Software Testing of Machine Learning Systems

Yamen Albdeiwi
Lund University
Lund, Sweden
mo4718al-s@student.lu.se

Max Fogwall
Lund University
Lund, Sweden
ma4782fo-s@student.lu.se

Emil Friberg
Lund University
Lund, Sweden
em5435fr-s@student.lu.se

Emil Eriksson
Lund University
Lund, Sweden
em5184er-s@student.lu.se

***Abstract*—In this paper, we conduct an exploratory study to investigate methods of, as well as reasoning for, using software testing principles to verify the behavior of machine learning systems. We find that traditional testing techniques are not always applicable, and those that are often have significant differences. Approaches and solutions to issues arising from these differences are discussed. We conclude that testing of machine learning systems is very different from that of traditional software, and that there are many challenges yet to be solved in this area.**

I. Introduction

In recent years, Machine Learning (ML) as a field of research for Artificial Intelligence (AI) has made significant progress, reaching and sometimes even surpassing human performance for some tasks. As a result, it has seen widespread adoption within the software industry, enabling and facilitating a variety of applications, such as self-driving vehicles, face and speech recognition, as well as malware detection [1].

With AI systems being crucial for many safety-critical domains such as robotics, automotive, and security, the predictability and correctness for rare inputs and edge cases are vital. ML systems, that is to say systems with a trained ML model as component, are not very well suited to this. This is due to the correctness in behavior being limited to the training data for the model, which is unlikely to cover most real-world data. The use of such systems in practice can therefore easily lead to disastrous consequences if not properly tested [2, 1].

An ML system in a self-driving car seeing a snowstorm for the first time, for example, will likely have undefined behavior which may be harmful to its passengers and surrounding traffic. Pei et al. found thousands of similar corner cases in state-of-the-art ML software in their DeepXplore paper, including malware being treated as benign software and cars crashing into guard rails [1].

Because of this, testing and quality assurance of ML systems are of great importance. The problem, however, is that unlike traditional software, the logic of an ML model is hidden and its structure dynamically changing, meaning traditional testing techniques are not always applicable [2].

A. Description

In this paper we conduct an academic literary synthesis to analyze and discuss the importance of testing ML systems and how that differs from traditional software testing. This is done from the perspective of the Software Testing (ETSN20) course at the Faculty of Engineering at Lund University (LTH).

The reason we chose this area is that it is currently highly relevant to the software industry and our course, and a well established yet growing area of research.

This paper aims to answer the following questions:

- Why is it important to test trained ML models at all?
- How do traditional software and ML differ in the context of software testing?
- What are a few examples of approaches and challenges related to testing ML systems?

B. Limitations

In this paper we focus specifically on the reasons for and various approaches to testing ML systems, as well as how that differs from traditional software testing. As such, we do not go into detail about how ML works, nor do we explain how traditional software testing works. While we do bring up a few examples of challenges in testing ML systems, we do not discuss the viability, differences, or properties of approaches beyond what is required to grasp their concepts.

II. Analysis

A. Importance of ML testing

Recent breakthroughs in Deep Learning and Reinforcement Learning have made machine learning increasingly useful for large-scale software systems. Many areas of our lives depend on machine learning applications, as mentioned in the introduction. The impacted domains additionally include medicine and finance [3], where the correctness of the system is not only crucial for business decisions, but also lives.

When training ML models, the programmer does not implement and design components like with traditional software. Instead, the programmer provides sample input data and scores the output based on the desired outcome, with the machine working out the underlying logic to maximize that score [2]. Without testing it is therefore difficult to ensure that the underlying logic is consistently applied in varying environments, and that the system is safe and stable enough to be used in the real world.

There are multiple ways the aforementioned correctness of behavior can be ensured through testing. One way is changing the input to values that should yield an equivalent output and ensuring the system is consistent as such. Another is employing another ML system as an adversary to the original and training the two in tandem. These approaches

are discussed in greater detail in sections II-C (**Solutions**) and II-D (**Approaches**) respectively.

