Manifold-based Testing of Machine Learning Systems

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Tae Joon Byun

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
Doctor of Philosophy

Dr. Mats P.E. Heimdahl and Dr. Sanjai Rayadurgam

August, 2022

© Tae Joon Byun 2022
ALL RIGHTS RESERVED

Acknowledgements

I would like to express my deepest gratitude to my advisor, Professor Mats P.E. Heimdahl, and my co-advisor, Professor Sanjai Rayadurgam, for advising me closely throughout my entire Ph.D. journey. It would have been impossible to complete my research or grow as an independent researcher without their leadership and caring support. I would also like to thank Dr. Darren Cofer at Collins Aerospace and Dr. Michael W. Whalen at Amazon Web Services for having me work on the Assured Autonomy project as an intern, which turned into a three-year research presented in this dissertation. I also appreciate the dissertation committee members for sharing their committment and guidance on completing my dissertation, and I thank the professors and peers I met in the department whom I have worked with closely—Professor Stephen McCamant, Dr. Vaibhav Sharma, and Abhishek Vijayakumar, more than anyone else. Finally, I send my sincere gratitutde to my M.S. advisor Professor Yunja Choi, who stood as my role model of computer scientist, who taught me so much about Software Verification and inspired me to follow her foot steps, and to my old friends and peers Mingyu Park and Hoon Jang, who initially introduced me the joy of proramming and the field of Software Engieering research. My career would not have existed as it is without each one of these people.

Dedication

To my lovely wife Yeonjoung and my little daughter Lesley.

Abstract

With the remarkable advancement of deep learning in many domains, such as in computer vision, learning-enabled systems are rapidly being adopted in safety-critical domains where it is crucial to verify and validate the system rigorously. However, due to the unique characteristics of the learning-enabled components compared to traditional systems, existing verification techniques do not work in many cases, which calls for new approaches to address this problem. In the literature, we identified that a practical and scalable testing technique is lacking for computer-vision deep neural networks (DNNs) that deal with high-dimensional and unstructured input data. Moreover, most of the existing approaches for addressing this problem are white-box solutions that are dependent on the DNN under test, solutions that may be inappropriate given the highly iterative model development workflow. To address this problem, we propose systematic testing techniques for DNNs that resolve the dependency on the model under test, since the dependency comes with several critical shortcomings. In doing so, we investigated the following three concrete ideas. First, we propose a test prioritization technique that can identify failure-revealing test inputs to help reduce the test construction cost. Second, we propose a DNN-independent test adequacy measurement technique that can measure the adequacy of testing, and also help construct a representative test suite. Third, we propose a DNN-independent test case generation technique that can synthesize realistic test cases that are effective at finding failures in the DNN under test. The last two approaches are black-box solutions in that the test adequacy measurement and the test case generation are performed independently of the DNN under test, a unique direction compared to existing approaches. The experiments showed that (1) test prioritization can effectively prioritize failure-revealing test cases, (2) the black-box coverage criterion can help construct representative test cases that achieve effectiveness comparable to those constructed with white-box criteria, but with much lower measurement cost, and (3) the black-box test generation can synthesize realistic test cases that are also effective at finding failures in the model under test. We believe that the black-box approaches bring complementary benefits to white-box approaches and that they deserve further investigation.

Contents

Acknowledgements i

Dedication ii

Abstract iii

List of Tables vii

List of Figures viii

1 Introduction 1

1.1 Objectives and Contributions 4

1.2 Structure of this document 6

2 Background and Related Work 8

2.1 Software Verification and Validation 8

2.1.1 Formal Verification 10

2.1.2 Software Testing 12

2.2 Preliminaries of Machine Learning 18

2.2.1 Deep Learning 21

2.2.2 Machine Learning Workflow 23

2.3 Verification and Validation of Learning-Enabled Systems 25

2.3.1 Formal Verification of Neural Networks 27

2.3.2 Machine Learning Testing 28

2.3.3 Desired Properties 37

2.4 Summary 42

3 Test Input Prioritization **43**

3.1 Sentiment Measures 44

3.1.1 Predicted Confidence 44

3.1.2 Bayesian Uncertainty 45

3.1.3 Input Surprise 48

3.2 Experiment 50

3.2.1 Systems Under Test 51

3.2.2 Model Configuration 51

3.2.3 Measure of Effectiveness 53

3.2.4 Implementation and Experiment Environment 54

3.3 Result 55

3.3.1 RQ1: Effectiveness of prioritization in finding failures 56

3.3.2 RQ2: Effectiveness of prioritization in retraining 60

3.3.3 Threats to Validity 63

3.4 Conclusion 64

4 Black-Box Testing for Deep Neural Networks **65**

4.1 Representation (Manifold) Learning 66

4.2 Variational Autoencoder 68

5 Black-Box Coverage Criterion **72**

5.1 Limitations of DNN White-Box Coverage Criteria 73

5.2 Manifold Combination Coverage 75

5.3 Evaluation Criteria 77

5.4 Experiment 79

5.4.1 Test Suite Construction 80

5.4.2 Datasets and Models Under Test 80

5.4.3 Experiment Configurations 81

5.5 Result 83

5.5.1 RQ1: Failure-Revealing Effectiveness 87

5.5.2 RQ2: Semantic Balance 88

5.5.3 RQ3: Retraining Efficacy 88
5.5.4 Discussion 89
5.5.5 Threats to Validity 90
5.6 Conclusion 90

6 Manifold-based Test Case Generation 92
6.1 Test Generation from Manifold 93
6.1.1 Considerations for Testing Image Classifiers 94
6.1.2 The Design of the Test Generator 96
6.1.3 Search-based Test Generation on Manifold 104
6.2 Experiment 109
6.2.1 Tasks and the Models Under Test 111
6.2.2 Training VAE Models 112
6.3 Result 114
6.3.1 Quality of the Images Generated By VAEs 114
6.3.2 RQ1: Can manifold-based test generation generate realistic test inputs? 117
6.3.3 RQ2: Can manifold-based test generation generate failure-revealing test cases? 118
6.3.4 RQ3: Can manifold-based test generation assign labels accurately? 121
6.3.5 Is search better than random sampling? 122
6.3.6 Discussion 122
6.4 Conclusion 123

7 Conclusion 125

References 127

List of Tables

2.1 Comparison between traditional software testing and ML testing 29
3.1 Four digit classification models 52
3.2 The efficacy of test input prioritization in finding failures 59
3.3 The efficacy of input prioritization in retraining 62
5.1 Models under test 81
5.2 The effectiveness of coverage criteria 85
5.3 Failure-revealing effectiveness of test suites constructed by each coverage criterion 86
6.1 Trained VAEs 113
6.2 The quality of the generated test cases 117
6.3 Failure-revealing test cases 119
6.4 The failure-revealing effectiveness of the generated test cases 120

List of Figures

2.1 The V Model 9
2.2 Formal verification procedure 10
2.3 Bias-Variance Trade-off 21
2.4 The nine stages of machine learning workflow 23
2.5 Category of Input Data 39
3.1 Cumulative sums of the errors found by prioritized test suites 54
3.2 The cumulative sum of revealed failures in prioritized suites for MNIST models 56
3.3 The cumulative sum of revealed failures in prioritized suites for TaxiNet Models 57
3.4 Representative inputs of high vs. low priority 58
4.1 Variational Autoencoder 70
5.1 Coverage analysis on 2D Manifold Space. 76
6.1 Variational Autoencoder (VAE) and a Manifold-Based Classifier 99
6.2 Conditional Variational Autoencoder (CVAE) and a Manifold-Based Classifier 99
6.3 Two Stage VAE 100
6.4 The Manifold-based Test Case Generator with CVAE 101
6.5 Synthesized Inputs Over A 2D Manifold 104
6.6 Uncertainty of the Manifold-Based Classifier Over a 2D Manifold 106
6.7 Uncertainty and synthesized inputs overlayed on 2D manifold 107
6.8 Images reconstructed by trained VAEs 115
6.9 Images generated by Two-Stage VAE 116
6.10 Failure-revealing test cases for MNIST 121

Chapter 1

Introduction

In recent years, machine learning (ML) has shown remarkable achievements in many domains such as computer vision and natural language processing. Most of these achievements were made possible by deep learning [1], which is a method of machine learning that uses deep neural networks (DNN), or computing systems inspired by biological neural networks, as its model of computation. Empowered by its versatility, more and more software systems are designed to contain these learning-enabled features (we refer to such a software component as a learning enabled component, or LEC for short), and LECs are even migrating into domains where safety is critical. Examples can be found in medical diagnostics, authentication, self-driving cars, and unmanned aerial vehicles, to name a few. As any failure in such systems can cause severe loss and damage to human lives and properties, these systems must be scrutinized thoroughly for safety through rigorous verification and validation (V&V).

V&V of software systems have never been easy problems to address because the complexity of the software systems has been increasing faster than our capability to address it [2]. The problem has only intensified with the rapid advancement and adoption of machine learning [3–5]. Most notably, DNNs are notoriously uninterpretable as they consist of millions to billions of trainable parameters that are tuned algorithmically. Formal techniques can perform sound—and sometimes complete—verification of logical properties on DNNs either through linear programming optimization or reachability analysis[6], but they only scale to DNNs of minuscule size. Testing, on the other hand, aims at exercising a subset of the system behavior such that defects are discovered and

fitness of the system—i.e., safety, robustness, etc.—is objectively assessed; and as a result, evidences can be gathered towards the assurance of the system. Although testing cannot achieve completeness in theory, principled approaches can effectively find errors by focusing on an *important* subset of the system behavior. Given the ever-increasing complexity of the DNNs, testing often remains as the only viable V&V approach for production-scale LECs composed of DNNs.

When it comes to testing traditional safety-critical systems, standards and practices are established to ensure that testing is performed adequately. Standards in avionics such as DO-178C [7], for instance, requires that a certain level of completeness is achieved in testing, through a measure of test adequacy—such as Modified Condition and Decision Coverage—defined over the structure of the source code. However, this measure is inapplicable to LECs that are composed of DNNs since they lack explicit control logic. Likewise, the unique characteristics that accompany DNNs, especially those that perform computer-vision tasks, pose multiple challenges to existing practices of testing. Most notably:

1. **High-dimensional and unstructured data:** The input space of the tasks that a typical LEC deals with—such as image classification—is highly unstructured and high-dimensional, making it very difficult to perform any analysis over this space—the phenomenon referred to as "the curse of dimensionality" [8].

2. **Structure of a DNN:** The structure of a DNN is very different from traditional programs and this deprecates all code-based testing techniques such as structural testing [9] and concolic testing [10], to name a few.

3. **Development Workflow:** The machine-learning development workflow [11] involves more iterative cycles and more rapid changes to the model under test compared to the traditional software development life cycle. The difference in workflow may necessitate reconsideration of how testing is performed throughout the development life cycle.

In response to the challenges of testing DNNs, the field of Machine Learning Testing [12] is being actively studied, especially on topics such as test reduction, test adequacy measurement, and test generation.

Test reduction aims to reduce the cost of test execution and analysis by reducing the number of test cases to run while retaining the effectiveness [13–16]. Test suite reduction, or test prioritization, is an important problem in day-to-day software engineering practices [17], but no known method or published literature existed when it comes to prioritizing test cases for DNNs.

Test adequacy measurement tries to solve the problem of quantifying the thoroughness of test execution by designing new test adequacy (coverage) criteria [18, 18–23]. All coverage criteria designed for DNN up to date are white-box coverage criteria, as the measurement is based on the DNN under test. For example, Neuron Coverage [21] measures the percentage of neurons in a DNN that are *activated* by running a test suite. Whether these coverage criteria are effective in finding failures of the model under test is often questioned [24, 25], but a better alternative is not known. More importantly, it is not clear whether a white-box coverage measuremet is even appropriate given the machine learning workflow [11] and the characteristics of DNNs. One of the obvious limitations is that the structure of a DNN can change completely through retraining, which invalidates any coverage analysis performed previously. This behavior is quite different from traditional software which mostly changes by increments. A coverage criterion effective at finding failures of the model under test that is not dependent on the model has not been studied.

Test generation aims at reducing the cost of test data collection by synthesizing realistic test cases that reveal new failures in the model under test [21, 23, 26–32]. Most of the test generation approaches rely on input-level transformations that preserve the label—called metamorphic transformations [30]—and are thus limited in their capability to the applicable algorithmic transformations [21, 23, 26–30, 33]. Also, many of these approaches introduce artificial input-level manipulations that may violate the assumptions of the environment in which the DNN operates [10, 21, 28, 29] diminishing the validity of the synthesized test inputs. Moreover, many approaches depend on the model under test for generating failure-revealing test cases; thus, the effectiveness is specific to the model under test. This dependence may be undesirable given the highly iterative ML workflow, especially when generating a regression test suite that can stay effective regardless of—or, at least, less sensitive to—the specific model under test. A DNN-independent test case generation approach that can synthesize realistic test inputs,

along with correct labels, without relying on metamorphic relations, is still desired.

In the aforementioned topics of testing, we observed that the dependence of the testing techniques to the DNN under test is a root cause of many problems. For the test adequacy criteria, all of the existing white-box criteria suffer from the same issues caused by model-dependence, such as high measurement cost due to the large size of DNNs and repeated measurements throughout the iterative ML development lifecycle. A majority of available test generation techniques also suffers from model-dependence since test case generated to reveal failure in one model might not be effective in another model after retraining with different model structure or different hyperparameters. A model-independent black-box approach, for both test adequacy measurement and test case generation, could solve these problems, but such an approach has not been studied actively in the literature.

1.1 Objectives and Contributions

The long range goal of this research is to develop systematic testing techniques for LECs composed of DNNs that address the unique challenges that accompany DNNs, such as (1) the high-dimensional and unstructured input data, (2) the distinct structure of the DNN, and (3) the iterative ML development workflow. For a testing technique to be scalable in the face of these challenges, challenges that seemingly only get worse with time, we set practicality and scalability as our top priorities, instead of looking for a solution that works only on small problems and then hoping to scale up. With this strategy in mind, we aim to achieve our goal through multiple techniques for various testing activites, including test reduction, test adequacy measurement, and test case generation. Concretely, this research aims to (1) develop techniques that reduce the cost of testing in large input spaces, (2) develop a test adequacy criterion that can measure coverage over the input domain rather than on the DNN, circumventing the need for dealing with the complexity of the DNN under test, and (3) develop a test case generation technique that can synthesize realistic and failure-revealing test cases without relying on the particular model under test.

On the path towards addressing the problems occuring from the white-box approaches, we introduce a complementary perspective called *manifold-based machine*

learning testing inspired by the ideas and principles of model-driven engineering and black-box testing. The central idea of manifold-based ML testing is to utilize a manifold—a low-dimensional subspace embedded within the input space wherein the high-dimensional real-world data lie—towards testing and assurance of LECs. By viewing the manifold as a domain model for the system under test, we harness the benefits of model-based engineering such as an ability to reason about the system in a higher level of abstraction, and cross-validation of software artifacts against each other.

The research discussed in this dissertation provides the following contributions:

Proposed and evaluated test prioritization techniques: As a step towards reducing the high cost of testing on high-dimensional input spaces, we proposed three methods that can measure the relative merit of a test input based on signals obtained from the DNN under test—(1) softmax cross entropy, (2) Bayesian uncertainty, and (3) input surprise. With these methods, one can prioritize test inputs that are more likely to reveal failures in the DNN under test, thus helping to determine smaller subset of test inputs to be labeled, or which test inputs are worth retaining in a regression test suite. In the experiment, we evaluated the effectiveness of test prioritization in terms of failure-revealing effectiveness of prioritized test suites. The results showed that all of the three methods are similarly effective at prioritizing failure-revealing test inputs; for example, 20% of the highest priority inputs contained 80% of the failure-revealing inputs in the original test suite.

Defined a black-box coverage criterion and evaluated its effectiveness: We defined a novel black-box test adequacy criterion called Manifold Combination Coverage (MCC) that base the coverage measurement on a manifold instead of the DNN itself. A *manifold* is obtained through an unsupervised manifold-learning technique called Variational Autoencoder, and we partition this manifold space into coverage requirements. By calculating the percentage of manifold space that a test suite covers, MCC can quantify the relative portion of the input space that a test suite exercised independently of the DNN. This independence eliminates the problems arising from the frequent changes in the DNN structure, which can happen very easily each time the model is retrained or when its structure is redefined. In the experiments, we empirically evaluated the

effectiveness of MCC in test suite construction in terms of (1) effectiveness of the constructed test suites in revealing failures of the model under test, (2) semantic balance of the constructed test suite, and (3) the effectiveness in retraining the model under test. The comparison with other white-box criteria showed that the effectiveness of MCC is comparable to other white-box criteria, and superior to the evaluated white-box criteria in terms of ensuring the semantic diversity of test data.

Developed a DNN-independent test case generation technique: As another activity of *manifold-based machine learning testing*, we developed an automated test case generation technique that utilizes a manifold for the creation of test cases. This approach solves the problem of generating realistic and novel test cases—both inputs and their expected outputs—that are effective at revealing failures in any model under test. This is achieved in a model-independent manner by utilizing a supervised classifier that can predict classification uncertainty for any encoding in the manifold space. The key part of generating test inputs is enabled by the generative decoder of the Variational Autoencoder that we used for manifold learning. The expected output (label) is also determined at generation-time by the manifold-based classifier. Additionally, we propose to apply search-based testing [34] on the manifold space so that one can search for failure-revealing test cases while staying within the data distribution captured by the learned manifold. The evaluation showed that this approach can generate quantitatively realistic inputs with correct labels. The comparison between the heuristic optimization against the naive random sampling showed that the on-manifold search can more effectively generate failure-revealing test cases, dropping the test accuracy to as low as 79% for a model that achieved higher than 99% validation accuracy.

1.2 Structure of this document

This dissertation is organized in seven chapters. Chapter 2 provides the preliminaries of machine learning and machine learning testing with surveys of related works. Chapter 3 describes test input prioritization techniques and evaluate their effectiveness with real-world image classification models. Chapter 4 introduces the overarching idea of this dissertation, about using manifold for testing activities. Chapter 5 defines a novel black-box coverage criterion called Manifold Combination Coverage and evaluates its

effectiveness. Chapter 6 provides manifold-based test case generation along with evaluation on image classification models. Finally, Chapter 7 summarizes the dissertation and provides concluding remarks.

Chapter 2

Background and Related Work

The background necessary for this research is presented in four sections. First, we introduce basic concepts of software verification and validation in Section 2.1 with a focus on software testing. Second, we present a brief preliminary on machine learning and deep learning in Section 2.2, including basic concepts and the workflow of machine learning engineering. Third, we discuss others' works related to the verification and validation of *learning-enabled systems* composed of neural networks in Section 2.3, to establish the context in which our proposed testing approaches fit in. Fourth, we discuss the properties that are desired of in learning-enabled systems in Section 2.3.3, and clarify the objectives of the proposed testing approaches.

2.1 Software Verification and Validation

Verification and validation (V&V) is one of the software engineering disciplines that help build quality into software [35]. V&V consists of a collection of activities across the software development life-cycle (SDLC). Borrowing the definition of Lake [36], "**verification** is the process of evaluating a system to determine whether the products of a given development phase satisfy the **specifications** imposed at the start of that phase." Verification answers the question of "was the system built correctly according to the specification?". On the other hand, **validation** is defined as "the process of evaluating a system to determine whether it satisfies the stakeholders of that system." Validation checks whether a developed system meets the expectation of the end user, and answers

the question of "was the right system built?" As can be noticed, these concepts are about the situation in which the *check* is performed, and the purpose thereof, and do not prescribe specific methods for performing the checks. Verification and validation are thus often referred together as V&V.

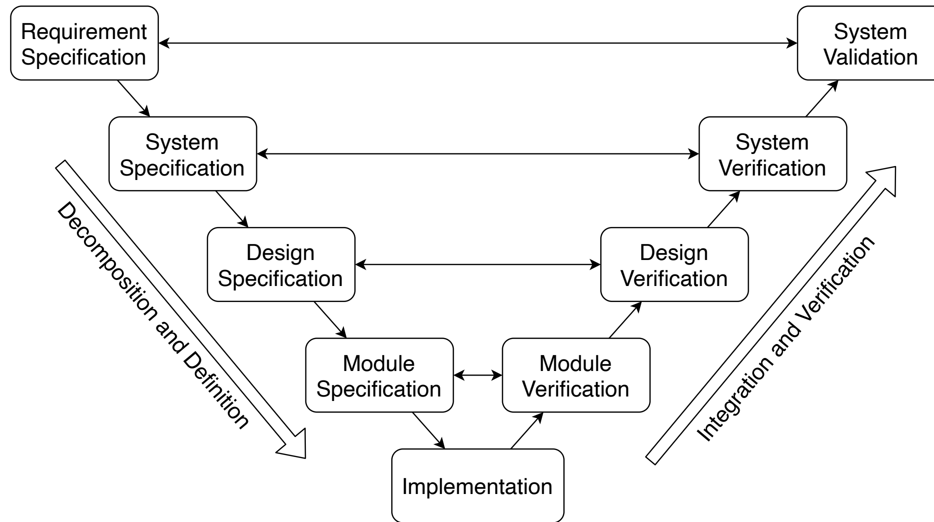


Figure 2.1: The V Model [37]

The role of V&V in a typical SDLC can be captured in the popular V model [37] illustrated in Figure 2.1. The activities depicted on the left side of the figure shows the decomposition and definition phase where the need of the system captured as **system requirements** is progressively refined and concretized into system implementation. The activities depicted on the right side of the figure shows the integration and verification phase, where the implemented modules are integrated from bottom up into a final system. Verification is performed at each upward step against the specification in the corresponding level, checking whether the software artifact is implemented correctly. Validation is usually performed at the last stage to check whether a right system was built with respect to the requirements of the end user.

The software V&V techniques can be roughly categorized into *static verification* that perform analysis of the system, and *dynamic verification* that performs experimentations with the system. Static methods analyze the system without executing it, using mathematical techniques. Mathematical proofs, when feasible, can provide a strong

guarantee on the correctness of the system, but it requires a deep expertise in the techniques and the system under verification to construct such a proof. Dynamic methods, on the other hand, execute the system and check its behavior. The barrier to applying dynamic methods is generally lower, as the implemented component or system can be executed as it is. However, as they require real executions, they are able to assess only a subset of the system's behavior, and cannot provide a proof in correctness. As such, static methods and dynamic methods complement each other. In practice, both static and dynamic methods are adopted, according to the criticality of the system under verification and the availability of resources. In the sequel, we briefly explain representative V&V techniques for each category: formal methods for static verification, and testing for dynamic verification.

2.1.1 Formal Verification

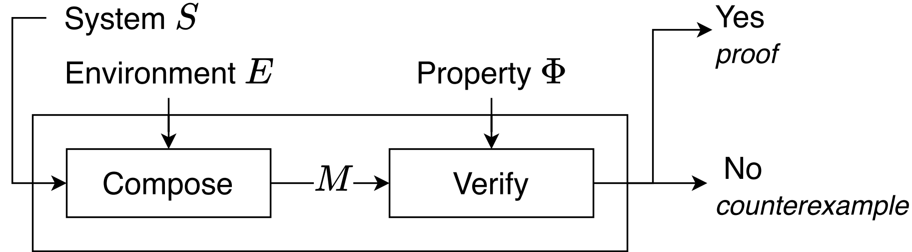


Figure 2.2: Formal verification procedure

Formal verification techniques algorithmically verify a model against formally specified properties. As illustrated in Figure 2.2, a typical formal verification procedure requires three inputs—(1) a model of the system to be verified, S, (2) a model of the environment E, and (3) the property to be verified Φ. The system S and the environment E are composed as M and proved against property Φ using computational proof engines such as Boolean satisfiability (SAT) solvers [38] and satisfiability modulo theories (SMT) solvers [39]. A representative example of a formal verification method is model checking [40]. Model checking is completely automated in that as long as S and E is described in a language that a model checker accepts, and the requirement is correctly formalized as Φ, model checker automatically checks the model for a violation of the

property Φ. When the property turns out to be true, it is guaranteed to hold for every possible state. When a violation is found, model checker returns a counterexample, or a sequence of inputs and actions that leads to a violation.

For the strengths of model checking, it can prove properties that express complex high-level requirements. For example, most of the model checkers accept temporal logic [40] as their property specification language; temporal logic can formalize requirements such as "When a passenger does not possess a passport or does not have a ticket, he/she shall not be on board the flight.", or "when a client submits a request, it should eventually be received by the server, and be processed immediately once it is received." Another strength is that once it proves that a property holds, it is guaranteed to hold for every possible state of the system captured by the model. This cannot be achieved by dynamic techniques such as testing except in trivial cases. Since a typical system has infinitely many number of states, and since testing can only execute the system for a finite number of times, testing cannot produce a proof, nor can it efficiently come up with a counterexample, no matter how much budget is spent on testing.

For the weaknesses of model checking, it generally requires an extra effort for constructing a model—an abstraction of the actual system with respect to the aspects of concern—as the verification algorithm cannot perform an efficient reasoning when all the implementation-level details are given. The creation of a formal model is an expensive process that requires a good understanding of the formal modeling language and also a deep understanding of the system's domain. Formalizing system requirements as property Φ is also a non-trivial task. Even when all the prior steps are done, there is a critical limitation in the scalability of the proof engine, as verification problems are undecidable. The model may turn out to be too complex for the model checker to handle, in which case the model checker runs indefinitely without producing any result. These limitations restrict the applicability of formal verification to domains such as hardware verification or verification of safety-critical embedded control logic, where the criticality of the systems justifies the high cost of verification and the problems can be reduced to tractable sizes.

To summarize, formal verification is a powerful method that is capable of providing a mathematical proof of the absence of faults for high-level requirements, but comes with a high modeling and verification cost. Formal verification is powerful for systems

that are small enough, but its applicability is limited by the size of the model it can handle.

2.1.2 Software Testing

Testing is defined as "an activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component" (IEEE-610) [41]. With the framework illustrated in Figure 2.2, testing dynamically executes S with some input $x \in X$ and checks whether $S(x)$ is acceptable with respect to the specification or requirements, which we will refer to collectively as **properties** P. In testing, this $x \in X$ is called a **test input**, X an input domain, and $S(x)$ the **system (program) output**. A test **passes** when $S(x)$ observes P, and **fails** when it does not. The mechanism for determining whether a test has passed is called **test oracle**, which can be expressed as a Boolean function O: $O(S(x), y) \rightarrow \{T, F\}$ for $y \in Y$. The output $y \in Y$ that evaluates $O(S(x), y)$ to T is called **expected output**. The tuple that consists of a test input and an expected output $(x, y) \cdot x \in X \wedge y \in Y$ is called a **test case**, and a set of test cases are called a **test suite**. A test case (x, y) that fails the test $O(S(x), y) = F$ is called a **failing test case**, or a **fault-finding test case**. To elaborate further on the basic concepts, a **failure** is observed when $\exists(x, y) \cdot O(S(x), y) = F$, which is preceded by an erroneous program state triggered by x propagating to one of the observable outputs of S. The program state becomes erroneous when a **fault**—or a **bug**—in the program is triggered and when it manages to *infect* the program state. A bug is introduced by **errors** or mistakes. The aim of testing is often to find as many failures as possible so that the faults can be revealed.

As testing performs the V&V of a system through dynamic executions, it can be applied much more broadly to any type of system for checking any type of property. It also has an advantage of being applicable at any stage of the software development life-cycle with minimal modification to the system under test. However, this flexibility and versatility comes with a fundamental limitation [42]. As testing requires executions of a system with a finite set of test cases, whereas the domain X is almost always infinitely large, it can only show the presence of faults, not their absence, unlike formal verification that can produce a sound proof for some types of properties. Instead, testing

techniques aim to find as many existing faults as possible within a given budget, so that the risk of using the software is reduced [43] and a higher confidence is gained on the quality of the software. Testing is often the only choice when other V&V methods are not applicable, or when the cost of applying more rigorous techniques is not justifiable.

Coverage Criteria

One of the most fundamental idea of testing is the use of **test adequacy criteria**, also called **coverage criteria**. The need for coverage criteria arises from the fundamental limitation of testing, that we cannot test all test inputs within the domain. A coverage criterion is typically used as a scoring function that measures the level of test adequacy, with 0% being not adequate and 100% being adequate. In other words, given a software system S and a finite set of test cases T, a coverage criterion c measures the **coverage score** of T with respect to S as $c : c(T, S) \rightarrow [0, 1] \in \mathbb{R}$. A coverage criterion does so by defining a finite set of **coverage requirements** over one or more of the software artifacts—such as the program under test or the specification—and by imposing these requirements on a test suite. The coverage score is the ratio of the number of coverage requirements satisfied by a test suite to the total number of coverage requirements. For example, *statement coverage* is a white-box coverage criterion that measures the ratio of program statements that are executed—and thus *covered*—by a test suite, to the total number of statements inside a program. A test suite that executes five statements out of a ten statements will achieve a 50% statement coverage. A test suite that executes all the ten statements will achieve 100% statement coverage, and called a statement-coverage-**adequate test suite**.

Coverage criteria can be used for several different purposes. The first use case is to evaluate the adequacy of the test suite, and repeatedly assess the adequacy during the construction of a test suite. In this use case, the coverage can pass a judgement on the adequacy of a test suite and guide the test suite construction indirectly by telling whether the coverage improves with additional test case or not. Test cases are created by a separate means—such as by deriving them from the system requirements—and the role of a coverage criterion is restricted to evaluating the adequacy of the test suite. In the second use case, the information about the coverage requirements can be used to directly guide the construction of a test suite. For example, an uncovered

statement can inform a tester on what additional test inputs can execute that statement. Furthermore, an automated technique can harness this information to create more test cases algorithmically, and doing so is called **coverage-guided test generation** [44, 45]. In the third use case, coverage criterion can be used to minimize a test suite. In this case, a test case is deemed useful only when it contributes to satisfying a coverage requirement. Test cases that do not contribute to increasing coverage are deemed redundant and discarded. A minimal set of test cases that most efficiently achieves an adequacy remains as a result. Coverage criteria are used most commonly in the three use cases described above, but its utility is not limited to them. The three scenarios are not mutually exclusive neither, as one can, for instance, combine coverage-directed test generation and coverage-guided minimization. In fact, a proper use of coverage criteria is still an active research topic (Discussed in Section 2.1.2).

White-Box vs. Black-Box Testing

There are several ways of classifying software testing activities [42]. One common taxonomy is by testing levels based on software activities, i.e., unit testing, system testing, etc. This categorization tells the context in which testing is performed in the SDLC (e.g., unit testing is performed during module verification as shown in Figure 2.1), but does not describe the characteristics of the testing technique applied. Another popular taxonomy is **white-box testing** vs. **black-box testing**, which classifies testing activities based on whether the structure of the implementation is utilized.

White-Box Testing White-box testing, also known as **structural testing**, refers to testing approaches that utilize the internal structure of the implementation and try to make sure that every structural element is exercised during testing [46]. This is achieved through white-box coverage criteria that defines coverage requirements over the structure of the program. White-box criteria provides an interpretable measure of test adequacy and provides a further guidance in test creation and selection. The rationale behind white-box coverage criteria is straight-forward: to find defects in the system, defects need to be triggered first as a precondition, which is asserted by coverage requirements. Coverage requirements can only serve as a precondition, because triggering a fault is often insufficient to incur an observable failure, which requires

additional steps—a triggered fault needs to corrupt the program state and the effect needs to propagates to observable variables. Achieving a structural coverage thus implies that a class of fault is triggered, which is a necessary, although not a sufficient, condition for the faults to be found. For instance, for a class of bugs that a simple execution is sufficient to trigger a fault, a test suite that achieves 100% statement coverage ensures that all of such faults are triggered. Other more powerful criteria such as branch coverage or Modified Condition and Decision Coverage (MC/DC) [47] defines more granular coverage requirements, and thus becomes more difficult for a test suite to satisfy. An MC/DC-coverage-adequate test suite is theoretically stronger than branch-coverage-adequate test suite, for instance, and is more capable of requiring test cases that catches subtle bugs.

For the advantages of white-box testing, the knowledge of the source code helps a thorough investigation of the implemented system. White-box criteria can provide principled guidance on constructing test cases and in determining when to stop testing [42]. On the opposite side, white-box testing adds complexity to testing since testers are required to understand the source code. There are often cases when some coverage requirements are not satisfiable, in which case the cause needs to be analyzed manually, further increasing the cost of white-box testing. But a fundamental limitation is that white-box testing only concerns with what is already implemented, and cannot discover missing functionality. For instance, one may implement only 50% of the requirements and achieve 100% structural coverage in testing. Structural coverage in this case does not tell us anything about whether all the requirements are implemented.

Black-Box Testing Black-box testing—also called **functional testing**—is an antithesis to white-box testing, referring to testing approaches that examine the functionality of the system without peering into its internal workings. Unlike white-box testing that focuses on testing the implementation itself, black-box testing focuses on testing the functionality that is expected to be implemented, which is described in software artifacts in higher abstraction level such as requirement, specification, input domain, or behavior model. In other words, black-box testing tests *what the system is supposed to do* rather than testing *how it does.* For example, functional testing derives test cases on the specifications of the system under test, with the goal of evaluating the compliance

of the software with specified functional requirements [48]. Model-based testing derives test cases from abstract models that represent the desired behavior of the system [49], to check the compliance of the system to the behavior model. Domain testing techniques such equivalence partitioning and boundary value analysis [50] derives test cases from the input domain model constructed using domain knowledge; the input domain is partitioned into semantically equivalent classes, and the values that lie around the boundary of the classes are identified and exercised.

