking design method.
4. Complete Json and script work for simulation for existing working DL testing tools.
5. Add more details and references for all DL testing scenarios, tasks and sub tasks.
6. Future work.
7. Results and Conclusion sections.
8. Research opportunities in DL Testing
9. References Check.
10. Remaining figures and tables for all the sections.
11. Preparation of the final thesis presentation
12. Masters thesis Writing Seminar.
13. Review of the final document.

Table 6.1 shows the status and progress of each section in this report and also plans for future work after half-report presentation.

Table 6.1: Results and Progress

Section/Task	Status	ETA	Comments
Introdcution	Complete	NA	NA
Related work and Background	Near to Complete	3 Days	Minor details in some sub-sections, cleanup and ´final review
Research Methods	90 %	3-4 Days	Minor work
Validity Threats	Will be done towards the end in the final report%	1 week	Major work
Results	RQ 1 : Research Done. RQ 2 : 40 - 50% RQ 3 : 0%	4 weeks	**RQ1: Done for 4 tools, documentation needs to be done for remaining tools.** **RQ2: Application Scenarios changed to test properties. Industry feedback: Unlikely** **Hypothesis 1: Done, documentation required.** **Hypothesis 2: Based on benchmarking design. Very Unlikely to get Industry feedback.** **RQ3: Possibility to get feedback from industry experts : Unlikely**
Conclusion and Future work	Based on Results	4-5 days	NA

The ETA is an approximate time we expect to complete the tasks. More details and real results of the tasks will be added towards the end of the thesis work. This is done in accordance to the guidelines published in this page : `http://www.robertfeldt.net/advice/master_thesis/feldt_thesis_halfway_questions_and_format.html`

6.2 Conclusion

This section will be filled up towards the end of thesis work in May 2020.

6.3 Future Work

This section will be filled up based on the final benchmarking work after half report presentation, will also include Research Opportunities in DL testing.

Bibliography

[1] Frisk, D. (2016) A Chalmers University of Technology Master's thesis template for LaTeX. Unpublished.

[2] GJ Myers, C Sandler, T Badgett. *The Art of Software Testing* arXiv preprint arXiv:2002.12543 (2020) http://barbie.uta.edu/ mehra/Book1_The%20Art%20of%20Software%20Testing.pdf.

[3] 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).

[4] 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).

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

[6] 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.*

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.*

[12] 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).

[13] 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.

[14] 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 *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (pp. 477-487).

[15] 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.

[16] 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).

[17] 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).

[18] 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).

[19] 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.

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

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

[22] 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.*

[23] 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).

[24] 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.*

[25] 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).

[26] 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.

[27] 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.*

[28] 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.*

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

[30] 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.

[31] Christian, M., Gail, E. K., Lifeng H., and Leon W., 2008 ,. Properties of machine learning applications for use in metamorphic testing. *In SEKE, 2008.*

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

[33] 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.

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

[35] D. Beyer, S. Löwe, and P. Wendler. Reliable benchmarking: Requirements and solutions. Int. J. Softw. Tools Technol. Transfer, 2017. *Springer Link, Published: 03 November 2017.*

[36] M.J. Spendolini, The Benchmarking Book, AMACOM, New York, NY, 1992. *https://www.semanticscholar.org*, DOI:10.4324/9780080943329Corpus ID: 107507540.

[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