B. Key Differences Between ML Systems and Other Software

1) Short Introduction to ML Systems and DNN: To understand the differences between ML systems and other software, one must understand the fundamental structure of an ML system. In this paper, ML and Deep Neural Networks (DNN) are used interchangeably, but the DNN structure will be the foundation for the following sections. In essence, a DNN consists of multiple *layers*, each consisting of multiple *neurons*. A neuron is the smallest computational unit in a DNN, and applies an *activation function* on its inputs and passes the output to multiple other neurons. Connections between neurons all carry different amounts of weight, and each layer transforms the input data to a higher level representation [1].

DNN layers have three distinct types: Input, output, and hidden. A DNN may have one input layer and one output layer, but often many more hidden layers transforming the data. Mathematically, a DNN may be seen as a multi-input, multiple-output parametric function consisting of many more parametric functions [1]. The weight of neuron connections, and their function, are generally defined by "teaching" the program, giving input data and expected output, and letting neurons and their connections change until the desired output is met. An example DNN can bee seen in Figure 1.

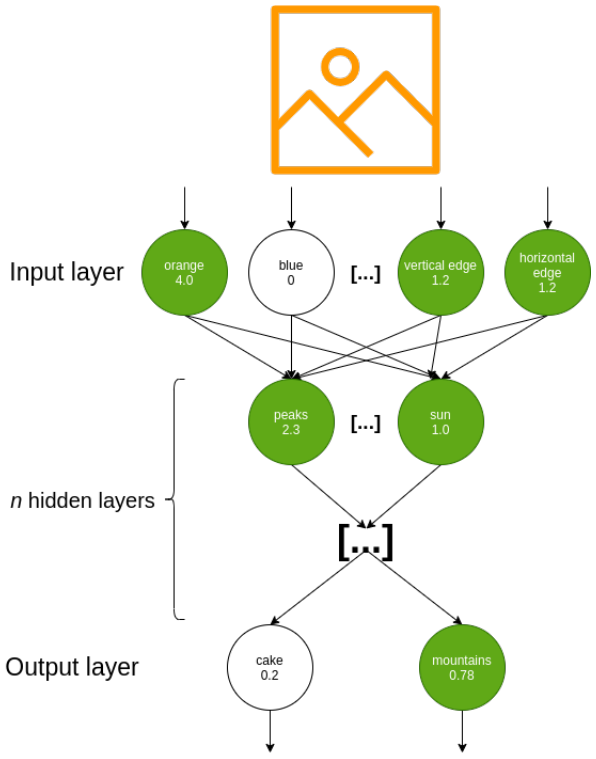

Fig. 1. Example of a DNN. Each circle is a hypothetical neuron with the certainty of which they believe the input data is what they represent. Green neurons have reached their threshold value.

This differs much from classical software, where the writer of the software often decides a desired set of instructions to transform some input to some desired output.

2) Code Coverage: It is in the nature of ML systems that the developer has a less direct impact on the program itself. While the behavior of a classic software product is directly controlled by the programming of its author, this is not the case for ML systems. Instead, the developer "guides" the software to desired behavior by influencing the training data, feature selection and algorithm tuning. The impact on decision logic is indirect [1].

These differences have serious impacts on the possibility of testing, and especially white-box testing. While code coverage may provide testers with knowledge of to which extent classic software is tested, the same can not be said for ML systems. A code coverage of 100% in the case of ML can be trivial to achieve, but this does not accurately describe the behavior of ML systems [1].

3) Datasets: The use of data in relation to ML systems is central, and an important factor in testing ML systems. MoleculeNet is an example of a benchmarking tool for ML systems used in chemistry. MoleculeNet builds upon a range of different datasets, and the authors also provide suggested splits for these datasets[4]. Splits are an important part of ML systems, and required to benchmark algorithms. Datasets are split into three parts; training, validation and test subsets. This is also the case for MoleculeNet. This does however raise a concern. If tests, validation and training relies on the same dataset, the requirements for a dataset is very important. MoleculeNet in particular reveals that the split of a dataset also is important for the performance. This is due to the underlying data and intended applications; In the case of MoleculeNet, separating structurally different molecules into different subsets offers a greater challenge for the learning algorithms [4].