Since black-box testing focuses on testing the functionality of the software, its main advantage is that it can discover faults in terms of required functionality, such as a missing feature. Also, black-box testing can be performed without an access to the source code. Test cases can be developed in the early development stages (Figure 2.1) before implementation. As these test cases can be executed without any knowledge of the system's internal workings, executing black-box test cases is relatively simpler and cheaper than white-box testing that requires additional apparatus for measuring and analyzing structural coverage. On the flip side, the disadvantages of black-box testing includes the reverse of the advantages of white-box testing. Black-box test cases may not exercise all the structural aspects of the implementation, potentially making the testing less thorough.

Summary As can be noticed, white-box testing and black-box testing are complementary approaches. White-box testing can more effectively find implementation-level issues, while black-box testing can more effectively identify issues in functionality. The two approaches are used together in practice, and different approaches are preferred depending on the level in which testing is performed. For instance, unit-level test cases are typically constructed with the help of white-box coverage criteria. System-level test cases are usually constructed from the requirements in order to test the compliance of the system. A hybrid gray-box approach is also frequently employed to harness the benefits of both approaches. For instance, test cases can be derived from the requirement for system-level testing, while the adequacy of testing is checked against the implementation using white-box criteria. This hybrid approach is also advocated for the certification of safety-critical software in avionics industry [51].

Effectiveness of Coverage Criteria

A coverage of 100% is rarely achievable because of various practical complications. For instance, there could be an error-handling logic that is unreachable unless a very unlikely event occurs. An inadequate test suite loses the guarantees described above, but intuitively, it is still appealing to think that a test suite that achieves 90% coverage is *better* than an alternative that only achieves 10%. Whether such a correlation exists cannot be proved theoretically, but it is hypothesized that the coverage score is positively correlated to a test suite's capability of finding faults. This hypothesis is called the Strong Coverage Hypothesis:

> **The Strong Coverage Hypothesis** (SCH): For the population of realistic software systems, test suites produced by human efforts or automated testing methods, and realistic faults, there is at least *a moderate statistical correlation between the level of coverage a suite achieves and its level of fault detection.* Moreover, this correlation is not the result of some trivial confounding factors that could be used in place of coverage, such as suite size. [52]

In other words, if SCH holds for a coverage criterion, one can interpret a higher coverage as a direct indication of a higher fault detection capability without regarding other variables. While one would desire that it holds, empirical evidences often suggest otherwise; for instance, test suite size is shown to affect the test suite effective quite significantly [53]. A weaker version of this hypothesis is given in The Weak Coverage Hypothesis:

> **The Weak Coverage Hypothesis** (WCH): For the population ... (omit; same sentence) ... of fault detection. This correlation is quite possibly the result of non-trivial, complex cause that produces both coverage and fault detection, but the existence of such causes is assumed by the use of different methods to produce test suites. Coverage still serves as a useful distinguishing measures for test suites, though we should not expect it to always be a significant predictor of fault detection for suites produced by the same underlying method. [52]

WCH states that various factors may influence the fault detection capability of a suite, but given the same method used for test creation, coverage positively correlated with fault detection capability of a test suite. WCH had been studied extensively for

popular white-box criterion such as branch coverage and MCDC coverage in the literature [52, 54, 55]. These studies sometimes reported conflicting results, and some other studies along a similar line also showed that effectiveness of white-box criteria are susceptible to the influence of the program structure and test generation method [56–58]. These results call for paying a careful attention when using coverage, regarding the variables that can influence the effectiveness of coverage criteria. Despite the controversy, white-box coverage criteria are still widely accepted and used both by practitioners and academic researchers.

2.2 Preliminaries of Machine Learning

Machine learning (ML) is the study of computer algorithms that improve automatically through experience. To be more precise, "a computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E. [59]" A learned program can thus perform tasks without being explicitly programmed unlike most of traditional software that require explicit programming of instructions on how to perform a task. Typical learning tasks T include classification, regression, prediction of structured output, anomaly detection, transcription, and machine translation [1]. The experience E is provided as a **dataset**, which is a collection of many examples. For supervised learning tasks, or the type of tasks that can encode E as a set of inputs and a set of corresponding outputs (called **labels**), the dataset can be represented as a set of tuples $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), ..., (\mathbf{x}_n, \mathbf{y}_n)\}$ with $\mathbf{x}_i \in X$ and $\mathbf{y}_i \in Y$. The performance measure P can be expressed per sample $(\mathbf{x}_i, \mathbf{y}_i)$ with function $L : Y \times Y \rightarrow \mathbb{R}$ called the **loss function**.

By the way that machine learning algorithms process the data during learning, machine learning approaches are roughly categorized into supervised, unsupervised, or reinforcement learning. **Supervised learning** algorithms experience dataset containing multiple input **features** together with the **label** associated with the features so that the unknown function $f : \mathcal{X} \rightarrow \mathcal{Y}$ can be learned. Supervised learning is again divided into **classification** when $\mathbf{y} \in \mathcal{Y}$ is categorical, and **regression** when $\mathbf{y}$ is continuous. For classification, the performance P per example is measured as a simple 0-1 loss function

defined as:

$$L(f(\mathbf{x}_i), \mathbf{y}_i) = I(f(\mathbf{x}_i) \neq \mathbf{y}_i) \tag{2.1}$$

where I is an indicator function and $\mathbf{y}$ is the ground-truth label for $\mathbf{x}$. The loss function for regression tasks is commonly defined as **squared error**:

$$L(f(\mathbf{x}_i), \mathbf{y}_i) = ||f(\mathbf{x}_i) - \mathbf{y}_i||^2 \tag{2.2}$$

Unsupervised learning algorithms, on the other hand, experience datasets containing multiple features, but without any labels, to learn useful properties of the structure of the dataset. One example of such task is clustering, which aims at grouping similar examples together. **Reinforcement learning** aims to learn a sequence of decisions for an *agent* in an interactive environment with a goal of maximizing the cumulative *reward*. A popular example is a StarCraft AI [60], a game-playing agent trained to win a match.

The theoretical framework of machine learning is provided by statistical learning theory, which concerns with finding a predictive function f based on data $\mathcal{D}$. We briefly explain the principle of **Empirical Risk Minimization** (ERM) to provide an intuition of how a learning problem is formulated. For supervised learning, the predictive function $f : \mathcal{X} \rightarrow \mathcal{Y}$—which is also referred to as a **model**—is considered to reside in a **hypothesis space** $\mathcal{H}$. The goal of learning algorithm is to search for $f \in \mathcal{H}$ with the best performance given an unknown data-generating distribution $P(\mathbf{x}, \mathbf{y})$. The mechanism for choosing the best hypothesis is to minimize the **expected risk** of choosing f, which is defined as:

$$R(f) = \mathbb{E}_{P(\mathbf{X},\mathbf{Y})}[L(f(\mathbf{x}), \mathbf{y})] = \int L(f(\mathbf{x}), \mathbf{y}) dP(\mathbf{x}, \mathbf{y}) \tag{2.3}$$

The notion of risk is associated with f, and calculated by integrating per-sample loss $L(f(\mathbf{x}), \mathbf{y})$ over $P(\mathbf{x}, \mathbf{y})$ with $\mathbf{y} \in \mathcal{Y}$ being the label associated with $\mathbf{x}$. The goal of learning algorithm is to find a hypothesis $f \in \mathcal{H}$ for which the risk $R(f)$ is minimal:

$$f^* = \arg\min_{f \in \mathcal{H}} R(f) \tag{2.4}$$

For a parameterized model $f_\theta \in \mathcal{H}$, the objective becomes $f_{\theta^*} = \arg\min_\theta R(\theta)$ with $R(\theta) = \mathbb{E}_{P(\mathbf{X},\mathbf{Y})}[L(f_\theta(\mathbf{x}), \mathbf{y})]$. This optimization, however, is intractable since the

ground-truth distribution $P(\mathbf{x}, \mathbf{y})$ is unknown. Instead, a proxy measure R_{emp} called **empirical risk** is introduced with a finite dataset $(\mathbf{X}, \mathbf{Y}) \sim P(\mathbf{x}, \mathbf{y})$:

$$f_{\theta^*} = \arg\min_{\theta} R_{emp}(\theta) \cdot R_{emp}(\theta) = \frac{1}{n} \sum_{i}^{n} L(f(\mathbf{x}_i), \mathbf{y}_i) \tag{2.5}$$

where $(\mathbf{x}_i, \mathbf{y}_i) \in (\mathbf{X}, \mathbf{Y})$ for $1 \leq i \leq n$. Informally, the principle of ERM states that an optimal parameter set θ for function f can be found by minimizing the empirical risk calculated as the average loss over the finite dataset $(\mathbf{X}, \mathbf{Y})$ sampled from the ground-truth distribution $P(\mathbf{x}, \mathbf{y})$. This procedure of estimating the unknown model parameters by solving the optimization problem above is called **training**, and the dataset $(\mathbf{X}, \mathbf{Y})$ is called a training dataset. The objective function for optimization is called **cost function**. These elements together—a model, a cost function, and an optimization procedure—constitute a machine learning algorithm. The settings to the learning algorithm that control this algorithm's behavior, but are not adjusted by the algorithm itself, are called **hyperparameters**.

If we apply ERM to a regression task with the squared error loss function in Equation 2.2 for L, the expected error for $(\mathbf{x}, \mathbf{y}) \sim P(\mathbf{x}, \mathbf{y})$ can be decomposed into three terms:

$$\mathbb{E}[(\mathbf{y} - f_{\theta^*}(\mathbf{x}))^2] = (Bias[f_{\theta^*}(\mathbf{x})])^2 + Var[f_{\theta^*}(\mathbf{x})] + \sigma^2 \tag{2.6}$$

where $Bias[f_{\theta^*}(\mathbf{x})] = \mathbb{E}[f_{\theta^*}(\mathbf{x})] - f(\mathbf{x})$ and $Var[f_{\theta^*}(\mathbf{x})] = \mathbb{E}[(\mathbb{E}[f_{\theta^*}(\mathbf{x})] - f_{\theta^*}(\mathbf{x}))^2]$, and σ^2 an irreducible error [61]. The bias term is the expectation of error between the truth and the prediction, and the variance term is the expectation of the variance of this error. In other words, Equation 2.6 decomposes mean squared error into three different kinds of error—bias error, variance error, and residual error. **Bias error** is introduced by erroneous assumptions about the learning algorithm. For example, a linear model that regresses a dataset sampled from a quadratic function inevitably introduces a high bias error. In this case, the model is said to **underfit**. **Variance error** is introduced by modeling the random noise in the dataset, and thus by **overfitting** to the training data. This model becomes too sensitive to small fluctuations in the input. For example, a quadratic model that approximates a linear function can easily overfit to random noise in the data. As these two errors are optimized simultaneously, they are in conflict with each other, creating a tradeoff between bias and variance.

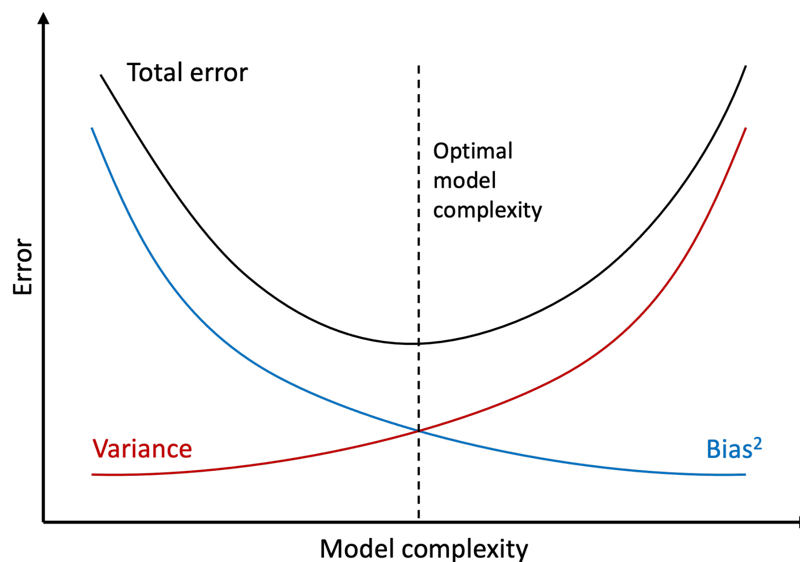


Figure 2.3: Bias-Variance Trade-off

If we conceptually project the models of varying complexities onto a linear scale, the relation between the model complexity and the error can be depicted as in Figure 2.3. If a model is too simple for the complexity of the dataset, the model will show a high test error, mostly consisting of bias error. If a model is too complex for the dataset, on the other hand, the model will also show a high test error, consisting mostly of variance error. Conceptually, an optimal model complexity can be found in the middle ground where the total error is minimized. However, this task is beyond the scope of optimization defined in Equation 2.5, and requires multiple rounds of experimentations with different assumptions about the model.

2.2.1 Deep Learning

Deep learning is a branch of machine learning that uses deep neural network (DNN) as a model. Like any machine learning algorithm, deep learning algorithms consist of (1) a model, (2) a cost function, and (3) a learning procedure. The elements that predominantly distinguish deep learning from other machine learning algorithms are the model and the learning procedure.

For the model, deep learning uses various kinds of deep neural network architectures

that are often specific to the type of task they perform. The simplest architecture of neural network is called **feed-forward neural network**, or multi-layer perceptron, and it only flows the information forward from input x to the output y with no feedback connection in-between. It can be represented mathematically as a composition of a finite number of heterogeneous functions. For example, we can have three functions $h^{(1)}$, $h^{(2)}$, and $h^{(3)}$ connected in a chain to form $f(x) = h^{(3)}(h^{(2)}(h^{(1)}(x)))$. The $h^{(i)}$ here is called the i-th **layer** of the network, and the number of layers in a network gives the **depth** of the network. In its most basic form, a layer is defined recursively as:

$$\mathbf{h}^{(i)} = g^{(i)}(\mathbf{W}^{(i)T}\mathbf{h}^{(i-1)} + \mathbf{b}^{(i)}) \tag{2.7}$$

where $\mathbf{h}^{(i-1)}$ is the output of the $(i-1)$-th layer, $\mathbf{W}$ is called the **weight** matrix, $\mathbf{b}$ is called the **bias**, and g is an element-wise non-linear function called **activation function**. In other words, a layer applies a linear transformation on the output of the previous layer and then apply an activation function to introduce non-linearity. The dimensionality of $\mathbf{W}^{(i)}$ and $\mathbf{b}^{(i)}$ determines the **width** of the i-th layer. These weights and biases in f_θ constitute the set of model parameters θ, which is also referred to as the set of **trainable parameters**.

For the learning procedure, deep learning mostly uses a variant of Stochastic Gradient Descent (SGD) with back-propagation for optimizing the parameters of the model θ [62]. At a high-level, gradient-descent refers to a family of iterative algorithms that updates a set of model parameters θ towards minimizing the loss, using the information about the sensitivity of the change in loss with respect to a change in the model parameter θ. A train dataset is randomly divided—thus stochastic—into subsets of equal sizes called **mini-batches**. In each **step**, the algorithm computes the direction and magnitude of change in θ that most quickly reduce the loss value with respect to a given mini-batch. This procedure is written in the following equation:

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta J \tag{2.8}$$

where $\nabla_\theta J$ refers to the gradient of the loss J with respect to θ and α refers to the **learning rate** that determines the rate at which the parameters are updated. Since $\nabla_\theta J$ represents the *steepest slope* of θ that can reduce the loss value by the largest amount, this iterative algorithm *descends* the gradient of the loss landscape by taking small

steps towards the steepest slope—thus called gradient descent. Each round of iterative updates that consumes the whole train dataset once is called an **epoch**. Gradient descent repeats for a finite number of epochs until the loss converges, and this number of epochs is determined empirically.

2.2.2 Machine Learning Workflow

Before discussing the verification and validation of learning-enabled components, we first establish the context in which verification is performed. In traditional models of software development life cycle such as waterfall model [63], software development consists of sequential activities—requirement elicitation, design, implementation, verification, and maintenance. An extended model like V-model shows that the former phase of development concerns with a refinement of abstract idea into concrete implementation, and that the latter phase concerns with verification and validation of implementation with respect to the former artifacts such as specifications and requirements. Although engineering an ML system roughly follows a similar sequence of stages, it is distinct in that the whole process is very much dependent on data, and that every stage is highly iterative. Unfortunately, a general consensus in the understanding of the ML system development life cycle does not seem to have been established yet, due to the rapid and disruptive adoption of machine learning into software development projects in the recent years. Here, we introduce one of the ML workflow models recently suggested by Amershi *et al.* [11].

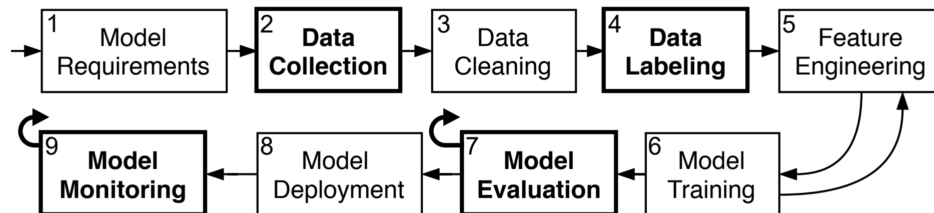


Figure 2.4: The nine stages of machine learning workflow [11]. A *loop* sign indicates that the process can loop back to any of the previous stages.

Figure 2.4 shows the workflow of ML model development consisting of nine stages. In *model requirement* stage, the requirement of the model is identified, and the type of

the model is decided for the given problem. Data is collected in *data collection* stage, and inaccurate and noisy data is corrected or removed in *data cleaning* stage; then ground-truth label is assigned in *data labeling* stage in the case of supervised learning. In *feature engineering* stage, the collected data is further processed in a way that makes the problem easier to solve. It can involve identifying relevant features, or finding an appropriate representation of the features. For some models such as convolutional neural network, this stage is less explicit—as it implicitly happens during training—and blended with the next stage. In *model training* stage, the chosen model is trained and tuned with the cleaned, labeled, and engineered data. *Model evaluation* stage tests the trained model using pre-defined metrics to verify the model against the requirements. If the model does not qualify, the development jumps back to any of the previous stages and repeats the process again until the cause of the problem is identified and fixed. The model is then deployed to the target environment with auxiliary code and infrastructure necessary to process the incoming data and make predictions. The deployed model is continuously monitored for a possible degradation in performance.

From a data-centric perspective, the development of an ML model can be described in three sequential stages supported by three disjoint datasets—(1) model creation with a **training set**, (2) model tuning with a **validation set**[1] , and (3) model evaluation with a **test set**. The training set is used to tune the parameters of the model, thus contributing directly to the creation of the model. The validation set is used to perform an independent evaluation of the model's performance during training, and estimate the true performance of the model. The loss computed with the validation set—called validation loss—is often compared against the loss computed with the training set—called train loss—to determine whether the model is overfit to the training data. While the validation set is not used directly to tune the parameters of the model, it is used as a basis for selecting among multiple candidates, and thus contributes indirectly to the model creation. Consequently, there exists a risk that the model overfits to the validation set—i.e. the model that achieves the highest performance measured by the validation set does not achieve the best performance with the real data. To resolve this issue, yet another independent dataset called test set is introduced. Test set provides an

[1] Note that the term *validation* here has a different meaning to the term validation used in software engineering.

unbiased estimate of the model's final performance independent of the model's creation and selection.

In practice, these three datasets are commonly sampled from the same distribution, i.e., by first sampling a dataset from the population and then by splitting it into three separate train–validation–test datasets in $8:1:1$ ratio. In this practice, notice that the size of the test set is much smaller than that of the train set, and no additional qualifier—such as the adequacy of this test set—is imposed on it. Although this small test set might provide a somewhat objective assessment of the model, we do not know whether it is sufficient to answer all the multi-faceted questions about the quality of the model. For instance, does the model generalize to input perturbations that occur commonly in the operational environment? Is the test set really sampled from the ground-truth data distribution? Otherwise, the performance measured with test set will not accurately estimate the actual performance after deployment.

In traditional software development, especially when it comes to safety-critical systems, a set of rigorous verification activities is performed to ensure that the specifications and the requirements are satisfied. Testing, for instance, is performed in an extensive manner so that as much of the implementation is exercised and checked for correctness against specifications and requirements. With the typical "testing" performed with a test set in ML context, this process is lacking, and does not provide any means to answer the questions raised above, such as "is the test set adequate?" We believe that answering these questions require a more thorough investigation on the trained model, which can be supported through rigorous verification and validation activities that push the model beyond "testing with the test set" in the ML context.

2.3 Verification and Validation of Learning-Enabled Systems

The growing capability of deep learning is facilitating a wide adoption of learning-enabled technologies in real-world applications and promises unprecedented gain in productivity. Deep learning sits at the core of many disruptive innovations in computer vision applications [64, 65] such as medical image analysis [66, 67] and autonomous driving [68, 69]. We call these systems as **Learning-Enabled System**s (LES), as one or

more of its core functionality is implemented by machine learning. In other words, a Learning-Enabled System refers to a system in which one or more of its components are *learned*—called **Learning-Enabled Component** (LEC)—instead of being explicitly programmed [70]. These systems adopted in domain such as medical image analysis or autonomous driving are safety-critical, meaning that a failure in such systems can cause harm in human lives and properties [71]. Safety-critical systems have to be verified and validated rigorously, yet the V&V techniques for LESs are catching up slowly [], hindering a wide-adoption of learning-enabled technologies in domains where a significant benefit is expected.

This begs the question, what is particular about the V&V of LESs? Although the V&V process for LESs is not fundamentally different from that of any software systems, V&V of LESs inherit all the problems of verifying non-ML software systems plus an additional set of ML-specific challenges [72]. One of the distinct challenges is dealing with randomness, which is an inherent property of LESs. Randomness is introduced at multiple levels—a finite amount of training data is sampled randomly from the population, the weights of the model are often initialized randomly [73], and the algorithm for updating the parameters is also stochastic [74]. With so many layers of randomness, it becomes very difficult to analytically deduce the cause of a failure even when one is observed during the V&V. Another unique challenge is the difficulty of (formal) specification in the component-level. Most notorious examples can be found in computer vision and natural language processing, which are the domains where deep learning showed the most remarkable achievements. For these applications where the input data is high dimensional and unstructured, it is impossible to accurately specify the requirements of a system in terms of input and output relations, either formally or informally. The requirements for these systems are often specified only in high-level in terms of probability measures—i.e., 99% classification accuracy—and cannot be further decomposed into lower-level specifications due to the monolithic nature of LECs. Furthermore, these probabilistic requirements generate a run-time V&V challenge for the LES as a whole. When an LEC can only guarantee its performance as a probabilistic measure, the system needs to be engineered in a way that can prevent a system-level failure even in the place of stochastic failure in the LEC. Although various prevention mechanisms have been proposed [75, 76], the problem of system-level verification of LESs

still remains to be addressed.

In an attempt to deal with the above-mentioned challenges, much effort is being expended on researching V&V techniques and developing assurance methods [77]. The V&V approaches can be roughly grouped into formal verification [6] and testing [12]. A brief overview of the two approaches and related works will be provided in the sequel.

2.3.1 Formal Verification of Neural Networks

The DNN verification algorithms typically accept a (trained) DNN model as the system S. Since there is no known method for composing S, which is implemented as a DNN, and E, which is typically represented as a state machine or mathematical formula, the environment is usually modeled as unconstrained [72]. The property Φ is restricted to constraints over the input and the output of the DNN model. For a model f that takes $\mathbf{x}$ as input and produces $\mathbf{y}$ as output, as in $\mathbf{y} = f(\mathbf{x})$, formal verification verifies whether the following assertion holds:

$$\mathbf{x} \in \mathcal{X} \implies \mathbf{y} = f(\mathbf{x}) \in \mathcal{Y} \tag{2.9}$$

where $\mathcal{X} \subset \mathcal{D}_x$ expresses the input constraint $\mathcal{Y} \subset \mathcal{D}_y$ expresses the output constraint, and $\mathcal{D}_x$ and $\mathcal{D}_y$ are the domain and the range of f. The proof is *sound*, meaning that a property is reported to hold only when it actually holds. For some algorithms, the proof is also *complete*, meaning that whenever a property holds, the algorithm will always report that it holds. Algorithms that are both sound and complete are prohibitively expensive and do not scale beyond neural networks with the size of a few thousand parameters [78, 79]. Further innovation in formal verification can possibly improve its scalability in the future, but given that the verification algorithms are NP, it may not be reasonable to expect formal techniques to scale to industry-scale neural networks in a near future. And even when it scales, the expressiveness of formal specification is currently limited to constraints over the range of input and output values. For instance, the functionality of a DNN that processes high-dimensional data—such as image classification model—cannot be formally specified.

The pros and cons of formal verification techniques are not unique to the neural network domain. For the V&V of traditional software in industry, formal verification techniques are applied only in a small segment of the industry where the high cost of

formal verification can be justifiable, and even there within, applied to a small critical subset of the system under verification to which the algorithms scale. Although a standard V&V practice for LESs have not yet been established, it is likely that a similar pattern is repeated for the V&V of LESs—formal methods are applied only to a small subset of small critical components. And for the majority of LESs, the verification will likely be performed by testing.

2.3.2 Machine Learning Testing

Machine learning testing (ML testing, in short) is an area of software testing research concerned with testing the software that is built using machine learning. One of the most widely accepted definition of machine learning testing is provided in a recent survey by Zhang *et al.* [12] as follows: "ML Testing refers to any activites designed to reveal machine learning bugs (ML bugs)", where an ML bug is defined as "any imperfections in a machine learning item that causes a discordance between the existing and the required conditions." However, we find this definition to be too general and informal—what is a *machine learning item*? What exactly do *existing and required conditions* or *imperfections* refer to? We provide an alternative definition as follows:

Definition 1 (ML Testing) *ML Testing refers an activity in which a Learning-Enabled System (LES) or a Learning-Enabled Component (LEC) is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.*

In this definition, the *specified conditions* refer to the constraints of the operational environment and the input domain. The *results* include the output of the system or component, along with non-functional aspects of the system or component such as the run-time performance. The *evaluation* is made with respect to the system or component-level specification. Note that the definition is the same as that of testing except that the subject is LES or LEC, and does not provide any insight into what makes ML Testing particular. The survey by Zhang *et al.* [12] summarized the difference between traditional testing and ML Testing in their survey, but we provide a revised summary that better fits our perspective, in Table 2.1. The characteristics are grouped into four categories of concern—target, artifacts, and objectives.

Characteristics	Traditional Testing	ML Testing
Implementation	Hand-written/binary code	Code and neural networks
Structure to test	Control-flow structure	Unknown (data-flow?)
Adequacy criteria	Structural coverage or mutation score	Unknown
Interpretability	Comprehensible	Difficult or impossible
Malleability	Relatively rigid	Easy to change (by retraining)
Test input	Structured	Unstructured and high-dimensional
Specification	Logical	Statistical
Test oracle	Logical statements	Defined per instance
Cause of fault	Human mistakes or wrong specification	Data, training algorithm, model, etc.
Fault attribution	Relatively easy	Very difficult
Fault repair	Bugs get fixed manually	Unknown (retraining?)

Table 2.1: Comparison between traditional software testing and ML testing

First, for the test target, traditional software is mostly implemented manually in bottom-up fashion, except for rare cases where code is automatically synthesized. There is an explicit structure to test, either in source code or binary code, and this is harnessed by structural testing techniques [42]. The code is comprehensible, as it is mostly written by human, which helps assigning meaning to exercising certain structural element that is covered or not covered by a test suite. The code evolves incrementally through development life-cycle, and the velocity of change in the codebase is limited by the human productivity. An LEC, on the other hand, is implemented with both code and DNN. The sturcture of a DNN is distinct from code. Most notably, it lacks explicit control flow structure, which invalidates all the testing philosophy and structural testing techniques that have been developed around the presence of the source code. These techniques include structural adequacy criteria [42], code-level mutation testing [80], symbolic execution [81], concolic testing [10], and bounded model checking [82], to name a few. DNN is very malleable, as the construction of a DNN other than the architecture design is done automatically by learning algorithms. For instnace, a completely new DNN can be trained in a matter of hours with a completely different architecture and a different set of training data. However, it is notoriously difficult to comprehend the internals of a trained DNN. This malleability and un-interpretability existed in small

doses in traditional software, which used to make testing difficult, but its intensified presence in LEC makes testing much more challenging.

Second, for the testing artifacts, inputs to traditional software is usually well-defined and well structured. In the case when large or high-dimensional data such as texts or videos are given as input, the operations that a program perform are limited to simple manipulation of data, and the semantics of the data do not play a major role in the functionality of the program. The specification of traditional software can mostly be written as logical statements, and often be formalized in propositional, predicate, or temporal logic. When the specification can be formalized, it can be used to create an automated test oracle that can check the conformance of system behavior for infinitely many test cases. The inputs for an LEC, on the other hand, are often unstructured and high-dimensional, such as text or images, and the behavior of an LEC is solely dependent on these inputs. The specification cannot be captured formally in many cases, and instead stated as statistical statement, such as a certain level of accuracy. An automated oracle cannot be synthesized; had an automated oracle existed, the LEC need not be developed in the first place. Instead, oracle is provided per instance through manually labled pairs of an input and an expected output.To reinforce our points, let us consider a hypothetical specification for a cat vs. dog image classifier, written as: *"Given an image that contains a clear picture of either a cat or a dog, the classifier shall assign a correct label of a cat or a dog with 95% accuracy."* The relation between the inputs and the outputs cannot be captured with logic. Checking this type of probabilistic statement would require testing to be performed over a *well-distributed* sample of test cases. And the test oracle can only be provided as labeled instances.

Third, for the objective of testing, that is to find and fix faults, the characteristics of faults are vastly different. In traditional software, fault is mostly caused either by a wrong specification or an incorrect implementation of the specification. Once a failure is detected, its root cause can be localized through manual analysis and fixed mostly by applying a local patch. On the other hand, a failure of LEC can be caused by many factors, or even a combination of them, including, but not limited to training algorith, model architecture, hyper-parameters, and training data. Idntifying the root cause of a failure is more difficult accordingly, if feasible. Repairing a fault is not straight-forward neither, and one needs to rely heavily on the intuitive understanding

of ML artifacts—such as optimization algorithm, the interplay of hyper-parameters to the model's performance and deep insight about the training data—in order to fix the root cause of a fault. Even if one can successfully localize and fix a bug, it is practically impossible to expect an ideal performance—such as 100% classification accuracy—in any of the realistic tasks, since, in the end, ML models can only approximate an ideal function with a finite amount of training data.

Given these differences, the ML Testing is under active research [12], trying to catch up with the revolutionary speed at which the ML techniques evolve. The topic of ML Testing research can be categorized in many ways, but according to Zhang *et al.*'s classification by testing workflow, the area of research can be roughly categorized into following topics: (1) test input generation, (2) test oracle identification, (3) test adequacy evaluation, (4) test prioritization and reduction, (5) bug report analysis, and (6) debug and repair. In the sequel, we discuss related work of the topics that are relevant to the proposed research—test input generation, test adequacy measurement, and the effectiveness of test adequacy criteria.

Test Input Generation

Test input generation concerns with automatically generating test inputs for LES under test. Doing so can save the cost of testing when the cost involved with test input collection exceeds that of automated test generation. Test generation for traditional software could be automated when the input is well structured and the constraints over the input space is clearly defined. Test generation paradigms range from random generation such as in coverage-directed random testing [83] or fuzzing [84], intelligent search as in search-based test generation [34], to techniques that automatically synthesize test inputs using SAT or SMT based automated reasoning tools [39] such as concolic testing [10]. However, when the input is high-dimensional such as images, and when the set of valid inputs resides only in a much smaller subspace of the input space, realistic test inputs cannot be synthesized from a scratch.

Some test generation techniques for testing DNNs work around this problem by synthesizing new inputs based on existing seed inputs. Note that most of the test generation research up to date are focused on generating still images for image classification or regression tasks. DeepXplore [21] demonstarted that test inputs can be

synthesized through a joint optimization that maximizes a disagreement among an emsemble of models under test. However, the resulting test inputs look unarguably synthetic. DeepTest [26] performed greedy search with realistic image transformations with the goal of generating feailure-revealing test inputs. The label of seed images is preserved under the applied image transformations, such as blurring and rotation. The resulting inputs are more realistic, but the transformations are limited to the extent of image filters. DeepRoad [27] introduced the idea of using generative adversarial networks (GAN) for applying more complex image transformations in order to generate realistic images that extend beyond simple filters. They used style-transfer GAN that can learn a style from a set of training dataset and transfer the style to other dataset, and applied it to Udacity Challenge dataset, an image dataset designed for developing self-driving cars. They learned the *style* of snowy driving scenes from one dataset and applied this generative model to the training dataset of the LEC under test. This technique can effectively expand the test dataset to environments where test data collection is difficult, with the upfront cost of training the style GAN once per each style.

Test Adequacy Measurement for Neural Networks

The idea of measuring test adequacy based on the structure of a neural network was first introduced by DeepXplore [21]. They interpreted neurons inside a neural network as an analog of program statements, and defined Neuron Coverage which measures the percentage of neurons activated by a set of test cases. Since then, a plethora of structural coverage criteria had been proposed for testing neural network [19, 20, 22, 23, 28, 85–89]. These criteria base their measurement on the structural element of the DNN under test, analogous to structural coverage—such as statement coverage or branch coverage—defined for testing traditional programs. We briefly introduce some of them.

