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 overwhelming 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. However, with ML-systems being adaptive and heavily reliant on training data reflecting real-world data to succeed, this can easily cause disastrous consequences [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. find 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 components of an ML system are 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.

In this paper we seek to answer the following questions:

- Why is it important to test ML systems 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?

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.

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.

The reason traditional software testing techniques, such as the division into unit, integration, and system tests, do not extend to ML systems is that the latter logic does not follow system designs and implementations. Instead of implementing and designing the components of the system, 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]. Therefore, tests of ML systems typically need to ensure that this learned logic remains consistent in varying environments, and that the system is safe and stable enough to be used in the real world.

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 often has three distinct types of layers: 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.

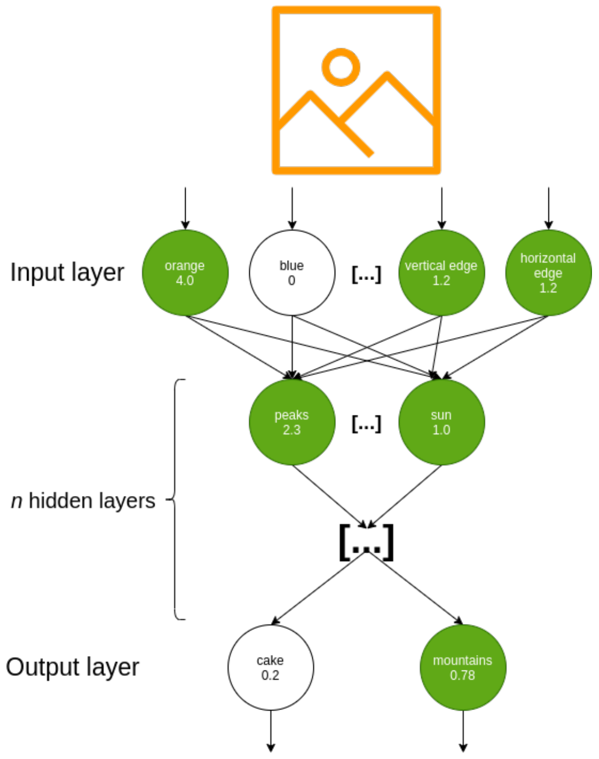

Fig. 1. Example of a DNN. Each circle is a hypothetical neuron with the certainty of which they believe the input data is what they represent. Green neurons have reached their threshold value.

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 software 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 software. 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 has 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 software. 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 software [1].

3) Datasets: The use of data in relation to ML software is central, and an important factor in testing ML software. MoleculeNet is an example of a benchmarking tool for ML software focusing on 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 [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 software [6]. Some of these new criteria relate to neuron coverage, that is, to check how many of the neurons in a 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 a ML software to make decisions. Due to the indirect nature of ML software 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 ML product. It should also be noted that code coverage is a much less useful metric in measuring the correctness of a ML system than neuron coverage [1]. 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.

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 software 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 a ML program [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] they try to use standard software techniques on ML systems to try to come up with generic and simple smoke tests. In other words they want to use standard test to find bugs and improve the quality off ML systems. One example of this is to use equivalence Classes and boundary-value analysis.

D. Approaches in ML Software Testing

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 [9]. 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 which accept that the models are carried out into programs without blunders, and concentration on giving instruments to distinguish expected mistakes in the alignment of the models.

1) Black-box Approaches: These 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 often worrying about its internal, learned limitations. The common factor for these approaches is the adversarial dataset generated to test the ML models. These approaches depend on statistical analysis techniques to construct a multidimensional arbitrary interaction that can generate 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 set of adversarial examples is used to determine whether a machine learning model is robust or not, by making small modifications to the input pixels, applying spatial transformations, or simple guess-and-check to find misclassified [9]. In fact, analyses of adversarial evasion in recent years [10] have shown that deep neural network models *DNN* have been shown to have weaknesses revealed by adversarial movements, see Figure 1.

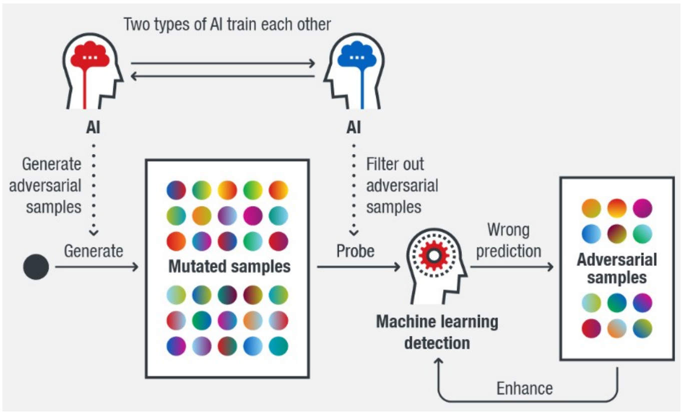
Two types of AI train each other
AI
AI
Generate adversarial samples
Filter out adversarial samples
Generate
Mutated samples
Probe
Wrong prediction
Adversarial samples
Machine learning detection
Enhance

Fig. 2. Adversarial samples can help identify weaknesses in an ML model [11]

2) White-box Approaches: TODO: here we will expand **Neuron Coverage & Differential Testing** section.

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.[12] 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 [3]. TODO: to be continued.

III. Conclusion

TODO

IV. Contributions

A. *Yamen Albdeiwi*

- Helped with **Points Of Interest** as well as added some references.
- Some comments have been added about what will be discussed in the results section.
- Wrote part of the **Importance of ML testing**.
- Wrote the **Approaches in ML Software Testing**.
- Added Figure 2.
- 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**, including its **Description**.
- Rewrote most of the **Analysis** → **Importance** section.

C. *Emil Friberg*

- Helped with **Points Of Interest** as well as added some references.
- Wrote parts of the **Importance of ML testing**.
- Wrote **Smoke Testing** in **Solutions**.

D. *Emil Eriksson*

- Helped with **Points Of Interest** as well as added some references.
- Wrote parts of the **Analysis**, II-C (except II-C3) and II-B
- 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] 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.

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

[11] 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.

[12] 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.