This problem is further increased by feedback loops in the ML product. Even small bugs in a dataset can become big faults, and generated big datasets can often be very hard to visualize [5]

4) The Oracle Problem: A major difference between ML testing and classical software testing is the so-called *Oracle Problem*. A *Test Oracle* is something that will act as a source of truth, to provide the answer to the question of "What is expected behavior?". For ML systems, this problem is in no way trivial. After all, if there were a way to know the solutions to problems solved by ML systems, there would be no need for these systems [5]. As will be explored in Section II-C, there exists proposed solutions to this problem, but the problem is currently subject to further study.

C. Possible Solutions to ML-specific Problems

1) Neuron Coverage & Differential Testing: The problems as presented above may have some solutions to solve or at least ease the testing of ML products. One such solution may be to define new criteria for the testing of ML systems [6]. Some of these new criteria relate to neuron coverage, that is, to check

how many of the neurons in an ML system is reaches their activation threshold for the input provided. This is analogous to code coverage for classical systems, as neurons provide the building blocks for an ML systems to make decisions. Due to the indirect nature of ML systems development, it is not as straightforward to increase neuron coverage as it is code coverage.

A recent development in this area has proven to make it easier to detect errors in a white-box environment, using neuron coverage as a metric. It has been seen that each neuron in a Deep Neural Network (DNN) tends to learn a different set of rules than its peers, meaning that testing more neurons would test more rules used by the DNN. It should also be noted that code coverage is a much less useful metric in measuring the correctness of an ML system than neuron coverage. However, reaching this neuron coverage is much less straightforward than code coverage. Tools such as *DeepXplore* have proven to increase a models' correctness by as much as 3%, but this still leaves room for error [1].

DeepXplore also provides a possible solution to the Oracle Problem by using similar models to cross-reference the model being tested [1]. This differential testing technique, however, comes with its own set of problems. As *DeepXplore* can only find erroneous behavior when models differ, it relies on the models to be different enough to be effective. It can also be noted that models cannot be tested in a vacuum, that is, without models for the same target.

2) Metamorphic Testing: Another proposed of solving the oracle problem is to use metamorphic testing. In metamorphic testing, the program is tested by changing the input values in ways that should make no difference for output (for example, decreasing light in an image) and expecting the ML systems to react in the same way as the unaltered input [5]. One can divide metamorphic categories in two categories; *Coarse-grained Data Transformation* and *Fine-grained Data Transformation*. A coarse-grained approach does not change the value of any data cells, but may enlarge the dataset or change the order of data. Fine-grained data transformation instead work with smaller changes on the dataset, such as changes of labels or pixels. One such fine-grained approach were able to identify 71% of injected bugs in an ML system [7].

3) Smoke Testing: All methods for testing ML does not have to be new, groundbreaking methods. In the article "Smoke Testing for Machine Learning" [8] the authors try to use standard software techniques on ML systems to try to come up with generic and simple smoke tests. A smoke test is defined as "A test suite that covers the main functionality of a component or system to determine whether it works properly before planned testing begins." [9], i.e. it focus on the most crucial functions but does not bother with the details. In their paper, [8] the authors defines 37 smoke tests for classification or clustering algorithms in ML. These tests showed that equivalence class coverage in combination with condition coverage can achieve coverage of hyper-parameters for testing linear growth. They also showed that textbook methods can be useful in ML testing, and with some modification it can become effective tests for ML libraries.

D. Approaches in ML Software Testing

The possible solutions to ML-specific problems introduced in Section II-C provide some clarity when coming from a traditional testing environment, but what are the more general approaches in the testing of ML systems, and how do they compare to the traditional testing approaches? This section uses the well-known terms *White-box* and *Black-box* to investigate more ways ML systems can be tested, as well as introducing some new approaches.