The first family of coverage criteria are neuron-level coverage. A nueron n is *covered* if there exists a test input in a test suite which triggers n to produce an intermediary output that is greater than a pre-configured threshold value, which is often set to zero. **Neuron Coverage (NC)** [21] requires a test suite to *activate* all the neurons inside a DNN, where this activation is determined by a user-specified threshold value. **k-Multisection Neuron Coverage (KMNC)** [19] is a more granular extension of

neuron coverage which partition the range of neuron's output into k sections, and require a test suite to cover each of them. **Neuron Boundary Coverage (NBC)** [19] requires a test suite to exercise the neurons beyond the range of values known from the train data, either below or above. It is thus designed to exercise corner-case behaviors of a DNN.

The second family of coverage criteria look beyond individual neurons, and investigate the relationships among sets of neurons. Below, we first explain some concepts that are necessary to describe this family of criteria.

- **Neuron pair**: A neuron pair $(n_{k,i}, n_{k+1,j})$ are two neurons in adjacent layers k and $k+1$ such that $1 \leq k \leq K-1$, $1 \leq i \leq s_k$, and $1 \leq j \leq s_{k+1}$.

- **Sign change**: Given a neuron $n_{k,l}$ and two test cases x_1 and x_2, we say that the sign change of $n_{k,l}$ is exploited by x_1 and x_2, denoted as $sc(n_{k,l}, x_1, x_2)$, if $sign(v_{k,l}[x_1]) \neq sign(v_{k,l}[x_2])$, where $v_{k,l}$ denotes the activation value of $n_{k,l}$.

- **Value change**: Given a neuron $n_{k,l}$ and two test cases x_1, and x_2, we say that the value change of $n_{k,l}$ is exploited with respect to a value function g by x_1 and x_2, denoted as $vc(g, n_{k,l}, x_1, x_2)$, if $g(u_{k,l}[x_1], u_{k,l}[x_2])$ = `true` and $\neg sc(n_{k,l}, x_1, x_2)$. $u_{k,l}[x_1]$ denotes the activation value of $n_{k,l}$ with input x_1.

- **Distance change**: Given the set of neurons $P_k = \{n_{k,l} | 1 \leq l \leq s_k\}$ in layer k and two test cases x_1 and x_2, we say that the distance change of P_k is exploited with respect to a distance function h by x_1 and x_2, denoted as $dc(h, k, x_1, x_2)$, if $h(u_k[x_1], u_k[x_2])$ = `true` and $\forall n_{k,l} \in P_k$, $sign(v_{k,l}[x_1]) = sign(v_{k,l}[x_2])$. The distance function $h(u_k[x_1], u_k[x_2])$ can be instantiated as norm-based distances, or structural similiarity distances.

Based on these definitions, Sun *et al.* [28] defined the following four coverage criteria.

- **(Sign-Sign Cover, or SS Cover)**: A neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is SS-covered by two test cases x_1, x_2, denoted as $cov_{SS}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$: $sc(n_{k,i}, x_1, x_2) \wedge \forall p_{k,l} \in P_k \backslash \{i\}. \neg sc(n_{k,l}, x_1, x_2) \wedge sc(n_{k+1,j}, x_1, x_2)$. In other words, $\alpha = (n_{k,i}, n_{k+1,j})$ is SS-covered when x_1 and x_2 can show that $n_{k,i}$ can independently affect the sign change of n_{k+1} while the signs of other neurons in layer k is held constant.

- **(Distance-Sign Cover, or DS Cover)**: Given a distance function h, a neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is DS-covered by x_1 and x_2, denoted as $cov^h_{DS}(\alpha, x_1, x_2)$, if the following conditions are satisfied by $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$: $dc(h, k, x_1, x_2) \wedge sc(n_{k+1}, x_1, x_2)$. In other words, DS Cover is achieved for a neuron $n_{k+1,j}$ when x_1 and x_2 demonstrates a significant change of distance in the output vector of the neurons in the k-th layer, and when the sign of $n_{k+1,j}$ flips as a result.

- **(Sign-Value Cover, or SV Cover)**: Given a value function g, a neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is SV-covered by two test cases x_1, x_2, denoted as $cov^g_{SV}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$: $sc(n_{k,i}, x_1, x_2) \wedge \forall p_{k,l} \in P_k \backslash \{i\}. \neg sc(n_{k,l}, x_1, x_2) \wedge vc(g, n_{k+1}, x_1, x_2)$.

- **(Distance-Value Cover, or DV Cover)**: Given a distance function h and a value function g, a neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is DV-covered by two test cases x_1, x_2, denoted as $cov^{h,g}_{DV}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$: $dc(h, k, x_1, x_2) \wedge vc(g, n_{k+1,j}, x_1, x_2)$.

These criteria are inspired by the design of Modified Condition and Decision Coverage (MC/DC) [47], a structural coverage criterion for hand-written programs that requires a test suite to demonstrate the indiependent effect of individual *conditions* inside every *decision* in conditional statements. To explain with an example, for a decision d that consists of three conditions a and b, such as $d = a \wedge \neg b$, an MC/DC-adequate test suite needs to demonstrate a case when a independently affects d while $\neg b$ is held constant, and another case when a is held constant and $\neg b$ independently affects d. The four coverage introduced by Sun *et al.*follow similar design to MC/DC, in a sense that for a neuron $n_{k+1,j}$, the outputs of neurons in the previous layer n_k were considered analogous to *conditions* that constitute a *decision*, and $n_{k+1,j}$ as a *decision* that is affected by the *conditions*.

The third family of criteria are neural network semantics coverage. These criteria require extra analysis that derives semantic information from the neural network under test, such as *relative surprise* or *importance*, and use them as a basis of coverage analysis. **Surprise adequacy (SA)** [20] takes the activation traces from the DNN and interprets them as representations of input data. The relative surprise of an input is measured by its relative proximity to inter-class and intra-class data, where the distance is measured

between the activation traces. The coverage is defined by computing the range of surprise values and dividing it into n sections, requiring a test suite to present a diverse range of surprise values. Importance Driven Coverage (IDC) [22] follows a similar design principle of capturing the relative importance of a test input, and it requires a test suite to exercise a full range of importance values.

The introduced coverage criteria are analogous to white-box structural coverage criteria in traditional testing, in that the internal structure of the DNN (as a program) under test is utilized to determine the adequacy of testing. For the hand-written programs, the rationale for white-box testing is that the structural element of a program—such as statement, branch, basic block, function, etc.—is constructed by human, directly encoding an intent of human programmers, which are essentially derived manually from the specification of the program. These elements are logical, and a human-understandable meaning can be assigned to them. The white-box criteria for DNNs, however, cannot be rationalized in the same manner. As neural networks are trained in a top-down fashion—the weights of the individual neurons are adjusted automatically by learning algorithms, governed by the cost function—where as hand-written programs are constructed bottom-up, the same justification cannot be found in the level of individual neuron or a group of neurons. A neuron, or a set of neuron, clearly embed a fraction of functionality that the network as a whole implements, but whether counting the individual coverage over them has any significance is unclear.

Effectiveness of DNN Coverage Criteria

Before discussing the effectiveness of coverage criteria, it needs to be clarified for what they try to achieve. We believe that the very intent of designing a coverage criterion is to measure the adequacy of testing, or to quantify how *well* a program is exercised by a set of test cases. And within this context, we can assess the effectiveness of a criterion by studying the correlation between the coverage score that a test suites achieves and the fault-finding effectiveness of that test suite; or in other words, by empirically studying the Strong Coverage Hypothesis. However, coverage criteria are employed in other use-cases, whether appropriately or inappropriately. For instance, Ashmore and Banks [90] suggested four different use cases of DNN coverage criteria: 1) to construct an optimal training dataset, 2) to compare test data agatinst training data, 3) for coverage-directed

test case generation, and 4) to assess the relative merit between different DNNs. While we do not agree that every use case is *appropriate*, the research community has not reached consensus yet, and we leave it upon the subjective judgement of the reader. We first introduce some works that investigated the effectiveness of coverage criteria in contexts other than as test adequacy criteria.

Dong *et al.* [91] studied the correlation between coverage score and the robustness of the DNN. They measured coverage score of a trained model with the validation datasets, and interpreted the score as if it is a measure of the model's adversarial robustness. We consider it a misuse of a coverage criterion, as coverage indicates the quality of a test suite, not the quality or any other property of the DNN under test. Another study conducted by Yan *et al.* [92] follows a similar line of thought, attempting to measure correlation between the coverage score and the DNN model's adversarial robustness. They used coverage to direct adversarial test case generation, applied adversarial training to *fix* the *bug*, and measured the correlation between the coverage score and the adversarial robustness that the retrained model achieved. We do not agree with this experiment design, or the very assumption behind this experiment design: that "higher test coverage suggests better software quality given the same subject software" [92]. In our view, based on our interpretation of coverage as an adequacy criterion, higher test coverage suggets a better test thoroughness, but nothing about the program under test. Yet, aside from this issue, they draw some interesting observations—1) adversarial examples generated with the guidance of coverage look more unnatural than gradient-based adversarial examples, and 2) effective adversarial examples may not lead to higher coverage.

Another thrust of research studied the correlation between coverage score and the fault-finding effectiveness of a test suite—which is the setting that we believe as the most appropriate usage of coverage criteria. The first-of-its-kind work in this category titled "structural coverage criteria for neural networks could be misleading" [24] concluded that the fault-detection capabilities conjectured from high coverage are more likely due to the adversary-oriented search (the test generation algorithm itself) rather than the effect of coverage used in the coverage-guided adversarial test generation. With a similar theme, Fabrice *et al.* [25] investigated the effectiveness of Neuron Coverage in generating adversarial examples, when used for coverage-guided test input generation. Within the

context of Neuron-Coverage-guided adversarial test case generation, they evaluated the ratio of adversarial examples, the naturalness of produced test inputs, and the output (label) distribution of the found adversarial examples. Contrary to expectation, they found that higher neuron coverage leads to fewer adversarial examples detected, less natural input (aligning with the observation made by Yan *et al.* [92]), with more biased label distribution. Abrecht also studied the effectiveness of Neuron Coverage in test generation and reached a similar conclusion [93].

In summary, DNN coverage criteria are still in the process of going through experimental scrutiny. The conclusions from recent investigations indicate that the coverage criteria are not very effective in the context of finding or generating adversarial examples. But an important piece of information is still missing. Finding adversarial examples is not the only goal of testing, and much less the most important goal. Given the role of testing in the ML workflow 2.4, the place wherein coverage criteria can play the most important role is during model evaluation, for the *functionality* of the model under test. With software testing analogy, testing for adversarial robustness is comparable to security testing under the presence of adversary, while testing the model for its performance, e.g., generalization, is comparable to testing the functional correctness of the system under test. Just as the functional correctness is a prerequesite for robustness, we argue that testing for functionality shall be the primary goal of ML testing, and the effectiveness of coverage criteria shall also be evaluated accordingly, with the main focus on their effectiveness in finding natural yet failure-revealing test cases.

Before embarking on this investigation, we elaborate what we mean by the *functionality* of a DNN.

2.3.3 Desired Properties

In program verification, one of the most popular formal systems that is used for reasoning about the correctness of a program is Hoare Logic [94]. The central feature of Hoare Logic is the Hoare Triple $\{P\}C\{Q\}$, where P is an assertion called the *preconditon*, Q is is an assertion called the *postcondition*, and C is called a *command*. The Triple asserts that when the precondition is met, executing the command establishes the postcondition. If we borrow this concept to testing of DNNs, P constrains the input $\mathbf{x}$, C corresponds to a DNN under test f, and Q constrains the output $\mathbf{y} = f(\mathbf{x})$. However,

describing P and Q for each input is inefficient, if ever possible, and we desire to find a generic expressions of P and Q for a *set* of $\mathbf{x}$s and $\mathbf{y}$s. We call a generic assertion over the precondition and postcondition a **property**. While some properties may be task-dependent, there are some generic properties that need to hold for a majority of DNNs under test. With an example in traditional software testing, *no null-pointer dereference* is one such generic property that need to hold for any program. In this section, we discuss some important generic properties that we desire to hold for most DNNs under test.

Category of Input Data

Generic properties of DNN assert that "for this category of inputs *(precondition)*, this postcondition shall hold". Describing the precondition, however, can be challenging for some DNNs, such as image classification models, as it cannot be captured precisely—for instance, how can we formulate the precondition of a valid hand-written digit? The precondition cannot be checked by the DNN under test neither, as an ML model can process any input data as long as the structure of the data matches, regardless of the semantic validity of the input. But still, we wish to draw boundaries between the *valid inputs*—the domain of inputs that the DNN is designed and trained to operate on—and the *invalid inputs*. And further, even within the category of *valid inputs*, there can be multiple subsets, which we wish to distinguish, albeit less precisely, so that we can define generic properties for testing DNNs.

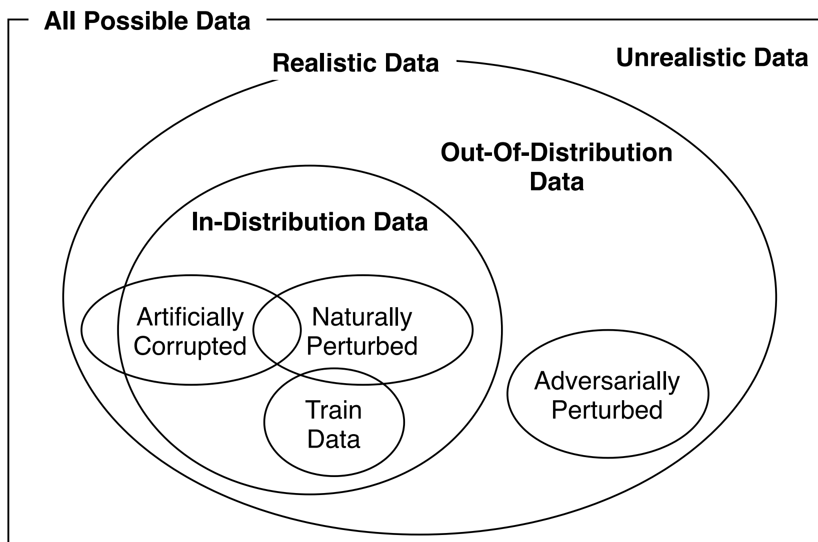


Figure 2.5: Category of Input Data

We address this concern with the conceptual map of the different categories of data, presented in Figure 2.5. The ground-truth distribution $P(\mathbf{x}, \mathbf{y})$ corresponds to the *in-distribution data*, and the *training data* represents $(\mathbf{X}, \mathbf{Y}) \sim P(\mathbf{x}, \mathbf{y})$. *Realistic data* is a category that contains both *in-distribution data*—those that have non-zero probability of being sampled from $P(\mathbf{x}, \mathbf{y})$—and the *out-of-distribution data* that are literally outside the ground-truth data generating distribution—for example, a plausible yet confusing hand-written digit that looks both like nine and seven, or even a hand-written alphabet. *Unrealistic data* are those that are assumed impossible by design, with respect to an operating environment. The scope of data that DNN verification has to deal with is typically the *realistic data* category, or in-distribution data, depending on the purpose of testing. Testing with the in-distribution data can be seen as functional testing in normal operating condition, whereas testing with the out-of-distribution data can be seen as robustness testing. The three unexplained categories within—artifically corrupted data, naturally perturbed data, and adversarially perturbed data—are the focus of the three generic properties that are described below.

Adversarial Robustness

Robustness is a property of an ML system that indicates the "degree to which it can function correctly in the presence of invalid inputs or stressful environmental conditions" [95]. [2] . Adversarial robustness is a sub-category of invalid inputs in which the difference between the malignant input and the benign one is imperceptible by human [78]. For a neural network f, *δ-local adversarial robustness* is defined as:

$$\forall x' \cdot ||x - x'|| \leq \delta \implies f(x) = f(x') \tag{2.10}$$

meaning that small adversarial perturbation of size δ or smaller shall not change the output of the model. Since the local robustness is specified per input, a more general *(δ, ϵ)-global-adversarial-robustness* is defined in [78] as:

$$\forall x_1, x_2 \in D \cdot ||x_1 - x_2|| \leq \delta \implies ||f(x_1) - f(x_2)|| < \epsilon \tag{2.11}$$

where D is the input domain and ϵ is the magnitude of tolerable output perturbation. In other words, a (δ, ϵ)-robust model f shall produce consistent output within output error tolerance ϵ for any input perturbation smaller than δ. Testing for adversarial robustness often means revealing the absence of adversarial robustness by creating or finding adversarial examples.

Corruption Robustness

Corruption robustness quantifies the robustness under natural corruptions that may occur in a normal operating environment, such as motion blur, change in brightness, and noise in sensory devices. It is contrary to adversarial robustness, which concerns imperceptible perturbations that are intentionally crafted to attack a DNN. Unless the objective of testing is to find security vulnerabilities, corruption robustness more closely captures the real-world robustness requirement of the model under test. When f is the model under test, $\mathcal{D}_{gt}$ is the ground-truth distribution of the data, and $\mathcal{C}$ be the distribution of corruptions in the real world, corruption robustness [96] is defined as:

$$\mathbb{E}_{c \sim \mathcal{C}}[\mathbb{E}_{x \sim \mathcal{D}_{gt}} \mathbb{I}[f(c(x)) = f(x)]] \tag{2.12}$$

[2] In Figure 2.5, this *invalid inputs* correspond to the *out-of-distribution data* category

where x is an input and $\mathbb{I}$ is an indicator function. In other words, Equation 2.12 is the expected performance of the model f measured by metric $\mathbb{I}$ under an expected set of corruptions $\mathcal{C}$. The goal of testing for corruption robustness is to find a $c \in C$ and an $x \in D$ such that $f(c(x)) \neq f(x)$.

Generalization

Although corruption robustness expresses a more general requirement than adversarial robustness, the operational domain of DNNs is not limited to enumerable corruptions. In fact, we wish our model f to *generalize* to any data drawn from a hypothetical ground-truth population $\mathcal{D}_{gt}$, instead of simply memorizing the training data. The generalization of the model is quantified as:

$$\mathbb{E}_{(x,y)\sim\mathcal{D}_{gt}}\mathbb{I}[f(x) = y] \tag{2.13}$$

Equation 2.13 quantifies the expected value of the correctness of the model f, determined by the indicator function $\mathbb{I}$, when the data (x, y) is sampled from the ground-truth distribution $\mathcal{D}_{gt}$. However, since $\mathcal{D}_{gt}$ is unknown, the performance of a trained model is instead measured on a test suite D_{test} as follows [97]:

$$\frac{1}{|D_{test}|} \sum_{(x,y)\in D_{test}} \mathbb{I}[f(x) = y] \tag{2.14}$$

This equation is equivalent to Equation 2.13 except that a concrete test dataset D_{test} is used as a proxy of $\mathcal{D}_{gt}$. If f generalizes, the performance of the model measured on a sufficiently large test dataset D_{test} drawn from $\mathcal{D}_{gt}$ shall match the true performance measured on $\mathcal{D}_{gt}$. The goal of testing for generalization can be twofold—1) to find $(x, y) \in \mathcal{D}_{gt} \cdot f(x) \neq y$, and 2) to find a test suite $\mathcal{D}_{test} \approx \mathcal{D}_{gt}$ that reveals discrepancy between the performance measured by Equation 2.13 and 2.14. The first goal is achieved by finding what is called natural adversarial examples [98, 99], or test cases that are found in the real world that trigger failures. The second goal of finding a *good* test suite, that finds many failures while having a distribution similar to $\mathcal{D}_{gt}$, is more challenging.

What Are We Testing For?

Since the discovery of adversarial examples [100, 101], adversarial attacks and defenses have been hot research topics [102, 103]. However, adversarial robustness that is measured with deliberately created attacks does not necessarily translate to better performance of the models on real-world data distribution [99]. Recently, people are paying more attention to generalization, albeit with different names, such as common corruption robustness [96], generalization [97, 104], and natural adversarial examples [99]. Some of the recent works acknowledge the importance of generalization to realistic real-world data, and either attempt to generate new test cases [27, 105] or measure the efficacy of coverage criteria for realistic test inputs [22]. We believe that testing research should pay more attention to testing for the generalization of the model rather than contending with the discovery of adversarial or unrealistic test inputs, as evidence suggests that adversarial examples are pervasive and no practical defense method exists yet [24, 106].

2.4 Summary

In this chapter, we reviewed the background and related work necessary for the proposed research. We reviewed the preliminaries of software V&V and the preliminaries of machine learning and deep learning, with a focus on the software engineering aspects. In the intersection of these two fields of knowledge, we introduced background on the V&V of learning-enabled system, and reviewed related work in Machine Learning Testing.

From our perspective on the current state of the art, the most critical gap that needs to be addressed, which is also a great research opportunity, is testing DNNs for generalization. While the majority of the ML Testing research has focused on developing techniques that can address adversarial robustness, we believe that generalization is the most important property of a DNN model that needs to be addressed first. We believe that the threads of ML Testing research, such as test case generation, test adequacy measurement, and the evaluation of test adequacy measures for their effectiveness, should focus around the generalization, in addition to adversarial robustness. The research we propose address the ML Testing problems with this observation in mind.

Chapter 3

Test Input Prioritization

One of the main reasons testing of DNNs is expensive is the large input space that testing has to cover. The cost associated with collecting test inputs can be high, but in many cases, the cost of labeling is much higher. This is because for many of the tasks that DNNs are designed to handle, the input data is abundant and easy to collect, while the oracles—the mechanism for assigning a correct output—cannot be fully automated. If otherwise, there is no need for the DNN in the first place. Thus, it is prudent to find ways to minimize the labeling effort for new test inputs.

One way to achieve this is to prioritize those inputs that are likely to reveal the weakness of a trained model so that the labeling effort can be focused only on prioritized inputs. We hypothesize that this priority can be determined by deriving some additional information about the computation performed by the DNN—its *sentiment*—when processing the inputs. Higher priority inputs are those for which the DNN under test expresses a stronger relevant sentiment. In particular, we study three sentiments—*confidence*, which is defined as the predicted probability associated with the output label in DNNs that use *softmax* output layers, *surprise*, which is defined as the distance of the neuron activation pattern on an input from the activation patterns on the training data, and *uncertainty*, which is defined for Bayesian Neural Networks based on the the probability distribution of the DNN's prediction. These measures are useful for prioritizing inputs that help us to more efficiently: (a) assess model weakness with reduced labeling cost, (b) assess model accuracy with a reduced test suite, and (c) retrain more effectively with fewer prioritized data.

We empirically assess how these measures perform as indicators of the test input's value using examples of DNNs for image classification and regression. The evaluation shows that the sentiment measures can effectively prioritize inputs that lead to erroneous DNN outputs.

In short, the key idea of test input prioritization is to capture the behavioral signature of a DNN model (which we call *sentiment*) unique to each input, in order to judge the relative value of that input compared to others for exercising the network. We hypothesize that the model's sentiment can serve as a signal to identify inputs that are more likely to reveal errors in the model. Although such sentiment measures are not readily provided by a typical DNN unless explicitly modeled, there exist multiple techniques that can capture model sentiment by inspecting the internal computation of the neural network, as will be introduced in the following subsection.

3.1 Sentiment Measures

We propose three sentiment measures—(a) predicted confidence, (b) Bayesian uncertainty, and (3) input surprise.

3.1.1 Predicted Confidence

Softmax is a logistic function that squashes a K-dimensional vector $\mathbf{z}$ of real values to a K-dimensional vector $\sigma(\mathbf{z})$ of real values where each entry of $\sigma(\mathbf{z})$ is in the range $[0, 1]$ and the entries add up to 1: $\sigma(\mathbf{z})_j = e^{z_j} / \sum_{k=1}^{K} e^{z_k},\ j = 1, ..., K$. It is typically used as the last layer of a DNN for a multi-class classification task so that the output can represent the categorical probability distribution of the K classes. When available, the priority score of test cases can be computed directly from the softmax output while incurring a minimal computational overhead. As an instantiation of the scoring function, we borrow the notion of entropy to summarize the distribution and assign a single score to an unseen test input $\mathbf{x}_i$:

Definition 2 (Predicted Confidence) *Given a classification DNN f that outputs a probability vector $\mathbf{y} =< y_1, y_2, ..., y_C >$ as $\mathbf{y} = f(\mathbf{x})$ where $\mathbf{y} \in [\mathbf{0}, \mathbf{1}]^{\mathbf{C}}$ and $\sum_{c=1}^{C} y_c = 1$,*

the confidence of f for an input $\mathbf{x}$ is defined as

$$-\sum_{c=1}^{C} f(\mathbf{x})_c \log f(\mathbf{x})_c \tag{3.1}$$

.

The score is lower when the DNN is *certain* about its output with only one y_c being high, and the score is higher when the DNN is *uncertain* about its classification, with the predicted probability distribution being spread out. Thus, a higher score, or a higher priority, is assigned to inputs with which the DNN under test is more uncertatin about its classification.

A limitation of softmax-based prioritization is that it can be applied only to classification DNNs that use a softmax output layer. However, a more fundamental limitation is that the predicted probability are known to be unreliable, as demonstrated by Gal and Ghahramani [107] and also shown in the case of adversarial input attacks [101, 108, 109]. For instance, an adversarially perturbed input that looks just like an ostrich to human eyes can be classified as a panda with 99% confidence. These limitations call for other prioritization measures that can be more reliable and also apply to regression models, which will be provided in sequel.

3.1.2 Bayesian Uncertainty

Model uncertainty is the degree to which a model is uncertain about its prediction for a given input. An uncertain prediction can be due to a lack of training data—known as epistemic uncertainty—or due to the inherent randomness in the data that cannot be reduced even with more information—known as aleatoric uncertainty [110]. But we do not distinguish the two for our purpose of prioritizing failure-revealing test cases. As it is practically impossible for a machine-learning model to achieve 100% accuracy, knowing model uncertainty is immensely useful for engineering a more robust learning-enabled component. In order to obtain a model's uncertainty along with the prediction, we need mathematically grounded techniques based on Bayesian probability theory. We briefly introduce Bayesian Neural Network and a technique to approximate it using existing neural networks. The uncertainty estimated by these techniques can then be used as scores to prioritize test inputs.

Bayesian Neural Network A typical (non-Bayesian) neural network has deterministic parameters that are optimized to have fixed values. A Bayesian neural network (BNN) [111], on the other hand, treats parameters as random variables that can encode distributions. For training, Bayesian inference [112] is used to update the posterior over the weights $\mathbf{W}$ given the data $\mathbf{X}$ and $\mathbf{Y}$: $p(\mathbf{W}|\mathbf{X}, \mathbf{Y}) = p(\mathbf{Y}|\mathbf{X}, \mathbf{W}) \times p(\mathbf{W})/p(\mathbf{Y}|\mathbf{X})$, which captures the set of plausible model parameters given the data. To make the training of the weights tractable, the weights are often fitted to a simple distribution such as the Gaussian, and the parameters (mean and variance in the case of the Gaussian distribution) of the distributions are optimized during training [113]. The likelihood of the prediction is often defined as a Gaussian with mean given by the model output: $p(\mathbf{y}|f^{\mathbf{W}}(\mathbf{x})) = \mathcal{N}(f^{\mathbf{W}}(\mathbf{x}))$ where $f^{\mathbf{W}}(\mathbf{x})$ denotes random output of the BNN [110] for an input $\mathbf{x}$ and $\mathcal{N}$ a normal distribution.

Uncertainty in Bayesian Neural Networks For a classification DNN f that outputs a softmax probability distribution $\mathbf{y} = f(\mathbf{x}) =< y_1, y_2, ..., y_C >$, the likelihood of predicting an output c for an input $\mathbf{x}$ is defined as:

$$p(y = c|\mathbf{x}, \mathbf{X}, \mathbf{Y}) \approx \frac{1}{T} \sum_{t=1}^{T} f^{\mathbf{W}}(\mathbf{x})_c \tag{3.2}$$

with T samples. For regression, the uncertainty is captured by the predictive variance which is approximated as:

$$Var(\mathbf{y}) \approx \frac{1}{T} \sum_{t=1}^{T} f^{\mathbf{W}}(\mathbf{x})^T f^{\mathbf{W}}(\mathbf{x}) - E(\mathbf{y})^T E(\mathbf{y}) \tag{3.3}$$

with T samples and the predicted mean $E(\mathbf{y}) \approx \frac{1}{T} \sum_{t=1}^{T} f^{\mathbf{W}}(\mathbf{x})$ [110]. In other words, the predictive variance is obtained by passing the input $\mathbf{x}$ T times to the model $f^{\mathbf{W}}$ and by computing the variance among T sampled outputs.

Monte-Carlo Dropout as a Bayesian Approximation Dropout is a simple regularization technique that prevents neural network models from over-fitting to training data [114]. It works during the training phase by randomly dropping out some neurons in specified layers with a given probability, so that the model parameters are changed only for the sampled neurons. Since the model parameters are adjusted only by an

infinitesimal amount in each iteration, the cost converges after sufficient training iterations, even with the variance introduced by random selection of neurons. At test time, the dropout is disabled so that every neuron participates in making a deterministic prediction. This simple technique is shown to be very effective in improving the performance of neural networks on supervised learning tasks. It was later discovered by Gal and Ghahramani [107] that a dropout network can approximate a Gaussian Process [115]. They proved that an arbitrary neural network with dropout applied before every weight layer is mathematically equivalent to an approximation of a probabilistic Gaussian process. They also showed that any deep neural network that uses dropout layers can be changed to produce uncertainty estimations by simply turning on the dropout at test time (unlike the typical use case where dropout is turned off), and the likelihood can be approximated with Monte Carlo integration. The uncertainty of the model can then be estimated in a same way as in Equation 3.2 and 3.3; the only difference being that the weight $\mathbf{W}$ varies by sample t and follows the dropout distribution such that $\hat{\mathbf{W}}_{\mathbf{t}} \sim q_\theta^*(\mathbf{W})$ where $q_\theta(\mathbf{W})$ is the dropout distribution. We refer more curious readers to the works by Gal and Ghahraman [107] and Kendall and Gal [110].

Bayesian Uncertainty for Classification and Regression DNNs By interpreting a DNN with Dropout as a Bayesian Neural Network, and by interpreting a run-time Monte-Carlo Dropout as a Bayesian approximation, we define Bayesian Uncertainty as a sentiment measure as follows:

Definition 3 (Bayesian Uncertainty (Classification)) *Given a classification DNN* $\mathbf{y} = f^{\mathbf{W}}(\mathbf{x}) =< y_1, y_2, ..., y_c >$ *trained with one or more Dropout layers, with the Monte-Carlo Dropout turned on at run-time, the Bayesian uncertainty is defined as*

$$-\sum_{c=1}^{C} E(\mathbf{y}) \log E(\mathbf{y}) \tag{3.4}$$

where $E(\mathbf{y}) \approx \frac{1}{T}\sum_{t=1}^{T} f^{\mathbf{W}}(\mathbf{x})$.

Definition 4 (Bayesian Uncertainty (Regression)) *Given a regression DNN trained with one or more Dropout layers, with the Monte-Carlo Dropout turned on at run-time,*

the Bayesian uncertainty is defined as

$$\frac{1}{T}\sum_{t=1}^{T} f^{\mathbf{W}}(\mathbf{x})^T f^{\mathbf{W}}(\mathbf{x}) - E(\mathbf{y})^T E(\mathbf{y}) \tag{3.5}$$

where $E(\mathbf{y}) \approx \frac{1}{T}\sum_{t=1}^{T} f^{\mathbf{W}}(\mathbf{x})$.

3.1.3 Input Surprise

Surprise Adequacy (SA, in short) is a test adequacy criterion defined by Kim *et al.* [20] to assess the adequacy of a test suite for testing deep learning systems. Informally, SA is achieved when a set of test inputs demonstrates varying degrees of model *surprise*, measured by a likelihood or a distance function, relative to the training data. The rationale is that a good test suite shall demonstrate a diverse and representative behavior of a trained model, and that the *surprise* can be a good representation of such diversity in behavior.

Unlike other coverage criteria introduced for testing neural network so far, such as Neuron Coverage [21], MC/DC-inspired criteria [28], or other structural criteria [19], SA is more fine-grained and unique in that it can assess the quality of each input individually. For example, SA can measure the relative surprise of an input to the training data and give it a numeric score—the higher the score is, the more surprising it is to the model. Our take on SA is that a surprising input may be more likely to reveal an erroneous behavior in the trained model since a high surprise may indicate that the model is not *well prepared* for the input, and, thus, should be given a high score.

Kim *et al.* [20] defined two ways of measuring the surprise, both make use of the activation trace of a DNN during classification. For the classification of every input, each neuron in the DNN f produces an output value (see Chapter 2.2). The vector of neuron outputs produced by a DNN can then be termed as the *activation trace* of that input. Given a set of neurons N in f, and a set of activation traces $A_N(T)$ for a known set of inputs, or training data T, the surprise of a new input $\mathbf{x}$ with respect to T can be computed by comparing the activation trace $\alpha_N(\mathbf{x})$ of with $A_f(T)$. Kim *et al.* [20] proposed two ways of making such comparisons.

1. A probability density function can be computed for the set of activation traces for known inputs. For a new input, we can compute the sum of differences between

its estimated density and densities of known inputs. The higher this sum is, the more surprising the new input is. This method is termed *Likelihood-based Surprise Adequacy (LSA)*.

2. Another method for comparing activation traces is to use a distance function to create another surprise adequacy criterion called *Distance-based Surprise Adequacy (DSA)*. Given the set of known inputs and a new input $\mathbf{x}$, DSA computation first finds the input $\mathbf{x}_a$ that is the closest neighbor of $\mathbf{x}$ with the same predicted class as $\mathbf{x}$. Next, it finds the input, $\mathbf{x}_b$, that is closest to $\mathbf{x}_a$, but has a predicted class different from the one predicted for $\mathbf{x}$. Next $dist_a$ and $dist_b$ is computed as:

$$dist_a = ||\alpha_N(\mathbf{x}) - \alpha_N(\mathbf{x}_a)|| \tag{3.6}$$

$$dist_b = ||\alpha_N(\mathbf{x}_a) - \alpha_N(\mathbf{x}_b)|| \tag{3.7}$$

with $\mathbf{x}_a$ and $\mathbf{x}_b$ defined as:

$$\mathbf{x}_a = \underset{f(\mathbf{x}_i)=c_\mathbf{x}}{\operatorname{argmin}} ||\alpha_N(\mathbf{x}) - \alpha_N(\mathbf{x}_i)|| \tag{3.8}$$

$$\mathbf{x}_b = \underset{f(\mathbf{x}_i)\in C\setminus\{c_\mathbf{x}\}}{\operatorname{argmin}} ||\alpha_N(\mathbf{x}_a) - \alpha_N(\mathbf{x}_i)|| \tag{3.9}$$

for any $\mathbf{x}_i \in T$. Finally, the DSA score for a new input x can be computed as:

$$DSA(x) = \frac{dist_a}{dist_b} \tag{3.10}$$

LSA is computationally more expensive and requires more parameter tuning than DSA. One parameter is the set of layers that need to be chosen for LSA computation. Activation traces for LSA will then only consist of activation values of neurons in these selected layers. Another parameter is the value for variance used to filter out neurons whose activation values were below a certain threshold. DSA, while still being sensitive to layer selection, benefits more than LSA from choosing deeper layers in the network and has fewer parameters that need to be tuned. For these reasons, we propose to use DSA as a measure of input surpirse, which is defined as folows:

Definition 5 (Input Surprise) *Given a classification DNN f and a set of neurons N in f, Input Surprise of an unseen input $\mathbf{x}$ is defined as*

$$\frac{||\alpha_N(\mathbf{x}) - \alpha_N(\mathbf{x}_a)||}{||\alpha_N(\mathbf{x}_a) - \alpha_N(\mathbf{x}_b)||} \tag{3.11}$$

where $\mathbf{x}_a = \text{argmin}_{f(\mathbf{x}_i)=c_\mathbf{x}} ||\alpha_N(\mathbf{x}) - \alpha_N(\mathbf{x}_i)||$ *and* $\mathbf{x}_b = \text{argmin}_{f(\mathbf{x}_i)\in C\setminus\{c_\mathbf{x}\}} ||\alpha_N(\mathbf{x}_a) - \alpha_N(\mathbf{x}_i)||$.

3.2 Experiment

We empirically assessed the efficacy of the input prioritization techniques in two use-case scenarios. The first use-case is finding failure-revealing data. The prioritized failure-revealing data can be used as additional training data, or as test data. The second use-case is retraining the model with the selected fraction of the prioritized data as in active learning, which is a natural next step for utilizing prioritized data. With these scenarios in mind, we propose the following research questions.

- **RQ1.** Can we effectively prioritize test inputs that reveal erroneous behavior in the model?
- **RQ2.** Can the prioritized inputs be used to retrain the model effectively?

The efficacy for RQ1 can be measured as the cumulative percentage of error revealing inputs after prioritization. The efficacy for RQ2 can be measured by comparing the accuracy of the model retrained with prioritized inputs against the baseline model retrained with randomly-selected inputs. For both RQ1 and RQ2 we use the same population of unseen test inputs.

We answered these research questions for each prioritization measure and compare their relative performance. As concrete instantiation of the prioritization techniques, we compared the following configurations: 1) softmax, 2) dropout Bayesian with 10 Monte-Carlo samples, 3) dropout with 100 samples, 4) Distance-based Surprise Adequacy (DSA) measured over the last one layer, and 5) DSA measured over the last two layers. For the techniques that require multiple Monte-Carlo sampling, we compared between 10 and 100 to assess the trade-off between sample size and prioritization efficacy. For DSA, we measured the distance of activation traces taken from the last one layer or last two layers. This choice is in accodance with the settings from the original paper that introduced DSA [20], with the rationale being that the layers closer to the output layer encodes the semantics of the input in higher-level abstractions, serving as a better representation of the input data. Although the efficacy of DSA can be higher when the

activation traces were taken from the middle layers of the neural networks according to Kim *et al.* [20], we limit our choice of layers due to the high space and time complexity of the DSA algorithm, which is quadratic to the length of the activation trace. The hidden layers deeper than the last two layers were typically too wide, producing a long activation trace, too long to be handled efficiently given our hardware constraints.

3.2.1 Systems Under Test

To simulate a realistic testing scenario where a trained model is scrutinized with additional test data, we chose two representative systems for image classification and image regression. The first system is a digit classification system trained with the 60,000 MNIST [116] training dataset. We test the system with the EMNIST [117] dataset, an extension of MNIST which is compatible to its predecessor. The second system is called TaxiNet, which is designed for an aircraft in ground operation to predict the distance to a center line and the heading angle deviation from a center line while taxiing. It is designed and developed by our industry partner Boeing as a research prototype to assess the applicability of learning-enabled components in the safety-critical domain. The data collection and training were done by ourselves.

To avoid the high cost of operating an actual aircraft in the real environment, we collected the dataset in the X-Plane 11 simulation environment wherein the graphics and the dynamics of the environment and the aircraft are accurately modeled. For a preliminary assessment, we fixed the runway to KMWH-04 and the aircraft to be Cessna 208B Grand Caravan, while varying the position and the angle of the aircraft together with the weather condition. We used 40,000 samples for training with some realistic image augmentations—such as brightness, contrast, blur, vertical affine transformation—turned on in order to maximize the utility of the training data and create a more robust model.

3.2.2 Model Configuration

The accuracy of a neural network depends on many factors including the amount and quality of training data, the structure of the network, and the training process. As the performance of our proposed prioritization techniques may also depend on these

factors, we treated the structural configuration as an independent variable. However, since it is infeasible to compare the effect of all the independent variables to the prioritization techniques, we configured a number of representative neural networks with different structures. We controlled the other hyper-parameters—such as learning rate and mini-batch size—to be constant across different configurations so that the effect of the structure alone can be studied. The hyper-parameters are configured according to the known good practices at the time of writing this paper so that we can objectively simulate a realistic testing scenario.

Table 3.1: Four digit classification models

Model	Train Params	Model Structure	Train Epochs	Val. Acc.	EMNIST Acc.
A	594,922	2 Conv2D - MaxPool - 2 Conv2D - MaxPool - Flatten - Dropout - 2 Dense	82	99.16%	95.74%
B	177,706	2 Conv2D - MaxPool - Flatten - Dropout - Dense x3	93	98.90%	89.66%
C	728,170	2 Conv2D - MaxPool - Flatten - Dropout - 3 Dense	138	98.81%	86.14%
D	111,514	Dense - Dropout - 3 Dense	102	97.74%	72.90%

For the digit classification task, we configured four networks as described in Table 3.1. For all layers except the last one, ReLu (rectified linear unit) was used as an activation function, and L2 kernel regularization was applied to prevent the parameters from over-fitting. During training, we check-pointed the epoch only when the validation accuracy (with the 10,000 validation set) improved over the previous epochs, and stopped the training when the validation accuracy did not improve for more than twenty consecutive epochs.

For the taxiing task, we compare two different networks named MobileNet and SimpleNet, supplied by our industry partner. MobileNet is a convolutional neural network

inspired by MobileNetV2 [118]. The structure of the network is similar to what is described in Table 2 in Sandler *et al.*'s paper [118], and it is relatively compact in size, with 2,358,642 trainable parameters. SimpleNet is also a convolutional neural network, with a simpler structure, but with 4,515,338 trainable parameters. It has five sets of convolution, batch normalization, and activation layers back-to-back, followed by a dropout layer and four dense layers. Both of the networks implement L2 regularization, and trained with stochastic gradient-descent algorithm with weight decay.

3.2.3 Measure of Effectiveness

An ideal prioritization technique would consistently assign high scores to all the error-revealing inputs and low scores to all the rest. For example, if there were 20 prioritized test inputs among which 5 were error-revealing, ideally, the first five inputs should all reveal errors and the rest should not. If we draw a graph of the cumulative sum that represents the cumulative number of errors revealed by executing each prioritized input, the graph will be monotonically increasing until it hits 5, which is the total number of error-revealing inputs in the given test suite (the orange line in Figure 3.1). A test suite without prioritization might produce a line like the blue line. In practice, the efficacy of a prioritization technique will be somewhere between random selection and an ideal prioritization, producing a curve that looks like the green line in Figure 3.1. The efficacy of a prioritization technique can then be captured by computing the area under the curve for each technique and computing the ratio of each to the area under the curve of the ideal prioritization criterion. This is a slight modification to the Average Percentage of Fault Detected (APFD) measure, which is typically used for measuring the efficacy of test prioritization in the literature [119, 120].

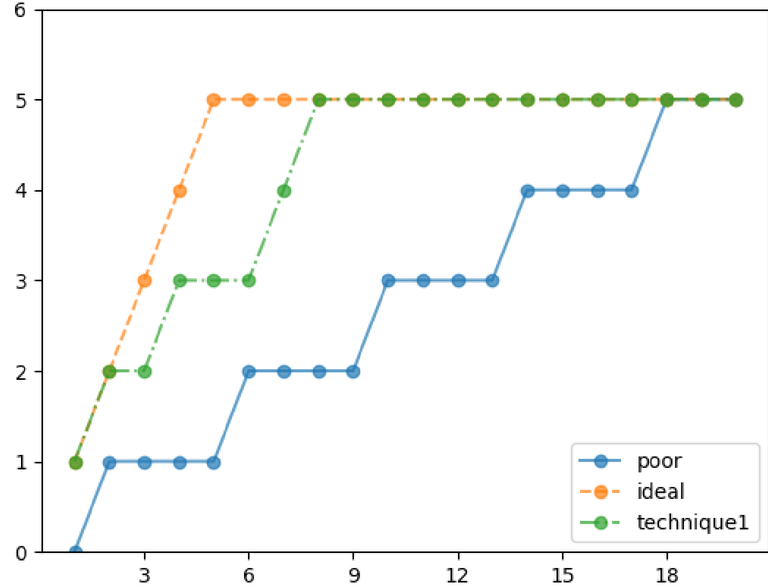


Figure 3.1: Cumulative sums of the errors found by test suites prioritized by each method. The x-axis corresponds to the test case, where the priority is higher for the former ones. The y-axis is the total number of errors found by executing the test cases up to x test cases. The efficacy score of *technique1* is the ratio of its area under curve to the area under the *ideal* curve: $83/90 = 92.23$.

3.2.4 Implementation and Experiment Environment

We implemented the three prioritization techniques—softmax, dropout Bayesian, and Surprise Adequacy—in Python on top of Keras [121], which is one of the most popular machine-learning libraries. Our tool is thus compatible with any trained model that abides by Keras' Model interface. The surprise adequacy measurement part is implemented in C++ to better utilize lower-level performance optimizations and thread-based parallelization. Every feature is integrated seamlessly and provided as a Python API. The tool is publicly available on GitHub at `http://www.github.com/bntejn/keras-prioritizer`.

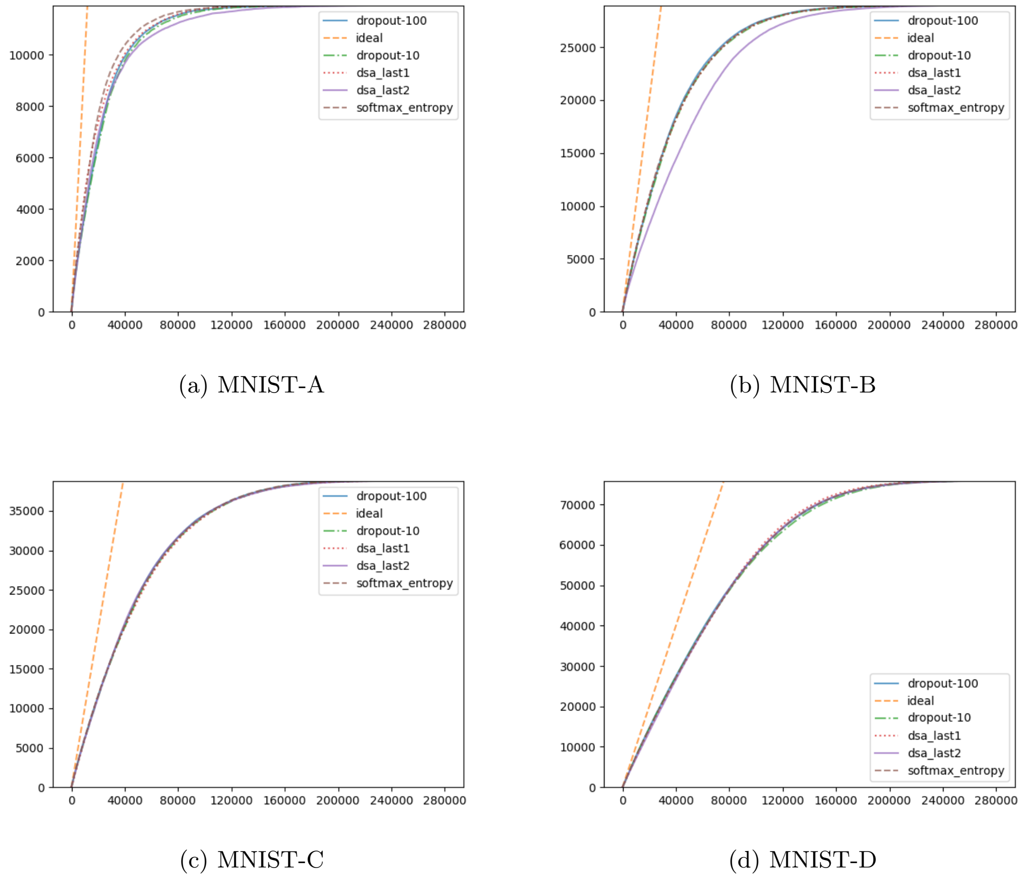
The experiments are performed on Ubuntu 16.04 running on an Intel i5 CPU, 32GB

DDR3 RAM, an SSD, and a single NVIDIA GTX 1080-Ti GPU.

3.3 Result

We ran our prioritization tool on the test datasets for all the trained models of MNIST and TaxiNet and measured the prioritization effectiveness in terms of finding failure and retraining improvement.

3.3.1 RQ1: Effectiveness of prioritization in finding failures



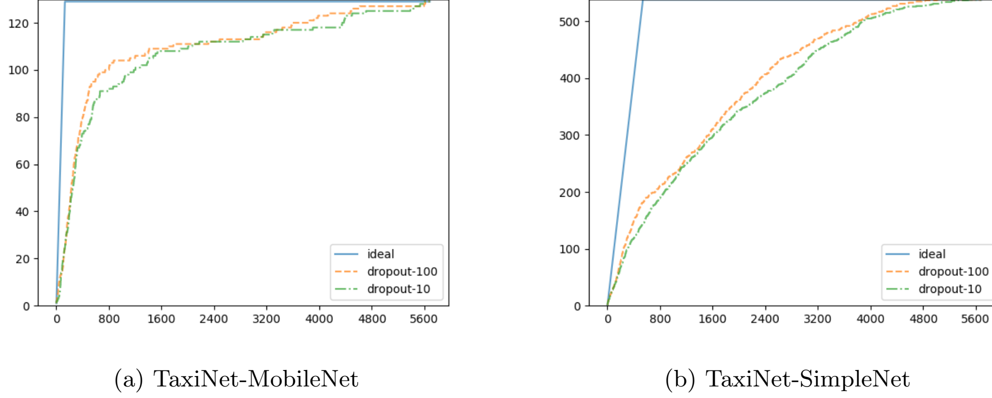
(a) MNIST-A

(b) MNIST-B

(c) MNIST-C

(d) MNIST-D

Figure 3.2: The cumulative sum of the failure-revealing inputs by test inputs of decreasing priority (Average Percentage of Fault Detection): The x-axis represents the test cases sorted in a decreasing order of priority. The y-axis shows the cumulative sum of failure-revealing inputs. An ideal prioritization should sort every failure-revealing inputs to the front, drawing a highly convex curve. A poor prioritization, on the other hand, will produce a curve with lower convexity.

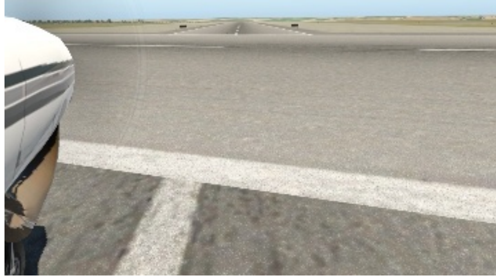


(a) TaxiNet-MobileNet

(b) TaxiNet-SimpleNet

Figure 3.3: The cumulative sum of revealed failures in prioritized suites for TaxiNet Models

The effectiveness of prioritization in finding failures is illustrated in Figure 3.2 and summarized in Table 3.2. For each model we present the validation accuracy, test accuracy, and the score of prioritization in Average Percentage of Faults Detected (APFD) as described in Section 3.2.3. The accuracy of classification is presented as the percentage of correct classification, and the accuracy of regression is presented as mean absolute error (MAE). The MAE for TaxiNet is defined as $\frac{1}{n}\Sigma_{i=1}^{n}|\mathbf{f}^{\mathbf{W}}(\mathbf{x})_i - \mathbf{y}_i|$ where n—the length of the output vector—is two for TaxiNet. The output of TaxiNet is normalized to be between -1 and 1, so the MAE is always between 0.0 and 1.0 where a lower error is more desirable. We also present the accuracy as a percentage; the correctness of an output is determined by a fixed error threshold on MAE of 0.25.



(a) High priority input

(b) Low priority input

Figure 3.4: Representative inputs of high vs. low priority: high utility score was assigned to inputs that produce high uncertainty—in this case, due to the lack of visible center line in the runway image taken around an intersection.

Table 3.2: The efficacy of test input prioritization in finding failures

Dataset	Architecture	Validation Accuracy	Test Accuracy	Sentiment Measures				
				Softmax	Dropout		Surprise (DSA)	
					10	100	last1	last2
MNIST	A (CNN + Batch norm)	99.16%	95.74%	**94.80**	93.20	93.57	94.26	92.80
	B (CNN)	98.90%	89.66%	91.10	90.87	**91.21**	90.99	87.06
	C (CNN)	98.81%	86.14%	89.30	89.09	**89.35**	88.98	89.26
	D (fully-connected)	97.74%	72.90%	87.90	87.58	88.02	**88.13**	87.77
TaxiNet	MobileNet	0.0394 (99.90%)	0.0764 (97.73%)	_	82.84	86.16	_	_
	SimpleNet	0.0575 (99.66%)	0.1243 (90.53%)	_	74.91	77.56	_	_

The efficacy of prioritization presented as APFD scores ranged from 74.91 to 94.80 over the different models, which suggests that test input prioritization works in identifying failure-revealing inputs, regardless of the type of task and the structure of the network. Among the different techniques, the efficacy of softmax, dropout Bayesian, and DSA were all similar for the same model, but the efficacy of the dropout Bayesian method was higher with more samples as larger samples can be used to more accurately depict the posterior distribution.

The efficacy of the prioritization metrics is correlated with the test accuracy of the model, or more precisely, the difference of the validation accuracy and the test accuracy of the model. The APFD was consistently high for the well-performing models and consistently low for the worse-performing models, regardless of the choice of sentiment measure. One plausible cause for this phenomenon is covariate shift [122], which is a situation when the distribution of the input data shifts from training dataset to test dataset. A stark decrease in the test accuracy for some models suggests that the distribution of the data shifted from the training data to test data, and some models (such as A and B) are relatively robust to the shift while the others are not. Model A, for instance, implements batch normalization, a technique known to reduce the internal covariate shift [123] and was more robust to the covariate shift in input distribution, which contributed to a higher prioritization effectiveness. In conclusion to our first research question:

> Prioritized inputs can effectively identify erroneous behavior in a trained model. The prioritization is more effective when the model has higher test accuracy. Our experiment indicates that the effectiveness is more dependent on the model rather than the choise of sentiment measure.

3.3.2 RQ2: Effectiveness of prioritization in retraining

We assess the utility of the prioritized inputs when a model is retrained with the training dataset augmented with the prioritized inputs. This evaluation is similar to the active learning scenario but different in that the model under test is already well-trained. We only sample 1% of the amount of training examples from the test set, which equals to 600 data points for the MNIST models and 400 data points for the TaxiNet models. The

baseline approach we compare against is random selection with the same sample size—we hypothesize that prioritization techniques perform better than the random selection. Hyper-parameters other than the augmented training data were kept constant in the retraining runs.

Table 3.3: The efficacy of input prioritization in retraining

Dataset	Architecture	Validation Accuracy Baseline	Test Accuracy Baseline	Sentiment Measures				
				Softmax	Dropout		Surprise (DSA)	
					10	100	last 1	last 2
MNIST	A	99.23%	97.48%	97.95%	98.09 %	98.06 %	98.18%	**98.44**%
	B	98.90%	95.66%	96.78%	**97.41**%	96.26%	97.35%	96.48%
	C	98.70%	95.02%	94.65%	**95.70**%	94.50%	95.02%	94.45%
	D	97.80%	90.57%	89.93%	87.58%	88.26%	91.87%	**92.00**%
TaxiNet	MobileNet	0.0336 (100.00%)	0.0364 (99.86%)	_	99.84%	**99.89**%	_	_
	Simple	0.0502 (99.85%)	0.0522 (99.05%)	_	**99.71**%	99.56%	_	_

Table 3.3 shows that the relative efficacy of retraining follows a similar trend to the failure-revealing efficacy presented in Table 3.2. The prioritized inputs could improve the accuracy of the retrained models more effectively than randomly sampled inputs in most cases, and the efficacy was more pronounced for MNIST model A and B than for model C and D. When model B and C are compared, model B consistently performed better when retrained with prioritized inputs while model C almost always performed worse when retrained with randomly sampled inputs. One hypothetical explanation could be that a well-architectured DNN model with a better generalization benefits more from learning the corner-cases, whereas an unoptimal DNN learns more from general cases. However, the exact reason for this phenomenon cannot be drawn from the limited experiment—a future investigation is necessary. In conclusion to the second research question:

> Sentiment measures can prioritize inputs that can augment the training dataset with which a better accuracy can be achieved. But random sampling was found more effective for the models that achieve low test accuracy.

3.3.3 Threats to Validity

In the experiment, we evaluated the sentiment measures with both an image classification task and an image regression task, and configured several DNNs with various structural features. Despite our effort, the representativeness of the configured DNNs were inevitably limited in number and variety, and our empirical findings might not generalize to other types of DNNs such as recurrent neural nets. Nevertheless, the DNNs we evaluated are of realistic sizes and implement some of the most widely used techniques that are applied in practical deep learning practices [1].

The second experiment for answering RQ2 was performed without the statistical rigor required for hypothesis testing due to the prohibitive cost of retraining a large model multiple times. We present the result and the finding as a preliminary assessment of the sentiment measures in the context of testing which calls for more rigorous empirical assessment in future work.

3.4 Conclusion

This chapter presented techniques for mitigating the oracle problem in testing DNNs by prioritizing error-revealing inputs based on white-box measures of DNN's sentiment—softmax confidence, Bayesian uncertainty, and input surprise. We evaluated the three techniques on two example systems for image classification and image regression, and multiple versions of the DNNs configured with different architectures. The experiment showed that the sentiment measures can prioritize error-revealing inputs with an average fault-detection rate of 74.9 to 94.8, indicating that input prioritization based on sentiment measures is a viable approach for effectively identifying weakness of trained models with reduced labeling cost.

We firmly believe that more attention should be paid to techniques that can facilitate field testing of safety-critical DNNs, which can be a laborious process and that test prioritization is an important step towards that goal, providing practical utility and good scalability. Further research is still needed for assessing the representativeness and completeness of test sets with respect to the operational environments of DNN-based systems.

Chapter 4

Black-Box Testing for Deep Neural Networks

Software testing approaches are traditionally categorized as either *white-box*—techniques that utilize the internal structure of the program under test—or *black-box*—techniques that do not require any knowledge of the internal structure of the program under test. For assessing test adequacy, white-box criteria measure the coverage over the structure of the program, whereas black-box criteria measure coverage of high-level artifacts such as requirements, specification, or the input domain [124, 125]. While white-box criteria have strengths, including being effective at finding implementation defects [52], when relied on exclusively, they entail the risk of missing the forest for the trees—e.g., missing functionality may escape undetected. Black-box testing provides a complementary top-down perspective, needed to overcome those short-comings. Both are thus necessary, and in practice, used in tandem.

A majority of recent ML Testing research can be categorized as white-box testing, as described in Section 2.3.2, and relatively little attention has been paid to techniques independent of the structure of the DNN under test. The reasons why black-box testing is desired are as follows:

- The DNN under test changes easily and frequently in the ML workflow (Figure 2.4). Model-dependent analysis needs to be performed again every time after there are changes in the model under test.

- DNNs are trained top-down algorithmically, unlike traditional software that is composed bottom-up by human programmers. A trained model, which is analogous to the implemented program in the traditional testing context, lacks the human-interpretable low-level constructs that can be isolated and analyzed.

Given these characteristics of DNNs, we investigated black-box approaches for testing DNNs. A black-box approach utilizes software artifacts other than the implementation, often of a higher level of abstraction, such as the requirement, specification, behavior model, and input domain. Given that data lies at the core of the ML workflow, serving as an instance-based specification of the task that a model has to learn, we deem it essential for our black-box technique to utilize this (training) data in some form. For simple tasks where the data dimensionality is relatively low—for instance, when there are only a couple of real-valued variables as inputs, traditional input-based testing approaches may be applicable, such as the category partition method [126] and combinatorial testing [127]. However, modern DNNs frequently have to deal with much larger input dimensionality. For instance, the input data of the MNIST [116] dataset, which is the "hello world" of image classification, belongs to $\mathbf{x} \in \mathbb{R}^{784}$. A naive application of input-based testing techniques does not scale. Further, even if we could, for instance, apply category partitioning on the input variables that, in this case, are pixel values for the MNIST hand-written digits, it does not *make sense* to apply partitioning at such a granularity, as the semantics of the input arise from multiple pixels, and a local view of a subspace in the input data cannot capture much meaningful information. For our input-based ML testing technique to work, we need a mechanism that can abstract the input domain, thus creating a domain model [128] that summarizes the semantics of the high-dimensional data in a low-dimensional space.

4.1 Representation (Manifold) Learning

There are key concepts that support the creation of a domain model: *manifold* and the *manifold hypothesis*. *Manifold* refers to a topological space that is locally Euclidean (e.g., the surface of the Earth). The *manifold hypothesis* assumes that real world data X presented in high-dimensional spaces $\mathbb{R}^{d_X}$ are expected to concentrate in the vicinity of a manifold M of a much lower dimension $d_M \ll d_X$ embedded in $\mathbb{R}^{d_X}$ [129]. In

other words, it assumes that the high dimensional data—such as an image—can be mapped to a much lower manifold dimension, and a data-generating distribution $P(X)$ can be modeled along the manifold structure that is locally Euclidean. With this, high-dimensional data can be explained with a number of factors that is much smaller than the original dimensionality of the input space. **Manifold learning** tries to capture such mapping so that a complex dataset can be encoded into a meaningful representation in a smaller dimension, serving several purposes such as, for example, data compression and visualization [130].

Another category of techniques that supports the construction of a domain model is representation learning. Representation learning [129] is:

> "A topic of Machine Learning that is concerned with learning representations of the data that makes it easier to extract useful information when building classifiers or other predictors. In the case of probabilistic models, a good representation captures the posterior distribution of the underlying explanatory factors for the observed input."

One reason why learning such representation is interesting is because they can conveniently express many general priors (beliefs) about the world around us [131]. Representation learning techniques aims at *learning* a representation that conforms to the general priors, some of which are briefly discussed below:

- Smoothness: Assumes that the function f to be learned is such that $x \approx y \implies f(x) \approx f(y)$.

- Multiple explanatory factors: The data is generated by different underlying factors, and one factor generalizes in many configurations of the other factors. The objective is to recover or disentangle these underlying factors of variations.

- Semi-supervised learning: With inputs X and label Y to predict, representations that are useful for $P(X)$ tend to be useful when learning $P(Y|X)$, allowing sharing of statistical strength between the unsupervised and supervised learning tasks.

- Manifolds: Probability mass concentrates near regions that have a much smaller dimensionality than the original space where the data live.

- Natural clustering: Local variations on the manifold tend to preserve the value of a category, and a linear interpolation between examples of different classes in general involves going through a low-density region.

- Sparsity: For any given observation x, only a small fraction of the possible factors are relevant.

For the purpose of our study, which is to construct a domain model and utilize it for testing activites, we assume a *good* representation/manifold learning technique that satisfies these priors. This representation learning technique shall produce a parametric mapping function q called encoder that maps from the input domain X of dimension d_x to an arbitrary encoding codomain M of dimension d_m, as $q : \mathbb{R}^{d_x} \rightarrow \mathbb{R}^{d_m}$, where $d_m \ll d_x$. We refer the encoding space $\mathbb{R}_{d_m}$ as the **manifold space**. The encoder shall work for any unseen data as well, producing an encoded representation in the manifold space. In the following subsections, we explain testing approaches that utilize this manifold.

4.2 Variational Autoencoder

VAE is a latent-variable generative model capable of producing outputs similar to inputs by determining a latent-variable space Z and associated probability density function (PDF) $P(z)$. The goal of a latent-variable model is to make sure that, for every datapoint x in a given dataset X, there is one or more settings of the latent variables z in a space Z that causes the model to generate $\hat{x}$ that is very similar to x. This goal is achieved by optimizing the parameter θ for a deterministic function $f : Z \times \Theta \rightarrow X$ such that the random variable $f(z; \theta)$ produces outputs similar to $x \in X$ when z is sampled from $P(z)$. In other words, we maximize the likelihood of producing X when X is conditioned by Z: $P(X) = \int P(X|z; \theta)P(z)dz$; here, a PDF $P(X|z; \theta)$ replaces $f(z; \theta)$. VAE does not assume a specific distribution for $P(z)$, but rather assumes that any probability distribution in the space Z can be represented by applying a sufficiently complicated function f_θ to a set of normally distributed variables z. With a set of decoder parameters θ, the probabilistic decoder of a VAE is given by:

$$P_\theta(x|z) = \mathcal{N}(x|f_{\mu_x}(z; \theta), \gamma I) \tag{4.1}$$

where γ is a tunable scalar hyperparameter—which is typically set as 1 to represent multivariate unit Gaussian distribution—and I is the identity matrix. We set γ as a trainable parameter, as a high γ is proven to be responsible for blurry images generated by VAEs, which was often considered as a practical limitation of VAEs [132].

For modeling the unknown PDF of latent variables $P(z|x)$ from which to run the decoder $P_\theta(x|z)$, we need a new PDF $Q(z|X)$ which can take an x and return a distribution over z that are likely to produce x. This $Q(z|x)$ is called probabilistic encoder, which is given by:

$$Q_\phi(z|x) = \mathcal{N}(z|g_{\mu_z}(x;\phi), g_{\sigma_z{}^2}(x;\phi)) \tag{4.2}$$

where ϕ is a set of encoder parameters and g is an encoder function approximated by a deep neural network. g is designed to produce two outputs g_{μ_z} and $g_{\sigma_z{}^2}$, which are mean and variance of the encoded z. In other words, g encodes each $x \in X$ as a distribution, where the mean g_{μ_z} has the highest probability of being reconstructed to x.

As $P(z|x)$ was assumed as multivariate Gaussian, the posterior distribution $Q_\phi(z|x)$ shall *match* the $P(z|x)$ so that we can relate $P(x)$ to $\mathbb{E}_{z\sim Q}P(x|z)$, or the expected value of generated input x given a latent variable z when z is sampled from the space encoded by encoder PDF Q. This is achieved by optimizing the following VAE loss function:

$$\mathcal{L}(\theta,\phi) = \int_{\mathcal{X}} -\mathbb{E}_{Q_\phi(z|x)}[\log P_\theta(x|z)] + \mathbb{KL}[Q_\phi(z|x)||P(z)]\mu_{gt}(dx) \tag{4.3}$$

where $\mu_{gt}(dx)$ is the ground-truth probability mass of a dx on X, which leads to $\int_X \mu_{gt}(dx) = 1$. The term $-\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z)]$ is the reconstruction cost, which penalizes poor reconstruction inputs in the input dataset. The term $\mathbb{KL}[q_\phi(z|x)||P(z)]$ is the Kullback-Leibler divergence between the encoder distribution and the prior distribution, which penalizes deviations from the distribution $P(z)$. [132]. We refer more curious readers to a tutorial on VAE [133].