In theory, it would be possible to apply some concepts from traditional software testing to test machine learning-based systems, but these systems present a range of challenges not faced by traditional software systems, making traditional testing techniques unfit for the task. Non-determinism, an inherent characteristic of ML, poses the biggest challenge to testing [10]. However, as of late, scientists have begun to foster new testing methods to help ML designers identify troubleshoot and test ML programs. We will focus on approaches that assume that models are implemented into programs without error, and focus on providing tools to detect expected errors in model alignment.

1) Black-box Approaches: Black-box approaches are experimental methods that do not require access to the internal execution subtleties of the model under test. The point of these methods is to guarantee that the model under test predicts the objective value with a high degree of accuracy, without worrying about its internal, learned limitations. The common factor for these approaches is the adversarial dataset generated to test the ML models. These approaches rely on statistical analysis techniques to construct a multidimensional arbitrary interaction that can produce information with similar measurable attributes as the input data. All the more explicitly, they build generative models that can fit a likelihood circulation that best portrays the information.

These generative models permit to test the likelihood circulation of information and create as numerous items depending on the situation for testing the ML models. In adversarial machine learning, see Figure 2, a series of adversarial examples is used to determine whether or not a machine learning model is robust by making small changes to the input pixels, applying spatial transformations, or by simply guessing and checking to detect misinterpreted [10]. In fact, analyses of adversarial evasion in recent years [11] have shown that DNN have weaknesses that are exposed by adversarial moves, see Figure 1.

2) White-box Approaches: Mutation Testing is a form of White-box testing, which includes changing components of an application's source code so that a software test suite can detect the changes. Mutation testing comprises in infusing mutants into a program under test and create test cases to distinguish them, see Figure 3. DeepMutation was developed by Ma et al.[13] based on a set of source-level mutation operators that are used to insert bugs into the source code of a machine learning program. It is the purpose of data

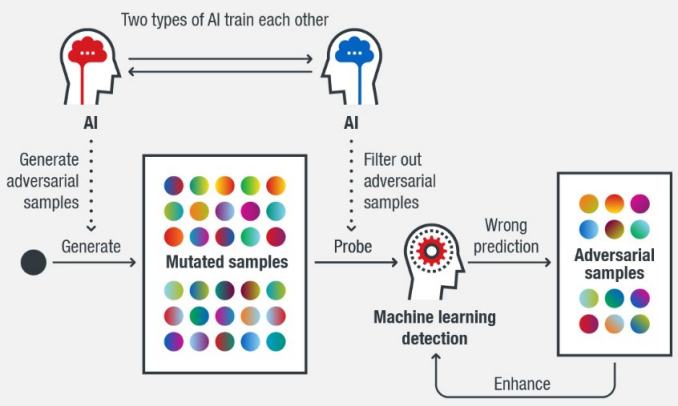

Fig. 2. Adversarial samples can help identify weaknesses in an ML model [12]

mutation operators to introduce possible data-related errors that are likely to occur during data acquisition, data cleaning, and/or data transformation. However, in advanced Deep Learning systems, the learning of the model parameters may take dozens of hours or even days, since the parameters of the model must be taught from scratch after source-level mutations [3]. Mutation testing evaluates the validity of the test data and indicates their shortcomings utilizing assessment measurements identified with the quantity of mutant models killed, after the mutant models are created.

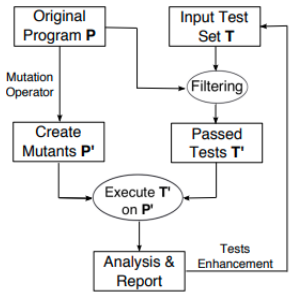

Fig. 3. Key process of general mutation testing [13]

3) Property-based Testing Approach: Property-based testing is the process of evaluating a computation by inferring its properties using the theory and formulating invariants the code must satisfy. By using this approach, ML code implementation errors are detected. Testing is conducted by generating and executing test cases based on formulated invariants at every stage of the computation to identify any errors. Property-based tests can be used to ensure that the probability laws hold throughout the execution of a model [3].