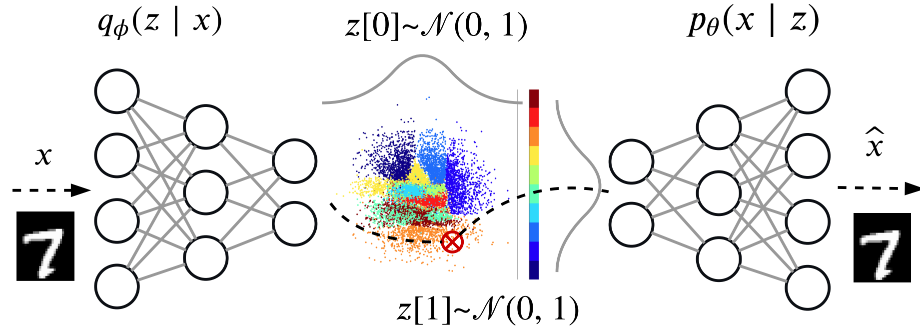


Figure 4.1: Variational Autoencoder

A VAE example Figure 4.1 illustrates the structure and the operation of a VAE with $\kappa = 2$ as the size of the latent dimension, and $\gamma = 1$. VAE encodes x as $z \sim \mathcal{N}(0, I_2)$, where I_2 is a 2×2 identity matrix, forming a circle-like mappings to the two-dimensional plan. As 99.73% of the datapoints fall inside the range $[-3\sigma, 3\sigma]$, the majority of the datapoints fall inside a circle of radius $3\sigma = 3$. The datapoints mapped in the plane shows that digits in the same class—color-coded from 0 (dark blue) to 9 (dark brown)—cluster together, illustrating that digits that *look* similar are encoded to be close to each other in the latent space. From the areas where different colors are mixed together, such as where mint-colored points representing digit 4 and dark-brown-colored points representing 9 are mixed in an adjacent space, we can infer that many fours and nines look similar to each other. If we sample new points from this subspace, the generated image may look somewhat like 4 and 9 at the same time.

Conditional VAE A basic VAE can generate images but not labels. Thus, it may be useful for test input generation, but without the labels, much time has to be spent assigning labels to solve the oracle problem. When implementing a VAE, note that the encoder $q_\phi(z|x)$ is conditioned solely on the inputs x, and similarly, the decoder $p_\theta(x|z)$ models x solely based on the latent-variable vector z. Conditional VAE (CVAE) implements a conditional variable c in both the encoder and decoder [134]. This yields the new loss function:

$$\mathcal{L}(\theta, \phi) = \int_{\mathcal{X}} -\mathbb{E}_{q_\phi(z|x)}[\log p_\theta(x|z, c)] + \mathbb{KL}[q_\phi(z|x, c)||P(z|c)]\mu_{gt}(dx) \qquad (4.4)$$

Note that $P(z)$ is now distributed under $P(z|c)$, a conditional distribution on c. Both the encoder and decoder are conditioned on c as well, which gives a specific distribution $P(z|c)$ for each class c. When training a CVAE for a classification task, we choose the values of c to be the class labels of the dataset. By sampling z from $P(z|c)$, we significantly increase the probability of obtaining a latent-variable vector z_0 such that $p_\theta(x, z_0, c)$ is a valid image of class c [135].

Two-stage VAE One drawback of the basic VAE is the inability to accurately reproduce the distribution of ground-truth data $P_{gt}(x)$, even with perfect reconstruction loss. Although the prior distribution $P(z)$ is a Gaussian, the encoder projects the ground-truth distribution as $Q(z)$ which is not necessarily Gaussian at the global optimum. As a result, when generating synthetic outputs using a basic VAE, the distribution of sampled latent vectors from $P(z)$ does not match the distribution of the ground-truth latent vectors $Q(z)$, and the ground-truth distribution X is not accurately reproduced.

In order to address this problem, Dai and Wipf introduced two-stage VAE [132], which makes use of a second-stage latent space U and associated density function $P(u)$. Simply speaking, after training the (first-stage) VAE with Equation 4.3 to generate $\hat{x}$, another (second-stage) VAE is trained with the following second-stage loss function:

$$\mathcal{L}(\theta', \phi') = \int_Z -\mathbb{E}_{Q'_{\phi'}(u|z)}[\log P'_{\theta'}(z|u)] + \mathbb{KL}[Q'_{\phi'}(u|z)||P'(u)]\mu_{gt}(dz) \tag{4.5}$$

Note that it is in the same form as Equation 4.3, with the θ, ϕ, Q, P, z, and x replaced with θ', ϕ', Q', P', u, and z, respectively. In other words, this second-stage VAE is trained with Z as the input dataset, and learns u as latent variables with which to encode Z. This second-stage VAE resolves the discrepancy between the prior distribution $P(z)$ and posterior distribution $Q(z|x)$ by introducing a second-stage latent distribution $P'(u)$ from which to sample new inputs. The second-stage latent distribution is proven to fit better to Gaussian prior, such that when new inputs are sampled from $u\ \mathcal{N}(0, I_\kappa)$ with κ being the size of the latent dimension, the reconstructed $\hat{z}$ lies in the ground truth distribution $Q(z)$. The $\hat{z}$ is then fed to the first-stage decoder, which generates $\hat{x}$ with a high $P(x)$.

The VAE discussed above will be used as a basis of the testing techniques introduced in the subsequent chapters.

Chapter 5

Black-Box Coverage Criterion

As described earlier in Section 2.3.2, every known DNN test adequacy criterion, to the best of our knowledge, is white-box in some form, requiring the DNN model under test (MUT) to be involved during adequacy measurement, and its internal structure to be utilized. Though potentially useful, the rationale for using the DNN structure as a basis for coverage measurement is not as apparent as structural criteria for traditional programs. For example, what does it *really mean* that a neuron is, or is not, covered, and what should one do about it? For instance, how could one know what test case should be added in order to cover an uncovered neuron? This is counter-intuitive unlike the white-box criteria for traditional programs, where an uncovered statement, branch, condition, decision, or anything else, can be analyzed, and from which a test input that can cover it can be deduced manually or automatically. Moreover, these criteria have the same limitations as the traditional white-box criteria in their dependence on the program under test, with the negative impact further exacerbated due to the highly iterative nature of the ML development workflow (Figure 2.4). Small changes to the DNN structure or weights can have large consequences for the measured coverage, invalidating any analysis done with prior coverage information.

Given the known drawbacks of the white-box criteria, we in this section propose a novel black-box coverage criterion called *Manifold Combination Coverage* (MCC) that measures the test adequacy on a *manifold*. Similar to black-box testing in a traditional sense, the use of MCC has the benefit of being independent of the model under test, is inexpensive to measure, scales well, and is interpretable by human.

While preparing to empirically assess MCC, we noticed that a standardized approach to evaluate and compare ML test adequacy criteria, taking into account how they are to be used in practice, is missing. Therefore, we first clarify the usages of coverage criteria in the ML workflow and define three metrics for empirically comparing the coverage criteria as indicators of test suite quality— *failure-revealing effectiveness* (how well a test suite triggers failures), *semantic balance* (how well a test suite resembles the expected input distribution), and *retraining efficacy* (how well a test suite *fixes* bugs via retraining). Using these metrics, we experimentally compare MCC and several white-box criteria on nine realistic models trained for MNIST [116], CIFAR-10 [136], and Udacity [137]. In the experiment we also use the test prioritization techniques to investigate how the prioritization influences the effectiveness of coverage criteria.

5.1 Limitations of DNN White-Box Coverage Criteria

To make a stronger case for a black-box coverage criterion, we discuss the limitations of white-box testing in more detail. Due to the difference between the ML workflow and a traditional software development life-cycle, white-box testing of DNNs has several shortcomings, such as a) its model-dependence, b) high cost of measurement, and c) its lack of interpretability.

Model Dependence The ML workflow is highly iterative (Figure 2.4). It is necessary to experiment with different configurations to obtain a good model. Unlike in a traditional software development life-cycle where the program evolves incrementally, the structure of a DNN can change completely by changing only a few settings among many moving parts—such as training data, the DNN architecture, and hyper-parameters. This nature of ML development makes white-box testing less suitable. For example, suppose a scenario of testing two different DNNs with neuron coverage (NC) where the first model dominantly uses *sigmoid* for the activation function whereas the second model uses *tanh*. If we set the threshold of NC to 0.0, the first model will achieve a near 100% coverage with a handful of test inputs as the range of sigmoid function is greater than 0. On the other hand, the second model will achieve much lower coverage with the same set of inputs because the range of *tanh* function is $[-1, 1]$. Although

the first model with sigmoid achieve higher coverage with the same set of inputs, high coverage does not mean that the testing was done more thoroughly on the first model, nor does it mean that the first model has a higher quality. In other words, the coverage score measured on one model becomes meaningless when the structure of the model changes, and the coverage loses its value as a quantitative measure of test adequacy. It means that even for two models that are equivalent in performance, when tested with the same test suite, the reported adequacy scores can be dramatically different, while this difference is not necessarily associated with the adequacy of the testing.

High Measurement Cost The time and space complexity for measuring the coverage for many popular DNN coverage criteria are linear in the number of neurons inside a DNN. The size of DNNs used in practical applications can be enormous, ranging up to a few tens of million parameters for computer vision models [138]. As the complexity is also linear in the number of test cases and in the number of models to test, the overall cost quickly becomes prohibitively expensive. For example, the coverage vector for *a single test run* on a 100M-parameter model consumes $10^8/(8 \times 2^{20}) = 11.9$ MB of memory space, accompanied with the I/O overhead of copying the DNN intermediary output from GPU to RAM. In fact, the experiment performed in DeepGauge [19] for evaluating various white-box criteria had to employ a compute cluster where each node is equipped with 196GB-RAM—a non-trivial cost for coverage measurement.

Lack of Interpretability Coverage criteria are often used for guiding test case creation or test selection. In the case of white-box testing a traditional piece of software, one can comprehend the implications of covered and uncovered obligations and logically reason about them so that more test cases can be created for achieving a higher coverage. With white-box criteria for DNNs, however, this is not possible since we do not have a means to understand the semantic implications of covering a structural element of a DNN. They cannot help an engineer answer this important question: "which test case should I create or collect in order to increase the coverage?"

5.2 Manifold Combination Coverage

To address the shortcomings of white-box criteria, we propose a novel black-box criterion called Manifold Combination Coverage (MCC) that is analogous to combinatorial coverage [139, 140] of input domains in traditional testing. The key idea is to assess test adequacy based on the semantics of the input instead of the structure of the model under test. The gist of our approach is to map the high-dimensional input data to $\mathbb{R}^d$ for a small enough d using an encoding technique and to partition each dimension into k contiguous ranges such that the partitioning is *uniform* with respect to the sample density of a reference set (e.g., the training data set for the MUT), i.e., an arbitrarily chosen element of the set has equal probability of falling into any of the k partitions. MCC measured for a test suite then quantifies the fraction of the k^d possible combinations of the partitions that the test data in the test suite map to. With the two tunable parameters k and d, we define (k, d)-Manifold Combination Coverage as follows.

Definition 6 ((k, d)-Manifold Combination Coverage) *The* ***k-section d-dimesional manifold combination coverage*** *of an input data domain $\mathbb{X}$ with respect to an encoding function $q : \mathbb{X} \to \mathbb{R}^d$ and a binning function $\rho_k^d : \mathbb{R}^d \to \mathbb{Z}_k^d$, is defined as a real-valued function $\mu : 2^{\mathbb{X}} \to [0, 1]$ given by*

$$\mu(A) = \frac{\left|\left\{\rho_k^d(q(x)) \mid x \in A\right\}\right|}{k^d} \tag{5.1}$$

for any $A \subseteq \mathbb{X}$.

$\mathbb{Z}_k$ denotes the set $\{0, ..., k-1\}$. The function $q(\cdot)$ encodes input data to the manifold space and the function ρ_k^d maps the low-dimensional data with the unique combination of the partitions of the dimensions that it falls into. The binning function ρ_k^d divides each dimension into k contiguous ranges containing equal training sample density per bin in each dimension. If the implicit semantic features of the dataset can be learnt as a manifold, and if k partitions are seen as distinct categories for each feature, then manifold combination coverage requires the inclusion of all possible interactions of those features, analogous to combinatorial testing of traditional software that can expose failures that arise from the interaction of multiple input variables or features. Since

semantic closeness is preserved as Euclidean distance in the manifold space, MCC also ensures that duplicate samples do not count for coverage.

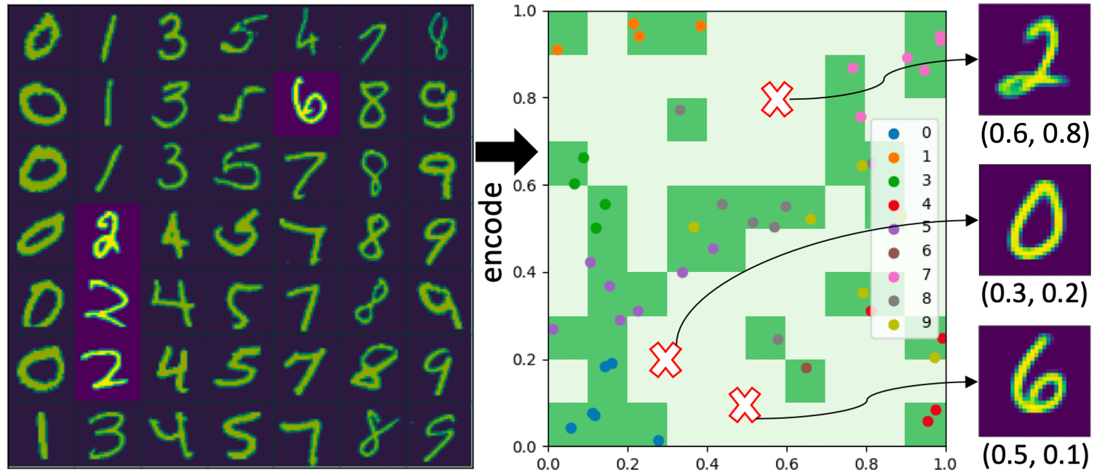


Figure 5.1: Coverage analysis on 2D Manifold Space.

Figure 5.1 illustrates the concept with $(10, 2)$-MCC for testing a model trained with the MNIST dataset [116], where $d = 2$ was chosen for a 2-D visualization. The left side of this figure shows $7 \times 7 = 49$ test inputs that are sorted in increasing order by the class label. There are 45 darker-colored hand-written digits that represent selected test inputs, and four lighter-colored ones that represent left-out test inputs. The 45 selected test inputs are encoded to the manifold space, and presented in the 2D grid on the right side of the Figure. These 45 inputs are color-coded by their class label, and it can be seen that the inputs that belong to the same class are clustered in the vicinity of each other. By our configuration of $(10, 2)$-MCC, with $k = 10$, each dimension is divided into 10 sections, creating 10^2 coverage obligations, depicted as a squared-shaped cells in the figure. The segments of the grid where there exist one or more encoded samples are highlighted in darker green. 34 out of 100 cells were *covered*, so the $(10, 2)$-MCC of the selected test suite calculates to 34%. The number of covered obligations is lower than the number of selected test cases, because some coverage obligations are satisfied by more than one semantically close test inputs. For example, there are four distinct selected inputs of class 1 that are encoded into four orange dots in the manifold space, but two of the four samples landed on the same cell. Those two samples, when identified in the

grid on the left side, are the digit at $(0, 1)$ and $(1, 1)$, which indeed look very similar. A more granular coverage configuration, such as when k was higher than 10, might make these two similar inputs correspond to different coverage obligations. Contrarily, a less granular coverage configuration, such as when $k = 2$ would map these four samples to the same coverage obligation. By setting a k value in accordance with the testing budget, or the required level of granularity, one can control the level of discrimination that MCC provides. For instance, when we want to create a test suite of size 100 that is representative of the $\mathcal{D}_{gt}$, we can perform coverage-based test selection with $(10, 2)$-MCC that can discriminate up to 100 samples.

The three red `X` marks in Figure 5.1 shows how the coverage can be *interpreted* to provide an intuitive guidance to the tester. The three coordinates are selected randomly within the manifold among uncovered obligations, and passed to the decoder [1] to synthesize new test inputs that correspond to the provided coordinates. For instance, from the coordinate $(0.3, 0.2)$, a hand-written digit of label 0 was synthesized, as shown in the second row on the right side of Figure 5.1. This means that when a test input that looks like this input was included in the selected test suite, it would have been encoded to $(0.3, 0.2)$ marking the coverage obligation that includes this point as covered. It can be also noticed that the input synthesized from $(0.3, 0.2)$ is similar enough to the nearby samples that are also labeled as 0s, but show a distinct visual feature compared to the other 6 0-labeled digits that are selected from the left. The other two `X`-marked samples can be explained similarly, and in this manner, all the other uncovered obligations can be visualized to provide further information to the tester about what kinds of inputs are needed to improve the coverage. This *interpretability* is a unique feature among DNN coverage criteria; for other criteria, such as Neuron Coverage, there is no known technique that can *explain* how to improve coverage, i.e., cover an uncovered neuron.

5.3 Evaluation Criteria

Unlike in traditional software, it is hard to discuss the effectiveness in terms of fault finding for DNNs since a fault in DNN is a product of multiple factors that influence

[1] A decoder p is an inverse of encoder q, such that $p : \mathbb{R}^d \implies \mathbb{X}$. In this example, a VAE decoder was used. See Appendix 4.2 for mor detail on VAE.

the trained model. Moreover, failure-revealing effectiveness itself is not the only goal because, when the purpose of testing is also to evaluate the performance of the model objectively, the semantic balance of the test suite—a low divergence between $\mathcal{D}_{test}$ and $\mathcal{D}_{gt}$—also matters. Hence, we capture the efficacy of coverage criteria from three different aspects—failure-revealing effectiveness, semantic balance, and retraining efficacy. These metrics are quantified over a test suite. The effectiveness of coverage criteria is measured indirectly through test suites created with coverage criteria under investigation.

First, the **fault-finding effectiveness** measures the effectiveness of a criterion in picking out test cases that reveal faults. Fault-finding effectiveness of a criterion c is quantified as the percentage of test cases that reveal generalization failures that are included in the test suite constructed with c:

$$\frac{|\{(x, y) \in T | f(x) \neq y\}|}{|T|} \tag{5.2}$$

For the regression tasks, we consider it as fault-revealing when the prediction error exceeds the threshold $||f(x) - y||_{l1} > \delta$.

Second, the **semantic balance** measures how a test suite matches the ground data distribution, which is important for objectively assessing the performance of the model without bias. For the benchmark dataset that we used, we assume that the training data follows the ground truth distribution. For each test suite $\mathcal{D}_{c_i}$, we can obtain a probability vector of semantic features $P_{feat}(\mathcal{D}_{c_i})$ and compute the divergence between $P_{feat}(\mathcal{D}_{c_i})$ and $P(\mathcal{D}_{test})$ with Jensen-Shannon divergence. We compute the score as:

$$1 - JSD(P_{feat}(\mathcal{D}_{c_i}) || P_{feat}(\mathcal{D}_{test})) \tag{5.3}$$

that evaluates to 1.0 when the two distributions are identical and converges to 0.0 when the two distributions diverge. In the proposed experiment, we plan to measure the distribution with respect to the class label, as that is the only annotated feature available for our datasets. For the DNNs that perform regression, we discretized the range of real-valued outputs.

Third, **retraining efficacy** measures the performance of a model after retraining with the respective test suites. While fault-finding effectiveness measures the number of fault-finding test cases, the numbers may not correspond directly to the number of

distinct faults, or the latent cause of the failures. Fault-revealing inputs are evidences of faults, and multiple evidences can actually stem from the same latent cause. As the number of faults cannot be measured directly, retraining efficacy measures how many of the failures caused by a lack of training data can be mitigated by adding the selected data to the training data set. We believe that this is an effective measure for assessing a coverage criterion, since in practice, one of the most effective and readily available solution for *fixing* a fault is to add more data. We assume that when a test suite TS_{c_1} selected by a criterion c_1 is better than TS_{c_2} selected by c_2, $\mathcal{D}_{train} \cup TS_{c_1}$ should yield a model of higher performance after retraining than when trained with $\mathcal{D}_{train} \cup TS_{c_1}$, given that every other variables such as epochs and learning rate are fixed. In our retraining experiments, we used the optimizer and the learning rate last used for training the original model, and retrained for 20 epochs. We also split the master suite into selection pool and test set with the ratio of 1:2, so that the retraining accuracy can be assessed with separate set-aside data. The size of TS_c was fixed at 1,000.

5.4 Experiment

We evaluated the efficacy of coverage criteria in test suite construction, either during on/off-line test data collection or during the selection of data to label. We postulate that the reason for using coverage criteria is to construct a set of test cases $\mathcal{D}_{test}$ that is 1) effective in finding faults, 2) representative and 3) minimal. First, the *failure-revealing effectiveness* is needed because the very goal of testing is to identify the discrepancy between the present and the desired conditions. Second, the *representativeness* is needed for a test suite to provide an unbiased measure of the performance of the model under test (MUT). Third, the *minimality* is desirable for the efficiency of testing. This can be achieved by avoiding duplication—i.e., when two test inputs cover the same set of coverage obligations, one is considered redundant. To assess these characteristics of each criterion with or without test prioritization, we ask the following research questions:

- **RQ1**: Is manifold coverage as effective as white-box criteria in terms of *failure-finding* effectiveness?

- **RQ2**: Is manifold coverage as effective as white-box criteria in creating *semantically balanced* test suites?
- **RQ3**: Is manifold coverage as effective as white-box criteria in creating a dataset for *retraining* the model?

5.4.1 Test Suite Construction

We compare the efficacy of coverage criteria indirectly by comparing the efficacy of test suites that are constructed with the help of each coverage criterion. The evaluation aims to simulate real-world use case of coverage criteria when used for test suite construction. We assume that in a typical use-case, coverage criteria are used as test selection criteria, or a predictor that determines which test case is of value. A test case that improves the coverage is selected, and a test case that does not is discarded. This greedy algorithm does not ensure the minimality of the down-selected test suites, but it resembles a realistic use case and widely adopted in existing testing research [57, 58, 141]. We also assume that the testing budget is finite, and that creating an *adequate test suite*—a test suite that achieves 100% coverage score—is impractical. Regardless of the achieved coverage score, we set a finite number of test cases to be the testing budget. For an objective comparison, every variable other than the coverage criterion itself is controlled. First, the pool from which test cases are selected—which we call *the master test suite*—is controlled and shared across different configurations. We constructed a master suite to include various kinds of test cases collected from different sources. Second, the order in which test cases are picked is controlled. Third, the testing budget, or the number of test cases to select, is controlled. Doing so eliminates the effect of test suite size on the efficacy of the constructed test suites [54].

5.4.2 Datasets and Models Under Test

The experiments are performed on three different computer vision tasks—two classification tasks and one regression task. A classification model outputs a Bernoulli distribution for the classes it predicts, and the class with the highest predicted probability is considered as the output of the model. A regression model outputs a set of real values.

The first task is ten-class hand-written digit classification, for which we trained

three models with MNIST dataset [116] consisting of 60,000 black-and-white images of 28 x 28 pixels. The second task is CIFAR-10 image classification [136]—the training data consists of 50,000 color images of size $32 \times 32 \times 3$. We trained three CIFAR-10 models of the same architecture. The third task is Udacity self-driving car dataset [137] which is a regression task for predicting the steering angle based on the dashboard camera images. We trained the same set of models, with 27,046 train data of size $96 \times 96 \times 3$. These models are implemented in and trained with TensorFlow 2.3 [142], and the implementations are adapted from the code written by Li [143]. The details of each model is described in Table 5.1. Their train, validation, and test performance is presented as classification accuracy for MNIST/CIFAR-10 and mean absolute error (MAE) for Udacity.

Table 5.1: Models under test

Task	Architecture	# Params	Train	Val.	Test
MNIST	LeNet [144]	44k	98.53%	98.86%	91.81%
	Network-in-Network [145]	957k	99.13%	99.11%	90.17%
	ResNet-32 [146]	468k	99.65%	99.41%	90.87%
CIFAR-10	LeNet	62k	72.43%	72.43%	68.87%
	Network-in-Network	967k	93.00%	87.90%	82.31%
	ResNet-32	468k	99.68%	92.36%	86.00%
Udacity	LeNet	54k	0.0304	0.0272	0.0610
	Network-in-Network	965k	0.0603	0.0614	0.0504
	ResNet-32	470k	0.0117	0.0102	0.0959

5.4.3 Experiment Configurations

Since DNNs lack convenient logic structure, most of the coverage criteria require configuration. As such, studying the impact of all the different configurations of each structural criterion is a challenging task, requiring an extensive empirical validation on its own. We instantiate a small number of representative configurations for each criterion.

Neuron-level Coverage

To allow for differentiating at least the number of test cases that we desire to collect—10,000 test cases, in our experiments—we instantiated NC with higher threshold values—1.5, 3.0, and 4.0. When the threshold value is set to a low value like 0, it becomes too easy to cover a majority of neurons with a small number of test cases. The implication of choosing the threshold value may change for a different DNN architecture, and it is especially susceptible to the choice of activation functions and batch normalization, as those dictate the range of intermediary neuron outputs. As KMNC is granular than NC, it can better differentiate test cases with higher setting of k. However, this merit makes KMNC become prohibitively expensive as the network grows larger in size; state-of-the-art CNNs are often comprised of as many as hundreds of millions of neurons. As an example, if we attempt to measure $k = 100$ MNC on a network with 100 million neurons, the number of coverage obligation becomes ten billion, which amounts to 1.25GB of storage space for strong a single bit vector that represents the coverage of a single test case. For these reasons, we limited our experiments to relatively small k values of 3, 10, and 20. We also limited the length of the coverage vector by measuring both NC and MNC on the last few layers of the larger neural networks—every layer for LeNet, last 7 layers for NiN, and last 11 layers for ResNet—since, those layers in CNNs are known to encode higher-level features [147]. Further investigation is needed to study the impact of the choice of layers.

Surprise Adequacy

SA is configurable in two main ways—selection of layer for the activation trace, and the granularity of coverage. For layer, we chose the last fully-connected layer before the output layer. For granularity, which determines how many test cases can be uniquely identified, we set it proportional to the number of test cases we desire to select for constructing a test suite. In order to ensure that the desired number of test cases are selected even when a test input does not exist for a specific surprise segment, we instantiated DSA1 with $n \times 2$ segments, and another $DSA2$ with $n \times 3$ segments, where n is the testing budget.

Manifold Combination Coverage

For the implementation of the VAE, we revised the open-sourced TensorFlow code [148] written by Dai *et al.* [132]. We used the same encoder and the decoder architecture inspired by InfoGAN [149] for the three datasets, and they are mainly composed of convolution and transposed convolution layers. The number of VAE parameters range from 13 to 38 million. We set the manifold dimension d to 7 for all three datasets. For configuring MCC, we instantiated two versions—$(5, 7)$-MCC and $(6, 7)$-MCC. These numbers were empirically determined based on our budget. The implication of changing these parameters is beyond the scope of this study.

Miscellaneous

For answering RQ3, we used the same optimizer and the learning rate last used for training the original model, and retrained for 20 epochs. We also split the master suite into a selection pool and a test set by 1:2 ratio, so that the retraining accuracy can be assessed with a separate set-aside dataset. The size of test suite T was fixed at 1,000.

5.5 Result

We present the raw data in Table 5.2. The columns show the coverage criterion used for constructing each test suite, and the rows show the specific MUT. The score is measured for the three evaluation criteria we proposed, and they are presented together by three groups of nine rows. In each row, the highest scores with and without prioritization are separately highlighted with bold-face, and the higher score (lower for MAE) between the two is underlined. Especially, to illustrate how the failure-revealing effectiveness changes as the size of the test suite grows, we visualized the correlation in Table 5.3. The efficacy with and without prioritization for the same setting is presented in the *Default* and *Prioritized* columns, respectively. The x-axis shows the test suite size as it grows by the coverage-guided selection, and the y-axis shows the cumulative number of failure-revealing test cases. The baseline (solid black) is a random selection without any guidance—with a truly random selection, the slope is equal to the percentage of failure-revealing test cases in the master suite. A coverage criterion is deemed effective if its average slope is greater than that of the baseline; the steeper the slope, the better.

A line that stops before reaching the end ($x = 10000$) means that the criterion failed to find additional test cases from the master suite to improve coverage. This is generally undesirable, since it is a sign that the specific instance of the criterion is too coarse.

Measure	Model	No Prioritization: Rand	MCC 1	MCC 2	DSA 1	DSA 2	NBC	kMNC 3	kMNC 10	kMNC 20	NC 1.5	NC 3.0	NC 4.5	With Prioritization: None	MCC 1	MCC 2	DSA 1	DSA 2	NBC	kMNC 3	kMNC 10	kMNC 20	NC 1.5	NC 3.0	NC 4.5
Failure-Revealing	M-L	.110	.071	.071	**.207**	.151	.014	.004	.018	.038	.001	.005	.005	**.410**	.209	.272	.239	.306	.025	.033	.099	.163	.012	.020	.023
	M-N	.115	.075	.077	**.244**	.191	.063	.022	.089	.107	.009	.013	.015	**.310**	.201	.247	.266	.295	.123	.087	.232	.290	.032	.040	.045
	M-R	.110	.077	.076	**.266**	.259	.045	.010	.039	.077	.019	.021	.019	**.409**	.221	.289	.263	.349	.116	.067	.180	.268	.067	.079	.069
	C-L	.114	**.117**	.115	.109	.114	.005	.013	.040	.073	.001	.004	.008	**.245**	.214	.232	.117	.160	.007	.041	.101	.164	.004	.016	.020
	C-N	.011	.114	.112	**.155**	.140	.039	.043	.093	.100	.012	.020	.024	**.372**	.297	.347	.253	.319	.050	.218	.337	.360	.060	.096	.118
	C-R	.112	.113	.116	**.178**	.138	.039	.036	.090	.101	.019	.037	.032	**.433**	.348	.402	.273	.351	.058	.214	.389	.414	.089	.163	.151
	U-L	.160	.345	**.467**	-	-	.204	.198	.199	.166	-	-	-	**.656**	.402	.517	-	-	.248	.363	.523	.599	-	-	-
	U-N	.174	.500	**.650**	-	-	.354	.320	.296	.226	-	-	-	**.783**	.535	.686	-	-	.411	.417	.631	.709	-	-	-
	U-R	.171	.309	**.400**	-	-	.155	.165	.350	.251	.100	.063	.044	**.557**	.311	.410	-	-	.159	.226	.398	.388	.125	.082	.046
Semantic Balance	M-L	.978	.951	**.976**	.921	.946	.848	.870	.865	.878	.796	.831	.895	.835	**.904**	.869	.766	.823	.790	.814	.828	.836	.713	.796	.836
	M-N	.980	.953	**.979**	.925	.953	.874	.910	.918	.953	.902	.921	.894	.865	**.911**	.875	.825	.871	.886	.864	.886	.881	.815	.856	.858
	M-R	.976	.976	.975	**.977**	.976	.967	.969	.972	.977	.960	.969	.951	.979	.977	**.982**	.972	.975	.980	.971	.977	.974	.961	.971	.972
	C-L	.990	.990	.989	.985	**.991**	.045	.972	.985	.989	.907	.946	.955	.806	.848	.819	.888	**.892**	.800	.864	.873	.886	.857	.820	.845
	C-N	.986	.989	.989	**.991**	.987	.989	.978	.988	.989	.943	.968	.975	.784	.852	.824	**.881**	.821	.855	.843	.822	.809	.835	.835	.847
	C-R	.985	.950	.977	.967	**.981**	.909	.917	.958	.972	.928	.924	.914	.897	.915	**.989**	.931	.926	.898	.885	.893	.894	.894	.898	9.883
	U-L	.637	**.807**	.770	-	-	.686	.684	.655	.645	-	-	-	.643	**.805**	.770	-	-	.694	.701	.675	.674	-	-	-
	U-N	.642	**.803**	.764	-	-	.689	.724	.675	.654	-	-	-	**.868**	.835	.789	-	-	.721	.837	.820	.846	-	-	-
	U-R	.641	**.797**	.766	-	-	.699	.753	.733	.687	.733	.761	.759	.689	**.808**	.769	-	-	.708	.771	.749	.720	.758	.777	.765
Retraining Efficacy	M-L	.990	.988	.988	.990	**.991**	.989	.987	.990	.990	.986	.987	.987	.990	.988	.988	.990	.990	.989	.989	.990	.990	.988	.988	.988
	M-N	.984	.987	.988	**.992**	.991	.990	.989	.989	.989	.989	.989	.990	.988	.987	.986	.988	.987	.987	.986	.986	.987	.989	**.989**	.987
	M-R	.994	.989	.991	.995	**.995**	.994	.992	.994	.994	.993	.993	.993	.996	.996	.996	.997	.997	**.998**	.996	.997	.996	.997	.997	.997
	C-L	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937	.937
	C-N	.956	.957	.956	.956	.956	.955	.957	.956	.957	.956	.956	.956	.957	.957	.957	.957	.957	.957	.957	.957	.957	.957	.957	.957
	C-R	.961	.960	.960	**.962**	.961	.959	.961	.960	.961	.961	.960	.960	.965	.966	.966	.967	.967	.969	.969	.969	.969	.970	.970	**.970**
	U-L	.088	.088	.083	-	-	.740	.760	.074	**.068**		-	-	.094	.085	.091	-	-	.083	.084	.075	**.075**	-	-	-
	U-N	.039	.039	.039	-	-	.039	.039	.039	.039	-	-	-	.041	.041	.041	-	-	.041	.041	.041	.041	-	-	-
	U-R	.057	.052	.049	-	-	.049	.048	.047	.046	.046	.046	**.045**	.050	.048	.048	-	-	.048	.046	.042	**.041**	.043	.043	.042

Table 5.2: Three measures of effectiveness of each criterion (column), for each model under test (row). The acronyms in *Model* column stand for M: MNIST, C: CIFAR-10, U: Udacity, and L: LeNet, N: Network-in-Network, R: ResNet.

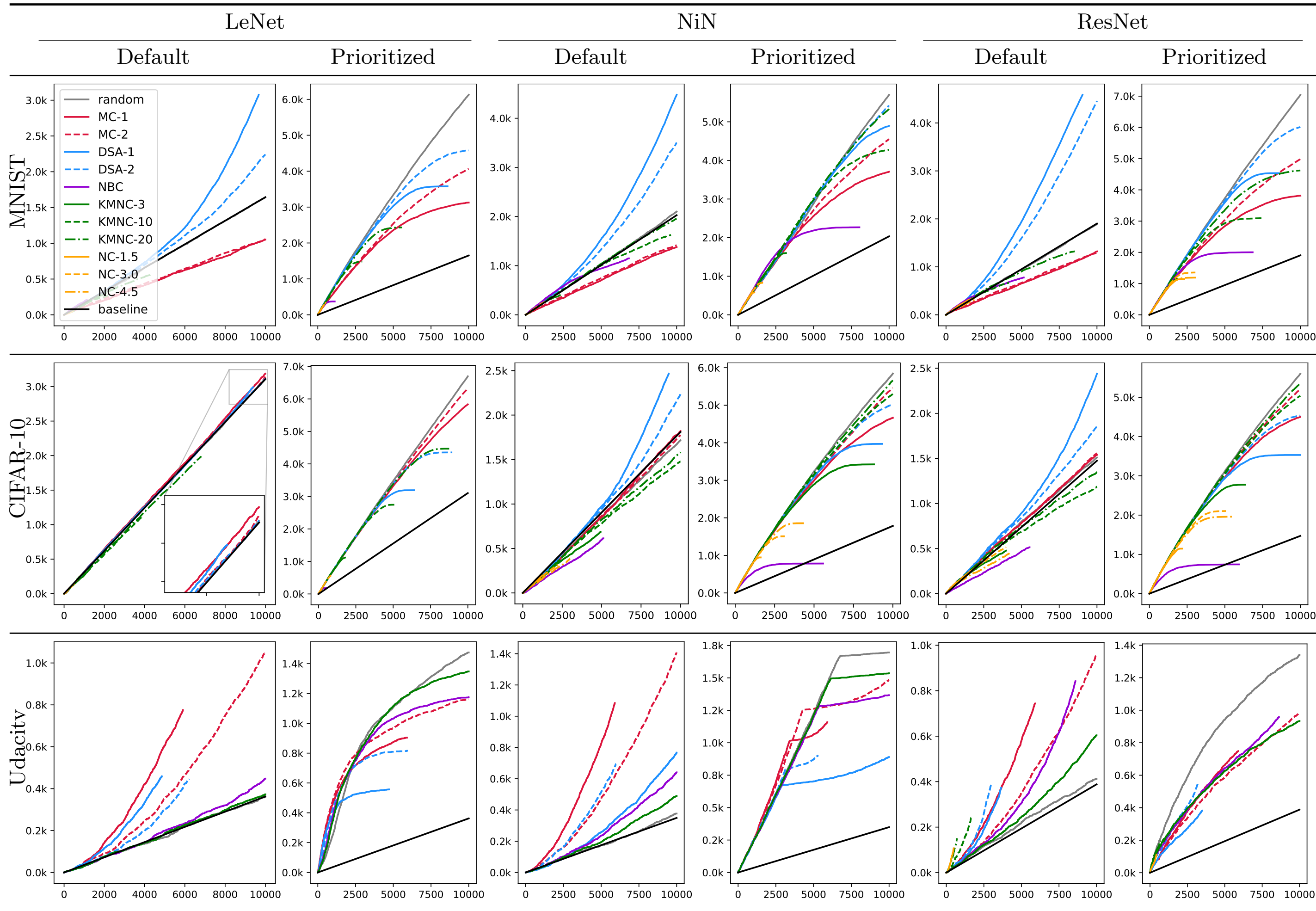


Table 5.3: Failure-revealing effectiveness of test suites constructed by each coverage criterion. The x-axis shows the size of the test suite, and the y-axis shows the number of failure-finding test cases in each test suite.

5.5.1 RQ1: Failure-Revealing Effectiveness

Table 5.3 shows the correlations of test suites with different coverage scores to the failure-revealing effectiveness of the test suites. Irrespective of test prioritization, most of the lines are above the baseline, indicating that coverage is positively correlated with failure-revealing effectiveness. The relative effectiveness of coverage criteria was not stable across different configurations. We will visit them case by case.

In the case of MNIST and CIFAR-10, without prioritization, the effectiveness of criteria were roughly in the order of DSA1, DSA2, and the rest. No white-box criterion purely based on the structure was meaningfully more effective than random baseline. In the case of Udacity, without prioritization, the effectiveness of criteria were roughly in the order of MC2, MC1, NBC, KMNC, and others. Here, the most effective criterion was manifold coverage, with lower granularity MC1 being more efficient when the suite size is small, but plateauing quickly just like with DSA, and DSA2 eventually becomes more effective. One of the reasons why Udacity results look very different from those of MNIST and CIFAR-10 is due to the unique setting of Udacity test dataset, where the training dataset and the master suite were set up differently to simulate the scenario of a model being evaluated for its generalization to a different environment. In these cases, coverage criteria that are designed to incentivize novelty beyond training dataset—such as MC1, MC2, NBC, N4.5—performed well. The precedence among coverage criteria stays relatively consistent across different models, except for $k = 3$ KMNC in *U-R–Default*. Although the clear reason is not known, we conjecture that the batch normalization [150] used uniquely in ResNet is normalizing the distribution of the internal neuron outputs, and that is positively affecting KMNC. For other criteria such as NC, the efficiency was very high in the beginning, as can be seen from the steepness of the slopes, but was not able to differentiate after a certain point. It is likely because NC is too easily achieved with a small number of test cases, that the small number of remaining obligations are satisfied only with corner-cases, which translated to initially high failure-revealing effectiveness.

Most notably, a stark difference is observed between the results with prioritization and the results without. In general, the presence of prioritization was a much stronger contributor to the effectiveness of a test suite than coverage criterion, the effectiveness changing as much as 243% with prioritization (*C-R*). With prioritization, combination

with *any* coverage criterion negatively affected the failure-revealing effectiveness. In summary, **the failure-revealing effectiveness of MCC is comparable to those of white-box criteria**. Also, when it comes to the failure-revealing effectiveness alone, **test prioritization was much more effective than coverage-based test selection** for test suites of the same size.

5.5.2 RQ2: Semantic Balance

A coverage criterion can be deemed to contribute to semantic balance when the balance score achieved by a test suite created with the criterion is higher than when the test suite is constructed randomly. The *semantic balance* for each configuration is presented in the second rows of each model in Table 5.2. The balance score of random test suites are presented in *Rand* column, and the scores of suites that are constructed only with prioritization is presented in *None* column. The scores of random suites for MNIST and CIFAR-10 are close to 1.0 since the master suite contains exactly the same number of test cases per each class. The scores of random suites for Udacity, on the other hand, are much lower than 1.0 at around 0.64, showing that the master suite is skewed when compared to the training dataset. For every case, the balance score decreased with prioritization, as shown in *None* column. The white-box criteria were not particularly helpful in creating a balanced test suite, whether with or without prioritization. Some criteria such as NBC and NC negatively impacted the balance in many cases. MCC, on the other hand, created test suites of higher balance scores than *Random*; especially in Udacity, the scores were around 0.8, where as the random suite scored around 0.64. MCC was also effective in preserving the semantic balance of the test suites under prioritization, even under a shift in distribution. Other white-box criteria, on the other hand, were shown to be less effective. In summary **MCC is more effective in creating semantically balanced test suites than white-box criteria regardless of prioritization**.

5.5.3 RQ3: Retraining Efficacy

The third rows for each model in Table 5.2 present the performance of the model measured as classification accuracy or MAE after retraining with the test suites constructed

by each criterion. The difference compared to baseline may seem trivial for some cases, but even a small difference may translate to a larger difference in actual operational environment, as demonstrated in experiments by Rechet *et al.* [97, 104]. Without prioritization, the retrained models were more effective when tests were selected with coverage criterion. For MNIST and CIFAR-10, DSA showed the highest retraining efficacy whereas MCC showed the lowest. Unlike in failure-revealing effectiveness where prioritization had a huge impact, the retraining efficacy were not consistently better with prioritization. We conjecture that this phenomenon is caused by the prevalence of failures that are caused by the same *faults*, but the exact reason is unknown. The relative efficacy among criteria could not be determined neither. In summary, **coverage-guided test selection, on average, showed marginally higher retraining efficacy compared to random selection, but MCC was worse than white-box criteria on average**. **No criterion was shown to perform consistently better than others**.

5.5.4 Discussion

Although our RQs were designed to compare our proposed approach against others, our experiments revealed that there are more fundamental issues with coverage criteria—their lack of consistent effectiveness. We observed that many of the test suites constructed with most of the white-box criteria except DSA are not consistently more effective than random suites of the same size. We also observed that test prioritization is very effective at constructing an effective test suite when combined with coverage criteria. This does not mean that coverage criteria are useless because prioritization and coverage criteria server different purposes. Prioritization cannot, for instance, tell us when to stop testing, measure the thoroughness of testing, or provide guidance for the semantic diversity of the test suites. Also, test prioritization is applicable only when we start with a pool of test inputs, which may not always be the case in practice. Given these facts, it is best to understand prioritization as a

To discuss more about MCC, what is the implication of the results on its practical utility? MCC showed advantage over white-box criteria on semantic balance. When combined with prioritization, its failure-revealing effectiveness was among the highest. Given these results and the unique benefits of black-box testing, MCC combined with prioritization can be one of the best low-cost choice for achieving high failure-revealing

effectiveness and semantic balance. It is also possible that a combination of these orthogonal approaches—black-box testing, white-box testing, and test prioritization—may help one construct test suites that are effective in every regards.

5.5.5 Threats to Validity

External

While we tried to ensure the variety of subjects, they were limited to specific choices of task and DNN architecture. Our findings may not generalize beyond tested configuration, and especially should not be expected to generalize beyond image classification and regression models. However, we believe that the choice of DNN architecture and the techniques employed within those networks is quite representative for image classification and regression tasks.

Internal

As can be seen from the result that test prioritization changes the efficacy of criteria, the ordering of test case selection matters, and our comparison is not free from the effect of ordering. Also, the upper bound on test suite size was arbitrarily chosen. The efficacy of the coverage criteria was shown to be highly sensitive to many factors such as the type of task, the choice of test dataset, the architecture of the MUT, and configuration of coverage criteria. For the scope of the present work, we did not attempt to reveal the full nuances; a more elaborate study is required to understand the influence of all the factors on the efficacy of the coverage criteria.

5.6 Conclusion

We provided a rigorous definition a new black-box coverage criterion for testing ML systems and systematically evaluated its effectiveness by establishing metrics for assessing coverage criteria on generic properties of interest. Empirical comparison with white-box criteria showed that the new criterion is effective for creating a semantically-balanced test suite with similar fault-revealing ability. The experiments also revealed the weaknesses of ML coverage criteria in general, and the need to further investigate the impact

of various factors that influence their effectiveness, an area for future work.

Chapter 6

Manifold-based Test Case Generation

An effective testing regime must adequately exercise the DNN under test with respect to the input domain, to inspire confidence that the input domain is thoroughly exercised, that any bug in the model is exposed. However, it can be costly to achieve a high adequacy, either qualitatively or quantitatively, since doing so requries a thorough investigation of the model under test across the vast input data space. The most significant cost is incurred by the collection of test input data and assigning label to them. When it comes to testing image classification DNNs, the labeling cost is particularly large because a human needs to provide a label for each sample. If the task of test data collection and labeling can be automated, even if partially, the cost-saving can be significant.

Test case generation attempts to solve this problem by automatically generating test cases—both inputs and their labels—that can be used either to find failures in the DNN under test, or towards building confidence in the DNN. For image classification DNNs it means (1) generating realistic images that are similar to the training data and (2) asigning correct labels to the generated images. The first problem is challenging to automate since images are complex and high-dimensional. Some of the viable options are (1) to apply transformations to existing sets of images, such as applying blur effect, or (2) to synthesize new images using a generative function such as generative adversarial

neural network (GAN). The second problem of generating correct labels is solved for the first approach of applying label-preserving image transformations when the seed images are drawn from a labeled dataset. However, when the seed images are not labeled, or when adopting the second approach of using a generative function, an automated mechanism that classifies images into known classes is required. But, that is exactly what the DNN under test is designed for, implying that such mechanism is not available.

To solve the problem of test case generation for image classifiers, we propose a test case generation method called **manifold-based test case generation** that can simultaneously synthesize test inputs along with their associated labels by learning the generative distribution of both. The data-generating distribution is learnt using a Conditional Variational Autoencoder (CVAE) (Section 4.2), which is a type of manfifold learning technique (Section 4.1) that produces a conditioned generative distribution from which new test inputs can be synthesized for a specified label. On top of this data distribution captured by CVAE, we further propose to apply search heuristics on this generative distribution with the goal of maximizing the probability of the generated test cases triggering failures in the DNN under test. The objective function used in the search process utilizes the classification uncertainty captured by a model-agnostic classifier trained using the features extracted from the manifold.

We evaluate manifold-based test generation in terms of three aspects—(1) how realistic the synthesized inputs are, (2) the failure-revealing effectiveness of the generated test cases, and (3) the accuracy of the synthesized labels. We also compare the effectiveness of the search-based test generation to the baseline of random test generation from the manifold. The results show that the search-based test generation can effectively synthesize a much larger number of failure-revealing test cases that lie on the decision boundaries compared to the baseline method.

6.1 Test Generation from Manifold

Before introducing the manifold-based test generation, we first visit the problem of testing image classifiers in detail and discuss available techniques that address this problem.

6.1.1 Considerations for Testing Image Classifiers

When testing image classifier DNNs, the first requirement for a test case generation technique is the model-independence. That is, given the highly iterative ML workflow (illustrated in Figure 2.4), the generated test cases are the most useful when they can *stay effective* throughout the iterative evolution of the model under test. In addition, we desire the generated test cases to achieve the following goals:

- (Realism): Each generated test input $\hat{x}$ shall be *realistic*, or $P_{\mathcal{D}_{gt}}(\hat{x}) > 0$, that is, the probability of obtaining a generated test input $\hat{x}$ from the ground-truth data distribution should be greater than zero. This can be hard for images that reside in high-dimensional data, unlike, for instance, when the input consists of only a couple of real-valued numbers.

- (Correctness): The label $\hat{y}$ for a synthesized test input $\hat{x}$ should be correctly assigned, or $(\hat{x}, \hat{y}) \sim \mathcal{D}_{gt}$. For image classification, label assignment is costly, as it requires human effort. From a software testing perspective, this is an *oracle problem*, or the problem of determining the correctness for test executions.

- (Effectiveness): The generated test suite $\hat{T}$ shall be *effective* at revealing failures in any model under test, in general. Concretely, given a model f and a validation dataset D_{val},

$$\frac{\sum_{(\hat{x},\hat{y})\in\hat{T}} \mathbb{I}(f(\hat{x}) \neq \hat{y})}{|\hat{T}|} > \frac{\sum_{(x,y)\in D_{val}} \mathbb{I}(f(x) \neq y)}{|D_{val}|} \tag{6.1}$$

  The failure is defined by the testing property (Section 2.3.3); in this proposal, we focus on the *generalization* of the model under test.

- (Completeness): The test generation technique shall, at least, be able to generate any sample $(\hat{x}, \hat{y})$ that has non-zero probability of being sampled from the ground-truth distribution $\mathcal{D}_{gt}$. In other words, for every $(x, y) \sim \mathcal{D}_{gt}$, the probability of generating $(\hat{x}, \hat{y}) \approx (x, y)$ shall be non-zero.

With respect to these goals, there are two broad categories of test generation techniques that can satisfy some of these goals—metamorphic test generation [151] and

black-box adversarial test generation [152]. Metamorphic testing approaches utilize existing dataset, such as a training set, to create other sets of data that are essentially similar to the original one, but hopefully, different enough to reveal failures. DeepTest [26] and DeepRoad [27] belong to this category as they can generate realistic and failure-revealing test cases by relying upon oracle-preserving metamorphic transformations, and by utilizing image translations enabled by image filters or generative adversarial networks. In terms of the four goals that we identified, these techniques satisfy three of them—*realism*, *correctness*, and *effectiveness*—but not *completeness*. The range of transformed test data is bound by the coverage of the applicable metamorphic transformations. Typical metamorphic transformations are not designed to generate less dramatic but arguably more important test cases—normal and realistic cases that look just like any of the training data, but trigger failures.

Another category of test case generation technique is adversarial perturbation [108]. Adversarial generation starts from an existing set of data, and applies input-level transformation, just like in metamorphic testing, with the goal of changing the output from the model under test while keeping the perturbation small enough to be imperceptible to human. However, we consider the adversarial test cases to be unreal, according to our earlier definition of *realism*, unless we assume the presence of an adversary who intentionally perform adversarial attacks. From the persepctive of testing properties (Section 2.3.3), adversarial examples can reveal failures only with respect to the adversarial robustness, and the generalization robustness is not addressed directly with adversarial examples [153].

One limitation that these two techniques have in common is that they both require labeled seed data. What if we want mutations outside of pre-defined transformations? Even further, what if we want to synthesize some novel test data *from scratch*? One can think of a naive random generation, but given the high-dimensional input space, random generation will likely yield nothing better than random noise. What if, hypotethically, there is a space of semantically valid data from which one can pick new test cases? Just as if we collect test data *"in the wild"* we only collect valid data, from this hypothetical space we can sample a variety of valid data that extends beyond training data. It turns out, as explained in Section 4.1, this hypothetical space is called a *manifold*, which is hypothesized to be embedded within the input data space. We might be able to utilize

the manifold captured by a manifold learning technique (Section 4.2) for generating interesting test cases.

Based on this intuition, we propose a novel black-box test case generation approach called *manifold-based test generation*. This approach relies on the idea of using a manifold for black-box testing activities (introduced in Section 4), and utilizes a generative model that can synthesize new samples that lie on the manifold. This approach avoids the oracle problem by conditioning the data-generating manifold by class label, so that a correct label for a sample on the manifold can be specified up-front. This approach generates effective (at failure-revealing) test cases, independent of the model under test, by utilizing an uncertainty measure obtained by the manifold-based classifier. The details of this approach are explained in the following subsections.

6.1.2 The Design of the Test Generator

Generative Model

At the core of our approach lies a technique that can create a manifold from an arbitrary dataset, one that can also generate new data (x, y) from the manifold. A parametric generative model—such as an autoencoder [154], a variational autoencoder (VAE) [133], or a generative adversarial network (GAN) [155]—can serve as a test data generator, but our additional requirement of generating new data *from a manifold* makes the variational autoencoder one of the most suitable choice. At a high level, a VAE is a pair of DNN models that are trained to encode the data into a compact representation, and then to decode the representation back to the original data faithfully, while keeping the distribution of the encoded representations close to a prescribed prior distribution. A compact representation is obtained by forcing the high-dimensional input data into an arbitrary small representation space called a bottleneck. During training, the VAE is trained to find an optimal representation, in an unsupervised manner, that can best summarize the complexity of the data in an economical way. Refer to Section 4.2 for more detail on VAE. We will use the same notation $q_\phi(z|x)$ and $p_\theta(x|z)$ for the encoder and decoder models with ϕ and θ as their respective trainable parameters.

After training, we are left with a VAE encoder that can encode any new data into the encoding space, and a VAE decoder that can decode any representation vector back

to the original input space. We take the posterior distribution of the training data encodings as a manifold structure, but the manifold structure itself cannot be captured mathematically. With a VAE, however, we can leverage its property that the encoder is trained to produce a posterior distribution that closely matches a prescribed prior distribution. As such, we can draw new samples directly from this known prior to obtain a generated dataset that closely resembles the ground-truth distribution $\mathcal{D}_{gt}$. For test case generation, we throw away the encoder and only use the decoder, along with the prior distribution that we specified when training the VAE. New test data x can be generated by randomly sampling from this prior distribution, which is usually defined as multivariate normal distribution.

With a basic VAE decoder as explained above, we can synthesize a new input x by drawing a new sample from the manifold, but the label y needs to be determined manually. Although synthesizing the test input x is useful, the labor required to assign correct label y to each x prevents us from fully automating the test case generation. To overcome this shortcoming, we can use a conditional VAE (CVAE), which can further generate a (x, y) pair by conditioning the decoder with a desired label y. The conditional encoder $q_\phi(z|x, y)$ of CVAE produces a latent encoding that is conditioned by both x and y unlike in the un-conditional encoder $q_\phi(z|x)$. The conditional decoder $p_\theta(x|z, y)$ of CVAE synthesizes x based on both z and y unlike in the basic VAE decoder $q_\phi(x|z)$. This capability of CVAE allows us to solve the oracle problem.

Although it looks more desirable to use CVAE over a basic VAE, we cannot say for sure which approach is better as we might be able to obtain *more interesting* test inputs with a VAE with some added cost of labeling. With a plain VAE, the whole dataset is crammed into one unconditioned manifold, that it might be easier to synthesize samples in the *decision boundary*—for example, a hand-written digit that lies in the decision boundary between label 7 and 9 may look like both 7 and 9, which can be a valuable test case. In this thesis we adopt both VAE and CVAE to empirically compare the pros and cons of the two methods.

Manifold-based Classifier

A naive test generation algorithm can utilize the trained decoder by sampling z from the manifold, or the prior distribution provided to the VAE when training. This approach,

however, would generate a copious number of plain test cases that are not particularly effective at finding failures in the model under test, since, if a VAE is well-trained, the synthesize dataset will have about the same distribution as the original training dataset. A bloated test suite like this is wasteful in many ways, in the time it takes to generate, the space required to store the test cases, and the time it takes to run them all on the models under test. To increase the effectiveness of the generated test cases, we hypothesize that there is a way to determine the *usefulness* of a test case (x, y) that would be generated from a latent code z, in the case of using VAE, or (z, y), in the case of using CVAE. In other words, if we can estimate the model-agnostic failure-revealing effectiveness of (x, y) given z or (z, y), we can apply *search* heuristics to find useful test cases in the manifold space.

To achieve this goal, we train a classification model g_ω called a *manifold-based classifier* that predicts the label y given a manifold representation z (Figure 6.1), or ζ (Figure 6.2), so that we can obtain the classification uncertainty of each run. Their main purpose is to estimate the classification uncertainty of the input $\hat{x}$ that would be synthesized from a latent code z (unconditioned) or (z, y) (conditioned) of our choice. Intuitively, if we pick a z or (z, y) that was already seen during training, the classification uncertainty σ will be low. On the other hand, if we pick a z or (z, y) that was not seen during training, or when it is inherently difficult to assign a correct class due to some confusing characteristics in the input, the classification uncertainty σ will be high. By using σ as a proxy measure of how tricky a synthesizing test case is to classify, we can iterate this loop of trial-and-error for picking the *best*—i.e., the most confusing—test cases without actually synthesizing $\hat{x}$ fully, and without involving the concrete model under test.

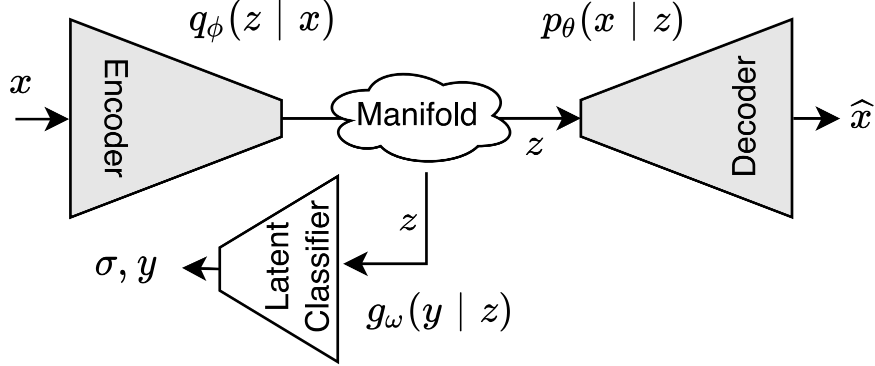


Figure 6.1: Training an unconditioned VAE and a manifold-based classifier: The encoder-decoder pair in VAE is trained together first. Then the manifold-based classifier g_ω is trained to predict the class label y from the latent code z, where $z \in \{z | z = q_\phi(x) \cdot x \in X\}$. g_ω is a Bayesian model that can also estimate classification uncertainty σ.

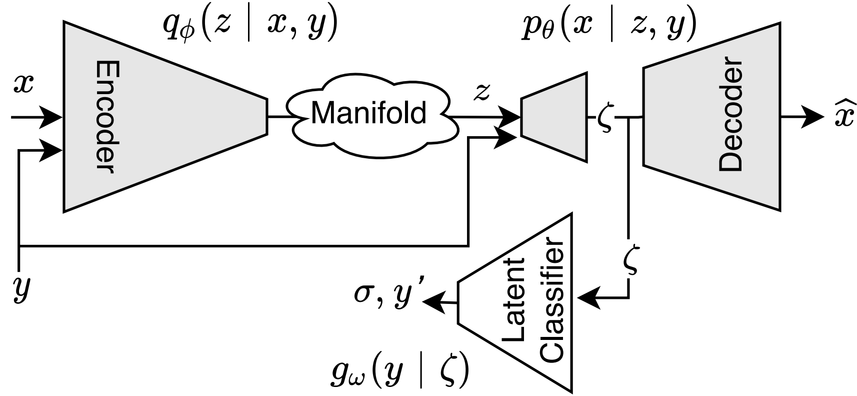


Figure 6.2: Training a Conditional Variational Autoencoder (CVAE) and a Manifold-Based Classifier: The encoder-decoder pair in CVAE is trained together first. Then the manifold-based classifier g_ω is trained to predict the class label y' from the feature vector ζ extracted from the CVAE decoder where $\zeta \in \{\zeta | \zeta = (p_\theta^1 \circ q_\phi)(x, y) \cdot (x, y) \in (X, Y)\}$. g_ω is a Bayesian model that can also estimate classification uncertainty σ.

The structure of the manifold-based classifier together with the VAE is illustrated in Figure 6.1 and Figure 6.2. From here, we only explain the case of CVAE (Figure 6.2) as

the other variant is very simliar and simpler. First, we optimize the CVAE parameters ϕ and θ with the CVAE loss function as in Equation 4.4. Second, we choose a feature layer in the conditional decoder such that we can obtain a label-agnostic representation of the latent code z. We chose the last fully-connected layer in the decoder that interfaces the convolution layers. Third, for the manifold-based classifier g_ω, we construct a Bayesian neural network that is capable of estimating the classification uncertainty (Section 3.1.2). This model is trained using the same training data (X, Y) that is used to train the CVAE, but the input to the manifold-based classifier is the feature layer representation $\zeta = (p^1_\phi \circ q_\phi)(x, y)$ of the training data. During training we freeze the CVAE parameters ϕ and θ so that we only optimize the classifier parameter ω.

Manifold-Based Classifier with a Two-Stage CVAE

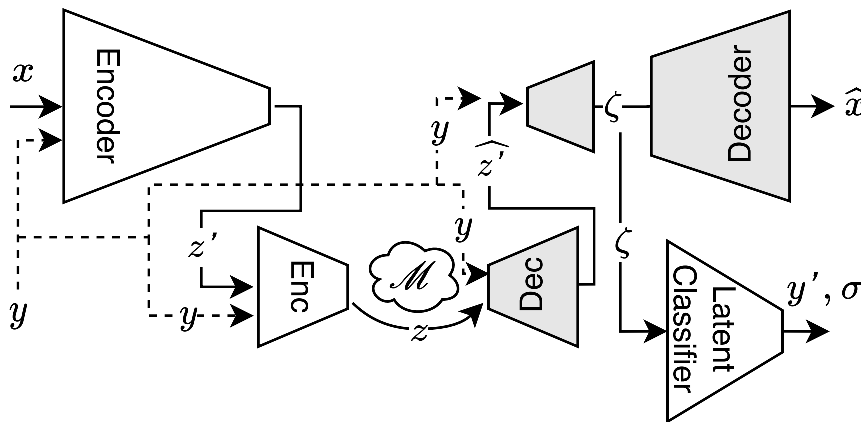


Figure 6.3: Two Stage VAE

Although Figure 6.2 and 6.1 illustrate the VAE as a single pair of encoder and decoder, the actaul variant of VAE/CVAE that we use is a *Two-stage VAE* (Section 4.2), which has another pair of encode-decoder nested within as shown in Figure 6.3. The additional layer of encoder-decoder introduces no theoretical difference as it merely reproduces z', the first-stage encoding, to $\hat{z}'$, the reproduction of z'. It does, however, create another manifold produced by the second-stage encoder. Due to the additional layer of transformation, the manifold produced by the second-stage encoding is more closely aligned with the provided prior distribution. As a result, when we draw samples from

the pior distribution and synthesize new inputs by feeding them to the two decoders sequentially, the distribution of the generated images follow the training data more closely, and, thus, look more realistic, compared to when using the first stage decoder alone [132]. We thus take the second-stage encoding as the manifold space from which to sample new tests. The feature representation ζ is still taken from the first-stage decoder.

The Test Case Generator

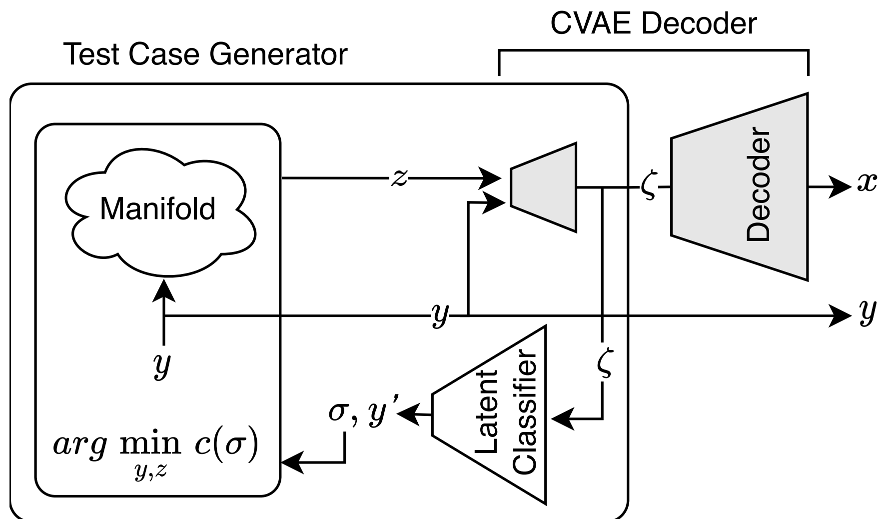


Figure 6.4: The Manifold-based Test Case Generator with CVAE: the optimizer (the inner rectangle) chooses an on-manifold sample (z, y) that minimizes the cost c that is a function of uncertainty σ.

With the VAE/CVAE and the manifold-based classifier trained as explained, we construct our test case generator as illustrated in Figure 6.4. On the right side of the figure is a CVAE decoder that is trained with the same training data used for training the model under test. The decoder is trained in the set-up illustrated in Figure 6.2, and the encoder part is thrown away after training. For simplicity, we abstracted the two-stage decoders as a single decoder in the figure. The smaller inner square on the left side of the figure shows the main test case generator component that searches for desirable test cases. In each iteration, this component attempts to minimize the cost c that is defined

as a function of uncertainty σ by finding a (z, y) that produces a local cost minima. The set of found (z, y) is decoded to a set of corresponding (x, y) pairs, which is a set of test cases.

The detailed procedure of manifold-based test generation is described in Algorithm 1. For each of the N test cases to generate, a label y is first chosen to condition the decoder. Then, an on-manifold latent code z is drawn from a known prior distribution—in this case, the prior was a multivariate normal distribution (line 5). Lines 6 to 13 show the iterative search procedure for finding a z that is likely to maximize the uncertainty σ of the manifold-based classifier $g_\omega(\zeta)$ where $\zeta = p_\theta(z, y)$. z is updated based on a search heuristic of choice (line 7) which will be detailed in the next subsection. The latent code z will be used together with y to predict the classification undertainty σ (lines 8-9). Once z turns out to produce a high uncertainty, or if a class prediction y' that is different from the desired label y, then the new z and σ replaces the current best. After the loop, a $\hat{z}$ with the largest uncertainty $\hat{\sigma}$ is left. Line 14 to 18 check if the new latent code $\hat{z}$ has already been found—we assume that a distinct z leads to a distinct $\hat{x}$ unless the VAE suffers from posterior collapse [156], which is a phenomenon of the VAE decoder learning to ignore one or more latent variables. Finally, the new $\hat{z}$, with its corresponding label $\hat{y}$, is passed to the decoder to synthesize a new input $\hat{x}$ (line 20), and the resulting test case $(\hat{x}, \hat{y})$ is added to the test suite T (line 21). The algorithm terminates when a desired number of N test cases are generated.