4) Proof-based Testing Approach: Proof-based testing is a strategy which utilizes numerical verification procedures to deteriorate a complicated detail that is hard to test straightforwardly into more modest properties that together coherently suggest the first particular, however are simpler to test. Selsam et al.[14] proposed to officially indicate the calculations of ML projects and build formal evidences of composed hypotheses that characterize what it implies for the projects to be right and error-free. Selsam et al. examined an ML system intended for enhancing over stochastic calculation charts, using the intelligent evidence associate Lean, and detailed it to be bug-free. Nonetheless, albeit this methodology permits to distinguish faults in the numerical plan of the calculations, it can't assist with recognizing execution mistakes because of mathematical insecurities, like the substitution of real numbers by floating-point with limited precision [3].

III. Reference Usage

In this paper, several sources are presented to collect information about the current state of the testing of ML systems. It shall be noted that while all references have been selected based on quality, some have been accessed through *arXiv*, which does not guarantee peer reviews [15]. It should be noted that this does not automatically mean the quality of these articles are inferior, nor that they will not be subject to peer in the future.

It should also be noted that the subjects discussed in this paper is under constant development, and that the literary synthesis provided in this paper may not provide a full coverage of the field, but it does provide insight in recent developments. Care has been taken to select recent sources, but due to the significant increase of articles in this field, it is possible that the facts presented may soon have to be updated. This becomes especially apparent when considering that 85% of publications regarding ML testing between 2007 and 2019 was published between 2016 and 2019 [5]. If this trend continues, many more insights will become apparent during the coming years.

IV. Conclusion

A. The importance of testing ML models

It is important to test ML systems for a number of different reasons. When it comes to ML systems where it affects the physical world directly, like self-driven cars, it is important that the system is predictable. If the expected behavior of the system cant be tested for specific situations, then it cant be used in systems where the outcome could be critical.

B. Difference in traditional Software and ML testing

ML systems differ from traditional software in a lot of ways, including when it comes to how to test it. A DNN consists of multiple layers that take in data and change the input in different ways until it matches the expected output. The developers have limited influence over these layers since they are configured by the learning algorithm. This makes it very hard to test if these layers have been trained correctly, and a problem called the oracle problem arises from this. The Oracle problem states that the expected behavior is needed if the current behavior is to be evaluated, and if the expected behavior is known the ML system is not needed in the first place. Another problem is that this also makes it a lot harder to conduct white-box testing. Furthermore code coverage does not work the same way that it does with traditional

software. While it is easy to reach high code coverage, it does not describe the behavior accurately. The data used to train a model is also very important and influences the final product considerably. Usually the test data is split into training, validation and test subsets. This means that if the same data is used for validation and testing as when the model is trained a lot of errors will not be detected. ML systems can also suffer from feedback loops which can make small bugs in the test data lead to big faults with the trained model.

C. Approaches and solutions for testing ML systems

As mentioned, the testing of an ML system differs a lot from traditional software testing, which of course come with their own challenges and solutions. This paper studied Neuron Coverage and Differential Testing, Metamorphic Testing, and Smoke testing as three possible solutions to the different challenges. Neuron Coverage is an replacement to Code Coverage in traditional software testing, and tools such as *DeepXplore* can be used, where it could increase a models' correctness by as much as 3%. *DeepXplore* can also be used to possibly solve the Oracle Problem by using a Differential Testing techniques, and cross-referencing similar models to the one being tested. However, Differential Testing has a few criteria that needs to be filled for it to work. Models for the same target is needed to test against, and the models need to be different enough for the testing algorithm to find where it differs.

Metamorphic testing is another possible solution to the Oracle Problem, which changes the input values in a way which should not affect the output values in the ML system. Lastly, Smoke Testing was investigated, which use standard software testing techniques to try to find errors in the ML systems. It focus on the most crucial functions instead of the details. They also showed that textbook methods with some modification can be useful in ML testing.