Algorithm 1: Manifold-Based Test Case Generation

Input: $x = p_\theta(z, y)$: decoder, $\zeta = p^1_\theta(z, y)$: feature extractor,
$y' = g_\omega(\zeta)$: manifold-based classifier, N: number of tests to generate,
k: distance threshold between latent vectors, n: number of search iterations

Output: T: a set of test cases, where $T_i = (\hat{x}_i, \hat{y}_i)$

1: $Z \leftarrow \phi$; $T \leftarrow \phi$; $is_duplicate \leftarrow False$
2: **while** $|T| \leq N$ **do**
3: $\quad \hat{\sigma} \leftarrow 0.0$
4: $\quad \hat{y} \leftarrow$ choose $\hat{y} \in Y$
5: $\quad z \leftarrow$ draw $z \sim N(0, I^{d_m})$
$\quad$ *Iterative search for a z that maximizes σ*
6: $\quad$ **for all** {1, 2, ..., n} **do**
7: $\quad\quad z \leftarrow update(z)$
8: $\quad\quad \zeta \leftarrow p^1_\theta(z, y)$
9: $\quad\quad y', \sigma \leftarrow g_\omega(\zeta)$
10: $\quad\quad$ **if** $y \neq y' \vee \sigma > \hat{\sigma}$ **then**
11: $\quad\quad\quad \hat{z}, \hat{\sigma} \leftarrow z, \sigma$
12: $\quad\quad$ **end if**
13: $\quad$ **end for**
14: $\quad$ **for all** $\{(\hat{z}_i, \hat{y}_i) | \hat{y}_i = \hat{y} \wedge (\hat{z}_i, \hat{y}_i) \in Z\}$ **do**
15: $\quad\quad$ **if** $|\hat{z}_i - z|_{l_2} < k$ **then**
16: $\quad\quad\quad is_duplicate \leftarrow True$; break
17: $\quad\quad$ **end if**
18: $\quad$ **end for**
19: $\quad$ **if** $!is_duplicate$ **then**
20: $\quad\quad \hat{x} \leftarrow p_\theta(\hat{z}, \hat{y})$
21: $\quad\quad T \leftarrow T \cup \{(\hat{x}, \hat{y})\}$; $Z \leftarrow Z \cup \{(\hat{z}, \hat{y})\}$
22: $\quad\quad is_duplicate \leftarrow False$
23: $\quad$ **end if**
24: **end while**
25: **return** T

6.1.3 Search-based Test Generation on Manifold

In the previous section, we talked about the structure of the test generator, how each component is made, and how the data flows among the components to finally generate test cases. We did not address the most important component, which is how the search procedure works (line 7 in Algorithm 1). We first start from an illustrative example, and then explain the cost function and the search heuristics we used.

Uncertainty Over a Manifold

We illustrate the intuition of manifold-based search with two visualizations.

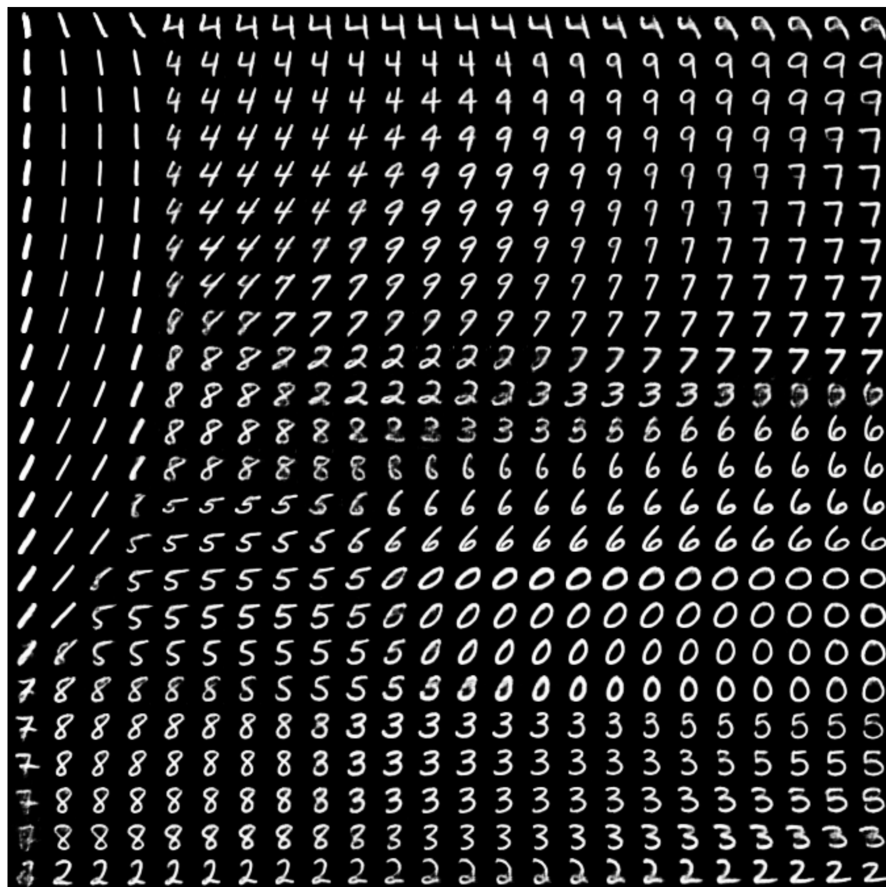


Figure 6.5: Synthesized Inputs Over A 2D Manifold

The first figure (Figure 6.5) shows the test inputs synthesized from a 2D manifold

obtained by a two-dimensional unconditioned VAE. Each dimension of the prior distribution, which is a bivariate normal distribution $\mathcal{N}_2(\mathbf{0}, \mathbf{I}^2)$, is transformed using the cumulative distribution function that maps $(-\infty, \infty)$ to $(0, 1)$, and then mapped to the x and y axis in Figure 6.5. Thus, the digits 7 around the center of the figure, for example, approximately corresponds to the original latent code of $(0, 0)$, and the digit 1 at the top left around $(-3, 3)$. This visualization is obtained by first creating a 2D grid within $(0, 1)^2$, mapping them back to the manifold space $(-\infty, \infty)^2$ with the Gaussian quantile function, running the decoder p_θ on each sample, and then by laying the synthesized inputs together in the original grid. We can observe that the digits that belong to the same class are clustered together, with relatively similar digits of different classes neighboring each other. For example, from the center of the figure to the right, a smooth transition can be observed from digit 7, to digit 9, and then to digit 4. There are noticable boundaries between different classes, but they are not clear-cut. The samples around those class boundaries are valuable as they are realistic examples of inherently confusing inputs, where the classification accuracy can vary among different models under test. They are likely to be very useful for testing if we can capture them automatically.

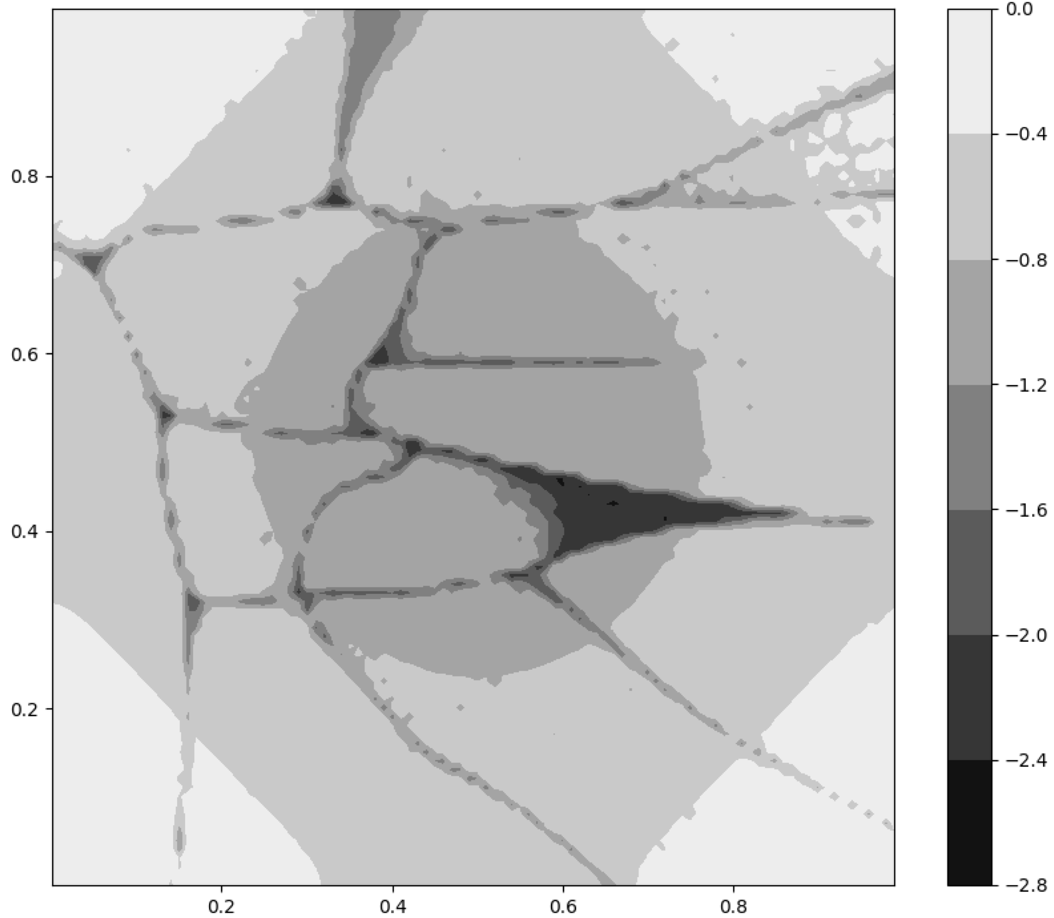


Figure 6.6: Uncertainty of the Manifold-Based Classifier Over a 2D Manifold

The second figure (Figure 6.6) visualizes the countour map over the same 2D manifold space mapped to $[0, 1]^2$ where the value illustrated as shade corresponds to the minus of classification uncertainty measured by the manifold-based classifier (Figure 6.1). It clearly shows that the uncertainty is high around some concentrated regions, which roughly correspond to the decision boundaries that we can observe in Figure 6.5. [1] If we identify these high-uncertainty regions in the manifold and synthesize test cases from there, we would obtain much more *boundary cases* than when blindly sampled from the pior distribution.

[1] any discrepancy between the actual decision boundaries and the one captured by the manifold-based classifier is likely due to the insufficient capacity of the manifold-based classifier.

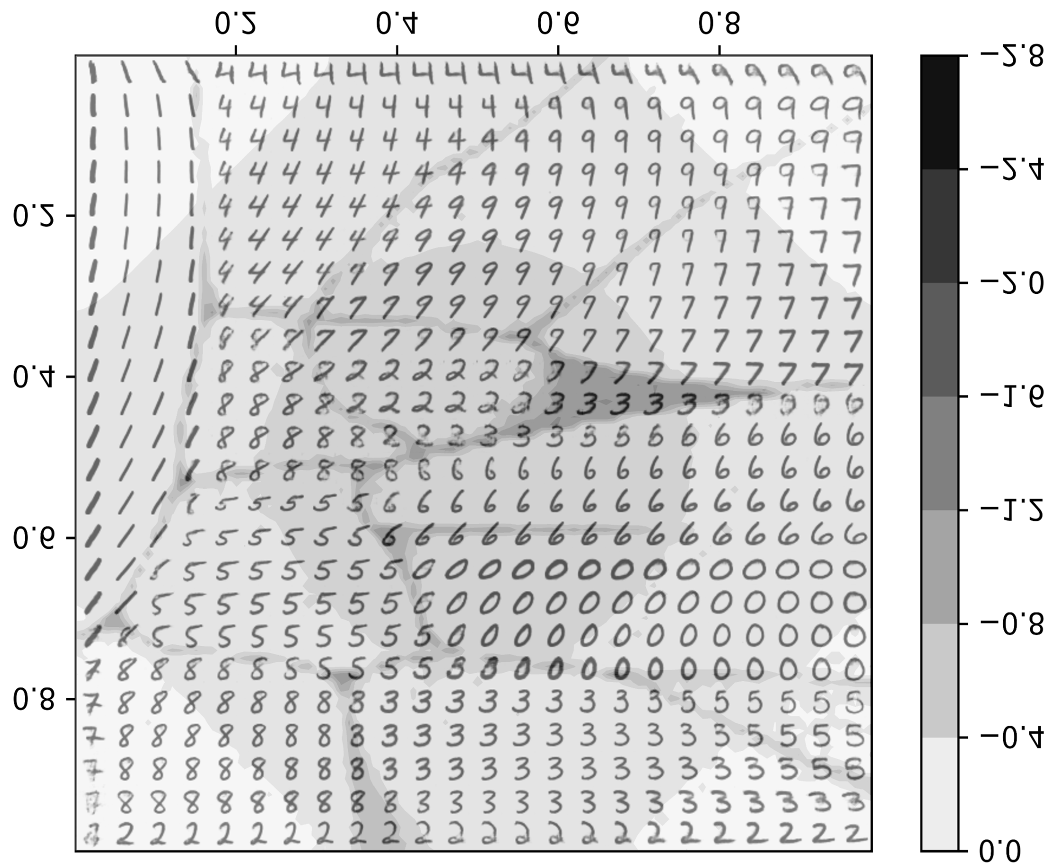


Figure 6.7: Uncertainty (Figure 6.6) and synthesized inputs (Figure 6.5) overlayed on 2D manifold: high-uncertainty regions (shaded in dark) correspond to class boundaries.

While it looks simple enough to identify these high-uncertainty regions in the contour map in Figure 6.6 , real-world dataset—including the MNIST dataset—requires much higher latent dimension to faithfully capture the intricacy of the dataset. We need a methodical search technique that can efficiently navigate the high-dimensional uncertainty landscape over the manifold space.

Search on Manifold

To apply search on manifold space, we first need to design a cost function. One of the simplist and minimal cost function is to just use the uncertainty of the manifold-based classifier as in Figure 6.6 , but there are other practical considerations that arise from

using a VAE 4.2 as a manifold learning technique.

The VAE regularizes the posterior distribution of the samples in the manifold dimension to closely fit a prescribed prior distribution. As such, the samples are concentrated around to the prior distribution, and our optimization shall account for the distribution to avoid sampling from outside the manifold, and also to account for the sample density on the manifold. This objective can be formulated with the probability density function of the prior distribution, such that in-distribution samples in the dense regions are given more weights than those in the sparse regions. As it typically is, we used a unit normal distribution as a prior, which has a probability density function $f(x|\mathcal{N}(0,1)) = (1/\sqrt{2\pi})e^{-0.5x^2}$. With a slight modification to adjust the range of this function to be $[0,1]$, we define the objective term called *sample plausibility* as follows:

$$o_1(z) = \sum_{i=0}^{\kappa} \frac{1}{e^{{z_i}^2}} \tag{6.2}$$

The $1/\sqrt{2\pi}$ term is removed so that the maximum value of $o_2(z)$ is 1 when $\hat{z}$ is 0. Latent codes that deviate further from the mean are penalized so that the z is encouraged to be sampled from the manifold. The other objective term is the uncertainty of the manifold-based classifier, with a slight modification to match the range to be $[0,1]$:

$$o_2(u) = \frac{e^{\sigma(z)} - 1}{e^{\sigma(z)} + 1} \tag{6.3}$$

With the two objective terms o_1 and o_2, by combining them with corresponding weight hyper-parameters w_1 and w_2, we obtain the following cost function for joint-optimization:

$$C(z) = w_1 o_1(z) + w_2 o_2(z) \tag{6.4}$$

The weights are determined empirically.

In choosing an optimization algorithm, we considered that we cannot assume anything about the shape of the optimization landscape. Also, we needed an algorithm that can quickly converge to many distinct local minima rather than a global optimum, since we need a set of diverse test cases rather than a small number of optimal ones. We thus chose to use a stochastic optimization method known as particle swarm optimization (PSO) [157] which evolves a set of candidate solutions into a minimizer through iterations. Intuitively, PSO can be likened to a swarm of herrings or a flock of starlings

that move collectively. Just like a starling in a flock, a *particle* in PSO is “sentient” with its own velocity, yet also closely follows the flock, which is called the *swarm* in PSO. Initially a set of particles are initialized randomly and they sweep around the optimization space to eventually converge to a local optimum. The position of each particle in each iteration is determined by its previous position and its current velocity, where the current velocity is determined by its adherence to its best local minimizer and the current global minimizer. The degree of which a particle maintains its current velocity is called *inertia*, the degree of it *sticking* to its personal best is called *cognitive parameter*, and the degree of which it adheres to the swarm’s global best is called *social parameter*. Three parameters determine the behavior of PSO—i.e., exploration vs. exploitation—and they have to be tuned per task. In our experiment we set a higher value for the cognitive parameter so that all the particles converge to a global best faster and get a chance to exploit around the global best for some iteration. A more detailed explanation can be found in Kennedy’s paper [157].

6.2 Experiment

We evaluate the effectiveness of manifold-based test generation, and compare the effectiveness of the two approaches explained in Section 6.1.3 with one baseline method:

1. **Random** (baseline): Sample new test cases randomly from the manifold. Use CVAE to synthesize both input and the output. New samples are drawn randomly from the prior distribution that the manifold is trained to fit. No search heuristic is applied.

2. **Conditional**: Use CVAE and apply search heuristic. The CVAE synthesizes both input and output. Search heuristic maximizes the uncertainty of the latent feature classifier while choosing samples to synthesize from the manifold.

3. **Unconditional**: Use unconditioned VAE and apply search heuristic. Unconditioned VAE only synthesizes unlabeled test inputs, and manifold-based classifier assigns labels based on the manifold encoding. Search heuristic maximizes the uncertainty of the manifold-based classifier. Since the picked manifold encodings

would typically entail high classification uncertainty, the labels assigned by the manifold-based classifier are likely to be inaccurate.

For each of these generation methods, we answer the following research questions:

- **RQ1**: Can manifold-based test generation generate realistic test inputs?
- **RQ2**: Can manifold-based test generation generate failure-revealing test cases?
- **RQ3**: Can manifold-based test generation assign labels accurately?
- **RQ4**: Is manifold-based search more effective at finding failure-revealing test cases than random sampling?

For each generation method, we assess whether we can synthesize realistic test cases (**RQ1**) that cause failures in the model under test (**RQ2**), with correct labels (**RQ3**). Then we finally compare whether manifold-based search is more effective at synthesizing failure-revealing test cases than random sampling (**RQ4**).

To answer the questions, we first have to clarify the context in which the test generation is conducted. In **Scenario 1**, the goal of test generation is to construct a model-agnostic test suite without involving any concrete model under test. en this scenario, we first generate a test suite of size 1,000 and manually *cleanse* the generated tests by removing invalid test inputs and manually correcting incorrect labelsgenerated tests . We generate a fixed-size test suite with each generation method and evaluate the failure-revealing effectiveness of the model with the manually cleansed test suite. In **Scenario 2**, the goal of test generation is to find failure-revealing test cases efficiently with minimal manual effort. The synthesized test cases are run on the model under test without any manual inspection, and the quality of the test case—whether the input is valid and the label is assigned correctly—is inspected manually only when the output of the model under test does not match the expected output. Note that the test cases that trigger the mismatch between the expected label and the actual output are not necessarily failure-revealing test cases since the generated test cases may contain invalid inputs or have inaccruate labels. A mismatch is a true positive when a test case is both valid and correctly labeled, and a false positive otherwise. It can be more efficient to use the synthesized test suites in this manner when the generated test suites are generally valid and correctly labeled, since a tester only need to inspect a smaller set of test

cases that triggers mismatches. As both scenarios are plausible in testing, we answer **RQ1**–**RQ2** for both of these scenarios.

RQ1 is answered by manually inspecting the validity of the generated inputs. All generated inputs will be inspected for Scenario 1, and only the test cases that trigger mismatching behavior will be inspected for Scenario 2. Invalid, or unrealistic, test cases are those that are judged by a human as out-of-distribution, such as when a hand-written digit does not at all look like a digit. Since this can be highly subjective, we also quantify the *realism* of the synthesized test inputs using Fréchet Inception Distance (FID) which is a widely adopted measure that is shown to correlate well with human judgements [158]. FID measures the Fréchet distance between Gaussians fitted over the feature representations of the two datasets—training data and the generated data—where the feature representation is obtained by running a set of data through the Inception V3 network [159] and extracting 2048-dimensional feature representation in the pool3 bottleneck layer.

RQ2 is answered by measuring the failure-revealing effectivenss of the test suites that are generated by each generation method on four different models under test. The failure-revealing effectiveness is measured in two different testing scenarios. For scenario 1, we manually filter out invalid test cases, reassign labels if incorrect, and measure the accuracy of the models under test with the filtered test suite that only contains valid test cases. This way, we evalaute the quality of the generated test cases when manual effort is involved. For scenario 2, we run the whole test suite of size 1,000 without any manual filtering or changing the label, and measure the accuracy of the models under test with this raw test suite. We then measure the ratio of false-positives in the failure-revealing inputs. A deisable result is when the failure-revealing effectiveness is high and when the false-positive rate is low.

RQ3 is answered by counting the number of accurate vs. inaccurate labels for valid inputs. For **RQ4** we compare the manifold-based search method against the random baseline in terms of realism, labeling accuracy, and failure-revealing effectiveness.

6.2.1 Tasks and the Models Under Test

The first part of the experiments (**RQ1**) are performed with four different image classification tasks. The first task is the popular MNIST hand-written digit classification [116].

The second task is fashion item image classification task, trained with the Fashion-MNIST dataset [160]. Fashion-MNIST is designed to be a drop-in replacement for the MNIST dataset, having the same image resolution and number of classes. However, it is known to be more difficult than MNIST, with the state-of-the-art validation accuracy of 96.7% with data augmentation [161]. The second task is CIFAR10 [136], which is a ten-class image classification task with 50,000 training data. Although the size of the images is small, the images are full-colored and complex, yet packed in a rather small 32 by 32 by 3 resolution. The state-of-the-art accuracy without extra training data is 97.92% [162]. The third task is TaxiNet, which is a dataset of runway images for autonomous taxiing task designed by our industry partner Boeing as a research prototype. Since the original version of TaxiNet is designed to produce continuous values (a cross-track error from the runway center-line and relative heading deviation from the heading of the runway), we modified the design to produce a categorical output only in terms of the cross-track error—namely: far left, left, center, right, far right.

We answer **RQ2** to **RQ4** with only the MNIST image classification task because the VAEs trained for the other tasks were not able to reproduce the original image as faithfully as we desired. For answering **RQ2**, we trained four different MNIST models of different architectures–LeNet [116], Network-in-Network (NiN) [145], MobileNetV2 [118], and ResNet [146]–for measuring the failure-revealing effectiveness. Each model is trained with 50,000 MNIST training data points. Their validation accuracy is presented along with the results.

6.2.2 Training VAE Models

For the implementation of the Conditional Two-stage VAE, we revised the open-sourced TensorFlow code on GitHub [148] implemented by the authors of the Two-Stage VAE paper [132]. We used the model architecture inspired by InfoGAN [149] except for TaxiNet which is based on the ResNet architecture that uses residual connections inside the deep convolution layers [146].

Table 6.1: Trained VAEs

	Trainable parameters	Training time	Latent dimension	FID recon.	FID sample
MNIST	21,860,515	6h 29m	32	17.85	27.71
MNIST (unconditioned)	21,819,555	3h 40m	32	26.07	52.19
Fashion	21,860,515	8h 19m	32	17.97	23.86
CIFAR10	26,074,053	6h 19m	64	78.06	91.49
TaxiNet	41,294,597	13h 42m	32	161.32	157.65

The success of a generative model can be measured by how faithfully it can produce a dataset that is similar to the training dataset, both in terms of the quality and the diversity. Fréchet Inception Distance (FID) is a widely adopted measure that is shown to correlate well with human judgements [158]. It measures the Fréchet distance between Gaussians fitted over the feature representations of the two datasets—training data and the generated data—where the feature representation is obtained by running a set of data through the Inception V3 network [159] and extracting 2048-dimensional feature representation in the pool3 bottleneck layer. We report the two FID scores for each VAE—one for reconstructing the validation dataset and another for generating new dataset. The reconstruction FID score shows how faithfully the model can encode and decode the validation dataset—the set that the VAE was not trained with. Ideally, the reconstructed images shall look exactly the same to the original inputs when the VAE loss (Equation 4.3) reaches its global optimum since reconstruction—$\hat{x} = p_\theta(q_\phi(x, y), y)$—is an identity operation. The sample FID score is measured on a synthesized dataset when the data is sampled from the learnt manifold from a VAE prior distribution that was used to train the VAE. The sample FID score should also be 0 in an ideal case, but this is more difficult than achieving a low reconstruction FID score because of the discrepancy between the prior distribution (such as unit normal distribution) and the actual posterior distribution of the training data encodings. This score shows the performance of the decoder alone.

The details of each VAE configuration and its FID score are shown in Table 6.1. The size of the latent dimension is a tunable hyper-parameter. Ideally, an optimal VAE should be produced, when each dimension is fit to encode a latent feature with no

redundancy. This however, is only hypothetical, and we do not have control over the unsupervised process of learning the manifold. We set the latent dimension size to 64 for all the tasks as suggested by Dai and Wipf [132], and halved it to 32 if the sample FID score remained about the same. Although it was argued in their paper [132] that VAEs are trained to ignore superfluous latent dimensions, we observed that finding a smallest possible latent dimension size that retains the generation quality is a key in obtaining in-distribution inputs with our test generation algorithm.

For training the VAEs and performing the experiments, we used a Ubuntu 16.04 machine on Intel i5 CPU, 32GB DDR3 RAM, SSD, and a single NVIDIA GTX 1080-Ti GPU.

6.3 Result

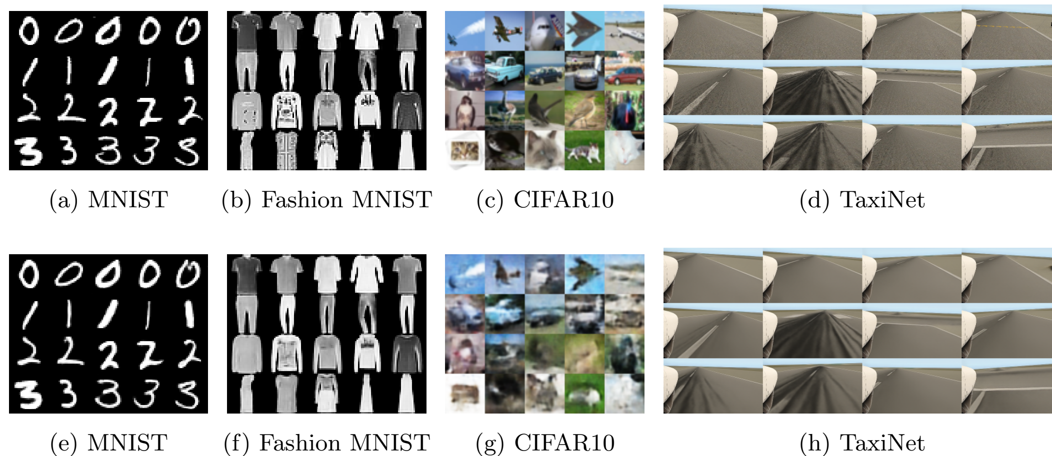
Before answering the four research questions, we first evaluate the quality of the VAEs by examining the images synthesized by them.

6.3.1 Quality of the Images Generated By VAEs

The realism of the synthesized inputs are first quantified by the two FID scores that we measured as we trained the VAEs, as shown in Table 6.1. The quality (realism) of the generated images is first bounded by the reconstruction capability of the trained VAE that is used for the test generation. The realism of the generated images is also bounded by the sample FID, or the FID score measured on the synthesized inputs sampled from the VAE prior distribution, because the search algorithm that we apply on the manifold for sampling failure-revealing inputs will worsen the realism by design. So we can interpret the sample FID score (the last column in Table 6.1) as the lower-bound FID score that we can achieve with our test generation methods.

The FID scores of each VAE range from around 17 to 161. Although it is difficult to understand how those scores translate to the visual quality of the images as humans perceive them, we can notice that the scores are lower for simpler tasks such as MNIST, and higher for more complex tasks such as CIFAR10 or TaxiNet. This makes intuitive sense as it will be more difficult to generate high-quality images for more complex image dataset. We can also observe that an unconditioned VAE produces lower FID

scores compared to the score produced by the conditioned VAE. This is because at a conceptual level, the conditioned VAE creates ten different manifolds, one for each class label, whereas an unconditioned VAE only creates one. For example, the 32 latent variables in the conditioned MNIST VAE only need to encode the characteristics of the digit that belongs to the same class, unlike in the case of the unconditioned VAE where all the training data points have to *share the same space* created by the 32 latent variables regardless of their class label.

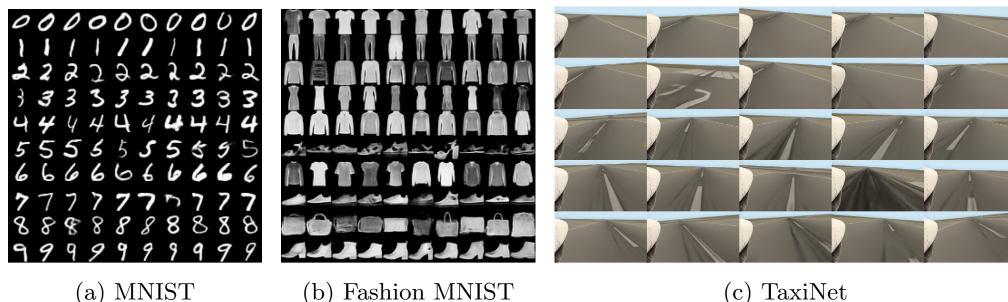


(a) MNIST (b) Fashion MNIST (c) CIFAR10 (d) TaxiNet

(e) MNIST (f) Fashion MNIST (g) CIFAR10 (h) TaxiNet

Figure 6.8: Images reconstructed by trained VAEs—original images are in the upper row, reconstructed images are in the lower row. The reconstructed images look realistic overall, although some fine-grained texture is lost for Fashion MNIST and TaxiNet. CIFAR10 images, however, are too blurry that the objects are not always recongnizable.

For a qualitative assessment of the image quality, we present the original images and the reconstructed images side-by-side in Figure 6.8. The images show that the reconstruction of MNIST and Fashion MNIST is highly faithful, although Fashion MNIST lost small details in reconstruction such as the patterns on the shirts. The reconstruction is not crisp enough for CIFAR10, presumably because the high complexity of the colored images that are packed into a tiny space of 28 by 28 pixels. The image quality of TaxiNet also looks subpar, as implied by the high FID score shown in Table 6.1, with lost details around skid marks on the runway, although the synthesized TaxiNet images preserve the salient features, such as the line marks, better than CIFAR-10.



(a) MNIST (b) Fashion MNIST (c) TaxiNet

Figure 6.9: Images generated by the Two-stage VAEs, sampled randomly from the second-stage manifold space. Images in each row are conditioned by the same class label (no cherry-picking).

To assess the quality of the synthesized images, we present in Figure 6.9 the generated images created by sampling from the VAE prior distribution. Each row of images are generated with the same class conditioning. From the top to the bottom, MNIST classes range from 0 to 9; Fashion MNIST classes are t-shirt/top, trouser, pullover, dress, coat, sandal, shirt, sneaker, bag, ankle boot; TaxiNet classes are far right, right, center, left, far left. We can see that most of the images are very crisp and has indistinguishable quality from the original images. It can be seen that each image matches the label it was conditioned with, and different varieties appear within the same class.

Although these reconstructed and sampled images showed good quality in general, the quality degradation was quite noticeable for all datasets except MNIST. The degradaed images generated through reconstruction 6.8, which is an identity operation in theory, show that the trained VAEs, especially the decoder parts, are not capable enough to generate high-fidelity images for some complex datasets. Given that the image quality achieved through reconstruction is the upper bound of the image quality, manifold-based search will likely only worsen the quality of the synthesized test inputs. While the degraded images may be useful for testing, especially when we assume that such degradation can happen in the real environment, the conclusions that we derive from the test cases synthesized from the subpar VAEs are likely the product of poor VAE rather than the assessment of the manifold-based test generation approach itself. As such, we perform the subsequent evaluation only with the MNIST dataset. This

limitations of the state-of-the-art VAEs are a separate research topic that is outside the scope of this dissertation, and we believe that at the current pace of development, we will soon have VAEs that are capable of generating nearly-perfect images even for more complex datasets.

6.3.2 RQ1: Can manifold-based test generation generate realistic test inputs?

For each generation method, we generated 1,000 test cases and manually counted the number of valid images. To answer **RQ1** for **Scenario 1**, we filter out invalid test inputs first as we construct a test suite independently of the model under test. We consider an input as invalid when the image does not look like any digit, or when it looks too confusing to assign a correct label consistently. The result is presented in Table 6.2. The row named `Ideal` shows that, ideally, there should be 1,000 valid test inputs and 1,000 test cases with correct labels. We cannot set what an ideal FID is because an effective and *interesting* test suite may not, or even should not, follow the training data distribution too closely.

Table 6.2: The number of valid and correctly-labeled test cases for each generation method. The column named *Accurate label* shows the number of correctly labeled test cases and the ratio of them to the number of valid test cases.