The different approaches in testing were also investigated in correlation to ML testing. Black-box, White-box, Property-based Testing, and Proof-based testing approaches were studied. Black-box is used to investigate the accuracy of the ML system in adversarial ML systems. For White-Box testing, the focus was on Mutation testing, and specifically the tool *DeepMutation*, which can be used to investigate the validity of the test data by introducing mutations in the system. Property-based Testing can be used to find implementation errors in ML code, and ensure that the probability laws hold. Lastly, the Proof-based Testing Approach, based on numerical verification procedures, can be used to give more straightforward properties that are easier to test by deteriorating complicated details. However, this can only distinguish errors, and not recognize execution mistakes.

V. Contributions

A. Yamen Albdeiwi

- Helped with the outline as well as added some references.
- Wrote part of the **Importance of ML testing**.
- Wrote the **Approaches in ML Software Testing**.
- Added Figures 2 & 3.
- Collected multiple references.
- Fixed some issues with references.

B. Max Fogwall

- Helped with the outline of the document and wrote brief comments on what sections should include.
- Helped with formatting and expansion of **References**.
- Wrote the **Abstract**.
- Wrote the **Introduction**, and its subsections.
- Rewrote most of the **Analysis** $\rightarrow$ **Importance** section.

C. Emil Friberg

- Helped with the outline and added some references.
- Wrote parts of the **Importance of ML testing**.
- Wrote **Smoke Testing** in **Solutions**.
- Wrote the **Conclusion**.

D. Emil Eriksson

- Helped with the outline
- Wrote parts of the **Analysis**, namely II-C (excluding II-C3) and II-B.
- Wrote III.
- Created Figure 1.
- Collected multiple references.

References

[1] J. Y. K. Pei Y. Cao and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," ser. SOSP '17, Shanghai, China: Association for Computing Machinery, 2017, 1–18, ISBN: 9781450350853.

[2] Q. Song, E. Engstrom, and P. Runeson, "Concepts in testing of autonomous systems: Academic literature and industry practice," English, ser. Proceedings - 2021 IEEE/ACM 1st Workshop on AI Engineering - Software Engineering for AI, WAIN 2021, United States: Institute of Electrical and Electronics Engineers Inc., 2021, pp. 74–81.

[3] H. B. Braiek and F. Khomh, "On testing machine learning programs," *Journal of Systems and Software*, vol. 164, p. 110 542, 2020, ISSN: 0164-1212.

[4] Z. e. a. Wu, "Moleculenet: A benchmark for molecular machine learning," *Chem. Sci.*, vol. 9, pp. 513–530, 2 2018.

[5] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, *Machine learning testing: Survey, landscapes and horizons*, 2019.

[6] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks," *ArXiv*, vol. abs/1803.04792, 2018.

[7] A. Dwarakanath *et al.*, "Identifying implementation bugs in machine learning based image classifiers using metamorphic testing," *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[8] S. Herbold and T. Haar, "Smoke testing for machine learning: Simple tests to discover severe defects," *ArXiv*, vol. abs/2009.01521, 2020.

[9] I. Glossary, *Smoke test*. [Online]. Available: https://glossary.istqb.org/en/search/smoke%20test.

[10] D. Marijan, A. Gotlieb, and M. Kumar Ahuja, "Challenges of testing machine learning based systems," in *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, 2019, pp. 101–102.

[11] S. Gu and L. Rigazio, *Towards deep neural network architectures robust to adversarial examples*, 2015.

[12] L. Huang, A. D. Joseph, B. Nelson, B. I. Rubinstein, and J. D. Tygar, "Adversarial machine learning," in *Proceedings of the 4th ACM workshop on Security and artificial intelligence*, 2011, pp. 43–58.

[13] L. Ma *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2018, pp. 100–111.

[14] D. Selsam, P. Liang, and D. L. Dill, "Developing bug-free machine learning systems with formal mathematics," in *International Conference on Machine Learning*, PMLR, 2017, pp. 3047–3056.

[15] *About arxiv*. [Online]. Available: https://arxiv.org/about (visited on 12/13/2021).