Generation Method	Valid	Accurate label	FID
Ideal	1000	1000	0.0
Random	977	976 (99.9%)	46.7
Conditional	994	888 (89.3%)	89.6
Unconditional	832	327 (39.3%)	505.5

According to the realism measured by FID, the generation method that produced the higest number of realistic test inputs was *Random*, followed by *Conditional* and then *Unconditional*. But according to the number of valid inputs, *Conditional* performed the best, generating valid test inputs with a 99.4% ratio. *Random* generated with a ratio of 97.7%, followed by *Unconditional* with a ratio of 83.2%. The low ratio for

Unconditional is mostly due to test inputs that are too confusing, with most common confusions occuring between digit 3 and 5, between 3 and 8, 4 and 9, 5 and 6, and 5 and 8. Although these confusing digits looked pretty real, we had to exclude them to minimize subjectivity in the subsequent evaluation. The high occurrence of these class confusions in the test suite constructed with *Unconditional* method explains why the FID was so high—the generated test suite is highly biased towards confusing inputs that lie on class boundaries. This may be a desirable characteristic for a test suite when its main purpose is to find failures in models under test. The problem of assigning suitable labels remains, however.

In summary, *Random* and *Conditional* generated test suite with high ratio of valid inputs (97.7% and 99.4%, respectively) whereas test suite generated with *Unconditional* method contained a larger number of invalid inputs (with a ratio of 16.8%) that are too confusing to classify even by a human.

6.3.3 RQ2: Can manifold-based test generation generate failure-revealing test cases?

We measure the failure-revealing effectiveness in two different scenarios. In Scenario 1, we generate test suites of size 1,000 for each generation method and manully filter out invalid test cases. We also manually inspect the labels of every test case so that we can use the test suite for regression testing. We then measure the test accuracies of the four model under test with the filtered test suite that only contains valid and correctly labeled test cases. A generated test suite is considered to be effective when the test accuracy achieved with the test suite is lower than the validation accuracy. The result is presented in Table 6.3.

Table 6.3: Test accuracy measured with test suites generated by each generation method in Scenario 1.

Generation Method	Test Suite Size	Test Accuracy			
		LeNet	NiN	MobileNet	ResNet
Validation	10,000	99.41%	99.01%	99.20%	99.41%
Random	977	99.8%	100.0%	100.0%	100.0%
Conditional	994	99.9%	100.0%	100.0%	100.0%
Unconditional	832	80.2%	84.0%	79.0%	84.4%

The first row of Table 6.3 shows the test accuracy achieved with the MNIST validation dataset for each model under test. Every model achieved higher than 99% validation accuracy, meaning that there were fewer than 100 failure-revealing test cases. With the test cases generated with the *Random* method, the test accuracy was always higher than the validation accuracy, which means that it was not particularly effective at finding failures in the model under test. With the *Conditional* method, the test accuracy was about the same as in *Random*. It shows that the *Conditional* search method was not effective at all at generating failure-revealing test cases. With the *Unconditional* VAE, the test accuracy was much lower than the validation accuracy across every models, ranging from 79.0% (175 failure-revealing test cases) to 84.0% (133 test cases) depending on the model under test. A large number of test cases that are sampled from the decision boundaries in the unconditioned manifold contributed to this effectiveness.

In Scenario 2, our goal is not to construct a model-independent regression test suite, but rather to test a specific model under test as efficiently as possible. For each configuration of generation method–model under test, we run the whole test suite first, look at the "failure-revealing" test cases, which we do not yet know whether those are true positive failures or not, and then manually inspect those test cases. A desired result is when a test suite shows a low test accuracy (finding many failures) but also shows a low false-positive rate. The result is presented in Table 6.4.

Table 6.4: The failure-revealing effectiveness measured with 1,000 test cases that are generated per each generation method, measured for Scenario 2. The *False Positive Ratio* column shows the percentage of test cases that are either invalid or incorrectly labeled among the "failure-revealing" test cases, or the test cases that triggered unexpected outputs that do not match the labels. The number in the parenthesis denote the number of false-positive test cases.

Generation Method	Test Accuracy				False Positive Ratio			
	LeNet	NiN	MobileNet	ResNet	LeNet	NiN	MobileNet	ResNet
Validation	99.41%	99.01%	99.20%	99.41%	-	-	-	-
Random	99.3%	99.6%	99.6%	99.6%	71.4% (5)	100.0% (4)	100.0% (4)	100.0% (4)
Conditional	99.8%	99.9%	100.0%	99.9%	50.0% (1)	100.0% (1)	0.0% (0)	100.0% (1)
Unconditional	34.9%	33.4%	33.9%	32.7%	78.6% (512)	78.5% (523)	78.2% (517)	78.9% (531)

Table 6.4 shows the test accuracy of the models under test with a test suite of size 1,000 without any manual filtering or cleansing. The results for *Random* and *Conditional* show very high accuracy above the validation accuracy. When we inspected the failure-revealing test cases, most of them turned out to be false-positives—in this case, caused by test inputs that are invalid. With the *un-conditional* test suite, on the other hand, the models showed very low accuracy between 32% to 35%. When we manually inspected the generated test cases that triggered the unexpected output in the models under test, most of them turned out to be false positives, caused by invalid inputs or incorrect labels, with the ratio of around 78%. The high false-positive ratio tells us that manual inspection of the generated test cases is always required in order to get an accurate assessment of the models under test.

Figure 6.10 showcases some of the failure-revealing test cases after filtering out invalid ones. It can be seen that they are indeed corner-case inputs, yet still looking realistic.

Figure 6.10: Failure-revealing test cases for MNIST (cherry-picked). The lower caption denote the prediction of the model under test, with the predicted probability marked inside the round brackets.

In conclusion, manifold-based test generation was effective at generating failure-revealing test cases with *Conditionl* VAE, but not with *Unconditional* VAE. Test cases generated with *Unconditional* VAE requires manual inspection of both the input and the label.

6.3.4 RQ3: Can manifold-based test generation assign labels accurately?

The accuracy of the labels generated with each generation method is presented in Table 6.2. The third column of the table shows that the label was accurate in 99.9% of the cases for *Random* generation, 89.3% with *Conditional* generation, and 39.3% with

Uncondotional generation. The accuracy was the lowest for *Unconditional*, because the search algorithm only picked the samples that yielded high uncertainty in the manifold-based classifier. Consequently, the labels assigned by the same manifold-based classifier were likely to be inaccurate. In summary, the label was mostly accurate for *Random* generation, but mostly inaccurate for *Unconditional* generation.

6.3.5 Is search better than random sampling?

Throughout **RQ1** through **RQ3**, the test suite generated by *Random* generation showed a high realism, high labeling accuracy, with among the lowest failure-revealing effectiveness. This generation can be useful when the goal is to obtain realistic test cases, but not for obtaining a effective test suite. The *Conditional* manifold-based search was not superior to any other generation method by any measure. The *Unconditional* manifold-based search showed the lowest realism, the lowest labeling accuracy, but the highest failure-revealing effectiveness regardless of the model under test. Across different generation methods, the failure-revealing effectiveness was inversely correlated with the realism of the test inputs and the correctness of the labels. One plausible explanation, at least for the MNIST case example used in our study, is that all of the model under were already trained very well achieving high accuracy, therefore the only way to trigger failures was to find samples on the decision boundaries that were difficult to classify. Samples that lie in the decision boundaries are difficult to label accurately even by the test generator, which explains the low labeling accuracy.

6.3.6 Discussion

Although the experiment demonstrated a promising direction of test case generation with a large number of realistic and failure-revealing test cases generated, the experiments were extremely limited, and pointed to several problems that require further research:

- Better effectiveness of conditional test generation: Unlike what we intended, the conditional test generation did not turn out to be effective at generating failure-revealing test cases, while unconditional generation was much more effective at finding failure-revealing test cases. From this contrasting result, we can conclude

that the uncertainty measured by the manifold-based classifier is a reliable predictor of the effectiveness of the generated test cases, whereas the uncertainty measured by the feature-based classifier is not. Given that we cannot use the manifold representation directly for predicting uncertainty in the case of conditional VAE, since it is already conditioned, the best approach of using conditional VAE for generating failure-revealing test cases might be to use a model-under test in the loop. In this approach, we generate the test cases as in *Random* method used in the experiment, while directly connecting the synthesized output to a model under test, and only take the test cases that trigger the model under test to produce an unexpected ouput that does not match the label. This design makes the whole approach model-dependent, which is why we did not investigate this direction, but it can be very effective at finding model-specific failures as the label and the output of the model under test can be compared easily and rapidly. Further research is needed to evaluate the effectiveness of conditional test generation with the model under test in the loop.

- Better generative model: The experiments shows that the generative capability of VAEs used in the experiments were limited for complicated image classification tasks. A better generative model that can handle higher-dimensional images will help us generate more reaslitic test cases.

- Better manifold-learning: Search-based optimization generated many inputs that are out-of-distribution partly due to a misalignment of the learnt manifold. An ideal manifold shall capture the data-generating distribution accurately, that all the samples that are drawn from the manifold shall be valid and in-distribution. A better manifold-learning technique that can help generate more test cases that are in-distribution.

6.4 Conclusion

We proposed an automated test case generation technique that can generate realistic test inputs with labels, that can cover the entire input domain, and is effective at revealing failures in the model under test. This approach relies on the concept of a manifold,

and using manifold as the domain from which to synthesize new test data that are in-distribution. A manifold, and a generative model that can synthesize new data from the manifold, can be obtained using a manifold-learning technique called VAE. The key idea of this approach is to design an objective function for finding valuable test cases that satisfy our desired goals, and to apply search-based optimization on the manifold. We evaluated this approach by examining the quality of the test cases generated by this approach in terms of the realism of the generated inputs, failure-revealing effectiveness, and the accuracy of generated labels. The results indicated that the manifold-based search with unconditioned VAE can generate test cases that are effective at finding failures in the model under test, but the validity of the test inputs and the assigned labels needed to be inspected manually. The experiment demonstrated the feasibility of the search-based test generation on manifold, but more research is needed for this approach to scale to more complex image classification tasks.

Chapter 7

Conclusion

A wide adoption of learning-enabled systems in the safety-critical domain is contingent upon the degree of confidence in the correctness and reliability of such systems that can be gained through verification and validation activities. A systematic assurance approach that scales to industry grade learning-enabled systems is crucial for verifying such systems and advancing their adoption. This dissertation proposed techniques to partially address this challenge through (1) test input prioritization, (2) a black-box coverage criterion, and (3) black-box test case generation. The coverage criterion and test case generation was aided by an input domain model that can be obtained through an unsupervised learning technique.

For test prioritization, we presented techniques for mitigating the oracle problem in testing DNNs by prioritizing error-revealing inputs based on white-box measures of DNN's sentiment—softmax confidence, Bayesian uncertainty, and input surprise. We evaluated the three techniques on two example systems for image classification and image regression, and multiple versions of the DNNs configured with different architectures. The experiment showed that the sentiment measures can prioritize error-revealing inputs with an average fault-detection rate of 74.9% to 94.8%, indicating that input prioritization based on sentiment measures is a viable approach for effectively identifying the weaknesses of trained models while at the same time reducing the labeling cost.

For the coverage criterion, we proposed a new black-box coverage criterion for testing ML systems and systematically evaluated its effectiveness by establishing metrics for assessing coverage criteria on generic properties of interest. Empirical comparison with

white-box criteria showed that the new criterion is effective for creating a semantically-balanced test suite with similar failure-revealing ability, while being much cheaper to measure compared to white-box coverage criteria. The experiments also revealed the weaknesses of ML coverage criteria in general and the need to further investigate the impact of various factors that influence their effectiveness, an area for future work.

For test case generation, we proposed an automated test case generation technique that can generate realistic test inputs with labels, that can cover the entire input domain, and is effective at revealing failures in the model under test. This approach relies on the concept of a manifold, and using manifold as the domain from which to synthesize new test data that are in-distribution. A manifold, and a generative model that can synthesize new data from the manifold, can be obtained using a manifold-learning technique called VAE. The key idea of this approach is to design an objective function for finding valuable test cases that satisfy our desired goals, and to apply search-based optimization on the manifold. We evaluated this approach by examining the quality of the test cases generated by this approach, in terms of the realism of the generated inputs, failure-revealing effectiveness, and the accuracy of generated labels. The results showed that the manifold-based search with unconditioned VAE can generate test cases that are effective at finding failures in the model under test, but the validity of the test inputs and the assigned labels needed to be inspected manually. The experiment demonstrated the feasibility of the search-based test generation on manifold, but more research is needed for this approach to scale to more complex image classification tasks.

We believe that the research presented in this dissertation is an important step towards developing a systemic and scalable assurance approach that can facilitate the adoption of machine learning in safety-critical domains.

References

[1] Ian Goodfellow, Yoshua Bengio, Aaron Courville, and Yoshua Bengio. *Deep learning*, volume 1. MIT press Cambridge, 2016.

[2] F Brooks and HJ Kugler. *No silver bullet*. April, 1987.

[3] Rasheed Hussain and Sherali Zeadally. Autonomous cars: Research results, issues, and future challenges. *IEEE Communications Surveys Tutorials*, 21(2):1275–1313, 2019.

[4] Markus Borg, Cristofer Englund, Krzysztof Wnuk, Boris Durán, Christoffer Levandowski, Shenjian Gao, Yanwen Tan, Henrik Kaijser, Henrik Lönn, and Jonas Törnqvist. Safely entering the deep: A review of verification and validation for machine learning and a challenge elicitation in the automotive industry. *CoRR*, abs/1812.05389, 2018, 1812.05389.

[5] Rob Ashmore, Radu Calinescu, and Colin Paterson. Assuring the machine learning lifecycle: Desiderata, methods, and challenges. *ACM Comput. Surv.*, 54(5), may 2021.

[6] Changliu Liu, Tomer Arnon, Christopher Lazarus, Clark Barrett, and Mykel J Kochenderfer. Algorithms for verifying deep neural networks. *arXiv preprint arXiv:1903.06758*, 2019.

[7] Leanna Rierson. *Developing safety-critical software: a practical guide for aviation software and DO-178C compliance*. CRC Press, 2017.

[8] G. V. Trunk. A problem of dimensionality: A simple example. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(3):306–307, 1979.

[9] Glenford J Myers, Tom Badgett, Todd M Thomas, and Corey Sandler. *The art of software testing*, volume 2. Wiley Online Library, 2004.

[10] Koushik Sen. Concolic testing. In *Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering*, ASE '07, page 571572, New York, NY, USA, 2007. Association for Computing Machinery.

[11] S. Amershi, A. Begel, C. Bird, R. DeLine, H. Gall, E. Kamar, N. Nagappan, B. Nushi, and T. Zimmermann. Software engineering for machine learning: A case study. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, pages 291–300, 2019.

[12] Jie M. Zhang, Mark Harman, Lei Ma, and Yang Liu. Machine learning testing: Survey, landscapes and horizons. *CoRR*, abs/1906.10742, 2019, 1906.10742.

[13] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. Deepgini: Prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 177188, New York, NY, USA, 2020. Association for Computing Machinery.

[14] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. Input prioritization for testing neural networks. In *2019 IEEE Intl. Conference On Artificial Intelligence Testing (AITest)*, pages 63–70, April 2019.

[15] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pages 397–409, 2021.

[16] Rongjie Yan, Yuhang Chen, Hongyu Gao, and Jun Yan. Test case prioritization with neuron valuation based pattern. *Science of Computer Programming*, 215:102761, 2022.

[17] Muhammad Khatibsyarbini, Mohd Adham Isa, Dayang N.A. Jawawi, and Rooster Tumeng. Test case prioritization approaches in regression testing: A systematic literature review. *Information and Software Technology*, 93:74–93, 2018.

[18] Youcheng Sun, Xiaowei Huang, Daniel Kroening, James Sharp, Matthew Hill, and Rob Ashmore. Structural test coverage criteria for deep neural networks. *ACM Trans. Embed. Comput. Syst.*, 18(5s), oct 2019.

[19] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu, J. Zhao, and Y. Wang. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE Intl. Conf. on Automated Software Engineering*, ASE 2018, pages 120–131, New York, NY, USA, 2018. ACM.

[20] Jinhan Kim, Robert Feldt, and Shin Yoo. Guiding deep learning system testing using surprise adequacy. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, pages 1039–1049. IEEE Press, 2019.

[21] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *SOSP '17*, pages 1–18, New York, New York, USA, 2017. ACM Press, 1705.06640.

[22] Simos Gerasimou, Hasan Ferit Eniser, Alper Sen, and Alper Cakan. Importance-driven deep learning system testing, 2020, 2002.03433.

[23] Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. Concolic testing for deep neural networks. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE 2018, pages 109–119, New York, NY, USA, 2018. ACM.

[24] Zenan Li, Xiaoxing Ma, Chang Xu, and Chun Cao. Structural coverage criteria for neural networks could be misleading. In *ICSE–NIER '19*, pages 89–92, Piscataway, NJ, USA, 2019. IEEE Press.

[25] Fabrice Harel-Canada, Lingxiao Wang, Muhammad Ali Gulzar, Quanquan Gu, and Miryung Kim. Is Neuron Coverage a Meaningful Measure for Testing Deep Neural Networks? 2020.

[26] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th*

International Conference on Software Engineering, ICSE '18, pages 303–314, New York, NY, USA, 2018. ACM.

[27] Mengshi Zhang, Yuqun Zhang, Lingming Zhang, Cong Liu, and Sarfraz Khurshid. 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*, ASE 2018, pages 132–142, New York, NY, USA, 2018. ACM.

[28] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. Testing deep neural networks. *CoRR*, abs/1803.04792, 2018, 1803.04792.

[29] Xiaofei Xie, Lei Ma, Felix Juefei-Xu, Minhui Xue, Hongxu Chen, Yang Liu, Jianjun Zhao, Bo Li, Jianxiong Yin, and Simon See. 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*, ISSTA 2019, page 146157, New York, NY, USA, 2019. Association for Computing Machinery.

[30] Zhi Quan Zhou and Liqun Sun. Metamorphic testing of driverless cars. *Communications of the ACM*, 62(3):61–67, 2019.

[31] Cumhur Erkan Tuncali, Georgios Fainekos, Danil Prokhorov, Hisahiro Ito, and James Kapinski. Requirements-driven test generation for autonomous vehicles with machine learning components. *IEEE Transactions on Intelligent Vehicles*, 5(2):265–280, 2020.

[32] Yasasa Abeysirigoonawardena, Florian Shkurti, and Gregory Dudek. Generating adversarial driving scenarios in high-fidelity simulators. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 8271–8277, 2019.

[33] H. Zhu, D. Liu, I. Bayley, R. Harrison, and F. Cuzzolin. Datamorphic testing: A method for testing intelligent applications. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 149–156, April 2019.

[34] P. McMinn. Search-based software testing: Past, present and future. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 153–163, March 2011.

[35] D.R. Wallace and R.U. Fujii. Software verification and validation: an overview. *IEEE Software*, 6(3):10–17, 1989.

[36] Jerome G Lake. 4 v & v in plain english. In *INCOSE International Symposium*, volume 9, pages 1134–1140. Wiley Online Library, 1999.

[37] Kevin Forsberg and Harold Mooz. The relationship of system engineering to the project cycle. In *INCOSE international symposium*, volume 1, pages 57–65. Wiley Online Library, 1991.

[38] Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[39] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[40] Edmund M Clarke Jr, Orna Grumberg, Daniel Kroening, Doron Peled, and Helmut Veith. *Model checking*. MIT press, 2018.

[41] Ieee standard computer dictionary: A compilation of ieee standard computer glossaries. *IEEE Std 610*, pages 1–217, 1991.

[42] Paul Ammann and Jeff Offutt. *Introduction to software testing*. Cambridge University Press, 2016.

[43] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[44] M. Benjamin, D. Geist, A. Hartman, Y. Wolfsthal, G. Mas, and R. Smeets. A study in coverage-driven test generation. In *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, pages 970–975, 1999.

[45] Rahul Pandita, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Guided test generation for coverage criteria. In *2010 IEEE International Conference on Software Maintenance*, pages 1–10, 2010.

[46] Kshirasagar Naik and Priyadarshi Tripathy. *Software testing and quality assurance: theory and practice*. John Wiley & Sons, 2011.

[47] John Joseph Chilenski and Steven P Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, 1994.

[48] IEEE Standards Association et al. Systems and software engineeringvocabulary iso/iec/ieee 24765: 2010. *Iso/Iec/Ieee*, 24765:1–418, 2010.

[49] Mark Utting and Bruno Legeard. *Practical model-based testing: a tools approach*. Elsevier, 2010.

[50] S.C. Reid. An empirical analysis of equivalence partitioning, boundary value analysis and random testing. In *Proceedings Fourth International Software Metrics Symposium*, pages 64–73, 1997.

[51] Uma D Ferrell. Rtca do-178c/eurocae ed-12c and the technical supplements. In *Digital Avionics Handbook*, pages 207–215. CRC Press, 2017.

[52] Alex Groce, Mohammad Amin Alipour, and Rahul Gopinath. Coverage and its discontents. In *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2014, page 255268, New York, NY, USA, 2014. Association for Computing Machinery.

[53] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *Proceedings of the Eighteenth International Symposium on Software Testing and Analysis*, ISSTA '09, page 5768, New York, NY, USA, 2009. Association for Computing Machinery.

[54] Laura Inozemtseva and Reid Holmes. Coverage is Not Strongly Correlated with Test Suite Effectiveness. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 435–445, New York, NY, USA, 2014. Association for Computing Machinery.

[55] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, pages 302–313, 2013.

[56] Ajitha Rajan, Michael W. Whalen, and Mats P.E. Heimdahl. The effect of program and model structure on mc/dc test adequacy coverage. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE 08, page 161170, New York, NY, USA, 2008. Association for Computing Machinery.

[57] Matt Staats, Gregory Gay, Michael Whalen, and Mats Heimdahl. On the danger of coverage directed test case generation. In Juan de Lara and Andrea Zisman, editors, *Fundamental Approaches to Software Engineering*, pages 409–424, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[58] G. Gay, M. Staats, M. Whalen, and M. P. E. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.

[59] Tom M Mitchell et al. Machine learning. 1997. *Burr Ridge, IL: McGraw Hill*, 45(37):870–877, 1997.

[60] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John P. Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017, 1708.04782.

[61] Gareth James, Daniela Witten, Trevor Hastie, and Robert Tibshirani. *An introduction to statistical learning*, volume 112. Springer, 2013.

[62] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In Yves Lechevallier and Gilbert Saporta, editors, *Proceedings of COMPSTAT'2010*, pages 177–186, Heidelberg, 2010. Physica-Verlag HD.

[63] Nayan B. Ruparelia. Software development lifecycle models. *SIGSOFT Softw. Eng. Notes*, 35(3):813, May 2010.

[64] Yanming Guo, Yu Liu, Ard Oerlemans, Songyang Lao, Song Wu, and Michael S. Lew. Deep learning for visual understanding: A review. *Neurocomputing*, 187:27 – 48, 2016. Recent Developments on Deep Big Vision.

[65] Waseem Rawat and Zenghui Wang. Deep convolutional neural networks for image classification: A comprehensive review. *Neural Computation*, 29(9):2352–2449, 2017. PMID: 28599112.

[66] Geert Litjens, Thijs Kooi, Babak Ehteshami Bejnordi, Arnaud Arindra Adiyoso Setio, Francesco Ciompi, Mohsen Ghafoorian, Jeroen A.W.M. van der Laak, Bram van Ginneken, and Clara I. Snchez. A survey on deep learning in medical image analysis. *Medical Image Analysis*, 42:60 – 88, 2017.

[67] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual Review of Biomedical Engineering*, 19(1):221–248, 2017, https://doi.org/10.1146/annurev-bioeng-071516-044442. PMID: 28301734.

[68] Ekim Yurtsever, Jacob Lambert, Alexander Carballo, and Kazuya Takeda. A Survey of Autonomous Driving: Common Practices and Emerging Technologies. *IEEE Access*, 8:58443–58469, 2020, 1906.05113.

[69] Sorin Grigorescu, Bogdan Trasnea, Tiberiu Cocias, and Gigel Macesanu. A survey of deep learning techniques for autonomous driving. *Journal of Field Robotics*, 37(3):362–386, 2020.

[70] Erfan Asaadi, Ewen Denney, and Ganesh Pai. Quantifying assurance in learning-enabled systems. In António Casimiro, Frank Ortmeier, Friedemann Bitsch, and Pedro Ferreira, editors, *Computer Safety, Reliability, and Security*, pages 270–286, Cham, 2020. Springer International Publishing.

[71] Nancy G Leveson. *Safeware: system safety and computers*. ACM, 1995.

[72] Sanjit A. Seshia and Dorsa Sadigh. Towards verified artificial intelligence. *CoRR*, abs/1606.08514, 2016, 1606.08514.

[73] Simone Scardapane and Dianhui Wang. Randomness in neural networks: An overview. *Wiley Int. Rev. Data Min. and Knowl. Disc.*, 7(2), March 2017.

[74] Léon Bottou. *Stochastic Gradient Descent Tricks*, pages 421–436. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[75] Shiyu Liang, Yixuan Li, and Rayadurgam Srikant. Enhancing the reliability of out-of-distribution image detection in neural networks. *arXiv preprint arXiv:1706.02690*, 2017.

[76] Victoria Hodge and Jim Austin. A survey of outlier detection methodologies. *Artificial intelligence review*, 22(2):85–126, 2004.

[77] Xiaowei Huang, Daniel Kroening, Wenjie Ruan, James Sharp, Youcheng Sun, Emese Thamo, Min Wu, and Xinping Yi. A Survey of Safety and Trustworthiness of Deep Neural Networks: Verification, Testing, Adversarial Attack and Defence, and Interpretability. *arXiv*, pages 0–94, 2020, 1812.08342.

[78] Guy Katz, Clark Barrett, David L. Dill, Kyle Julian, and Mykel J. Kochenderfer. Towards proving the adversarial robustness of deep neural networks. *Electronic Proceedings in Theoretical Computer Science*, 257:1926, Sep 2017.

[79] Shiqi Wang, Kexin Pei, Justin Whitehouse, Junfeng Yang, and Suman Jana. Formal security analysis of neural networks using symbolic intervals. *CoRR*, abs/1804.10829, 2018, 1804.10829.

[80] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE transactions on software engineering*, 37(5):649–678, 2010.

[81] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[82] Armin Biere, Alessandro Cimatti, Edmund M Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. 2003.

[83] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, 2005.

[84] Patrice Godefroid, Michael Y Levin, David A Molnar, et al. Automated whitebox fuzz testing. In *NDSS*, volume 8, pages 151–166, 2008.

[85] Dong Wang, Yanshan Chen, and Zhenyu Chen. DeepPath : Path-driven Testing Criteria for Deep Neural Networks. *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*, pages 119–120, 2019.

[86] L. Ma, F. Juefei-Xu, M. Xue, B. Li, L. Li, Y. Liu, and J. Zhao. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 614–618, Feb 2019.

[87] Xiaoning Du, Xiaofei Xie, Yi Li, Lei Ma, Jianjun Zhao, and Yang Liu. Deepcruiser: Automated guided testing for stateful deep learning systems, 2018, 1812.05339.

[88] J. Sekhon and C. Fleming. Towards improved testing for deep learning. In *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pages 85–88, 2019.

[89] Lei Ma, Fuyuan Zhang, Minhui Xue, Bo Li, Yang Liu, Jianjun Zhao, and Yadong Wang. Combinatorial Testing for Deep Learning Systems. pages 1–14, jun 2018, 1806.07723.

[90] Rob Ashmore and Alec Banks. The utility of neural network test coverage measures. In *SafeAI@AAAI*, 2021.

[91] Yizhen Dong, Peixin Zhang, Jingyi Wang, Shuang Liu, Jun Sun, Jianye Hao, Xinyu Wang, Li Wang, Jin Song Dong, and Dai Ting. There is limited correlation between coverage and robustness for deep neural networks, 2019, 1911.05904.

[92] Shenao Yan, Guanhong Tao, Xuwei Liu, Juan Zhai, Shiqing Ma, Lei Xu, and Xiangyu Zhang. Correlations between deep neural network model coverage criteria and model quality. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2020, page 775787, New York, NY, USA, 2020. Association for Computing Machinery.

[93] Stephanie Abrecht, Maram Akila, Sujan Gannamaneni, Konrad Groh, Christian Heinzemann, Sebastian Houben, and Matthias Woehrle. Revisiting neuron coverage and its application to test generation. 09 2020.

[94] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576580, October 1969.

[95] Ali Shahrokni and Robert Feldt. A systematic review of software robustness. *Information and Software Technology*, 55(1):1–17, 2013.

[96] Dan Hendrycks and Thomas Dietterich. Benchmarking neural network robustness to common corruptions and perturbations. In *7th International Conference on Learning Representations, ICLR 2019*, pages 1–16, 2019, 1903.12261.

[97] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and U C Berkeley. Do CIFAR-10 Classifiers Generalize to CIFAR-10 ? pages 1–25, 2018, arXiv:1806.00451v1.

[98] Moustafa Alzantot, Yash Sharma, Ahmed Elgohary, Bo-Jhang Ho, Mani Srivastava, and Kai-Wei Chang. Generating Natural Adversarial Examples. (2016):1–15, 2018, 1804.07998.

[99] Dan Hendrycks, Kevin Zhao, Steven Basart, Jacob Steinhardt, and Dawn Song. Natural Adversarial Examples. 2019, 1907.07174.

[100] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[101] A. Nguyen, J. Yosinski, and J. Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 427–436, June 2015.

[102] X. Yuan, P. He, Q. Zhu, and X. Li. Adversarial examples: Attacks and defenses for deep learning. *IEEE Transactions on Neural Networks and Learning Systems*, 30(9):2805–2824, 2019.

[103] Alexey Kurakin, Ian Goodfellow, and Samy Bengio. Adversarial machine learning at scale, 2016, 1611.01236.

[104] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do ImageNet Classifiers Generalize to ImageNet? pages 1–76, 2019, 1902.10811.

[105] Taejoon Byun, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. Manifold-based test generation for image classifiers. *2020 IEEE International Conference On Artificial Intelligence Testing (AITest)*, Aug 2020.

[106] Alhussein Fawzi, Hamza Fawzi, and Omar Fawzi. Adversarial vulnerability for any classifier. In *Advances in neural information processing systems*, pages 1178–1187, 2018.

[107] Yarin Gal and Zoubin Ghahramani. Dropout as a bayesian approximation: Representing model uncertainty in deep learning. In *Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48*, ICML'16, pages 1050–1059. JMLR.org, 2016.

[108] Ian J. Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and Harnessing Adversarial Examples. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, dec 2014, 1412.6572.

[109] Akshayvarun Subramanya, Suraj Srinivas, and R. Venkatesh Babu. Confidence estimation in Deep Neural networks via density modelling. 2017, 1707.07013.

[110] Alex Kendall and Yarin Gal. What uncertainties do we need in bayesian deep learning for computer vision? *CoRR*, abs/1703.04977, 2017, 1703.04977.

[111] Michael D Richard and Richard P Lippmann. Neural network classifiers estimate bayesian a posteriori probabilities. *Neural computation*, 3(4):461–483, 1991.

[112] Radford M Neal. *Bayesian learning for neural networks*, volume 118. Springer Science & Business Media, 2012.

[113] Alex Graves. Practical variational inference for neural networks. In *Advances in neural information processing systems*, pages 2348–2356, 2011.

[114] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: A Simple Way to Prevent Neural Networks from Overfitting. *Journal of Machine Learning Research*, 15:1929–1958, 2014, 1102.4807.

[115] Andreas C. Damianou and Neil D. Lawrence. Deep Gaussian Processes. 31, 2012, 1211.0358.

[116] Yann LeCun. The MNIST database of handwritten digits. 1998.

[117] Gregory Cohen, Saeed Afshar, Jonathan Tapson, and André van Schaik. EMNIST: an extension of MNIST to handwritten letters. *CoRR*, abs/1702.05373, 2017, 1702.05373.

[118] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Inverted residuals and linear bottlenecks: Mobile networks for classification, detection and segmentation. *CoRR*, abs/1801.04381, 2018, 1801.04381.

[119] G. Rothermel, R.H. Untch, Chengyun Chu, and M.J. Harrold. Test case prioritization: an empirical study. In *Proceedings IEEE International Conference on Software Maintenance - 1999 (ICSM'99). 'Software Maintenance for Business Change' (Cat. No.99CB36360)*, pages 179–188, 1999.

[120] R Pradeepa and K VimalDevi. Effectiveness of testcase prioritization using apfd metric: Survey. In *International Conference on Research Trends in Computer Technologies (ICRTCT2013). Proceedings published in International Journal of Computer Applications®(IJCA)*, pages 0975–8887, 2013.

[121] Antonio Gulli and Sujit Pal. *Deep Learning with Keras.* Packt Publishing Ltd, 2017.

[122] Masashi Sugiyama and Motoaki Kawanabe. *Machine learning in non-stationary environments: Introduction to covariate shift adaptation.* MIT press, 2012.

[123] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015, 1502.03167.

[124] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA '06, page 2536, New York, NY, USA, 2006. Association for Computing Machinery.

[125] Michael W. Whalen, Ajitha Rajan, Mats P.E. Heimdahl, and Steven P. Miller. Coverage metrics for requirements-based testing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*, ISSTA'06, pages 25–36, New York, NY, USA, 2006. ACM.

[126] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, June 1988.

[127] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2), February 2011.

[128] A. von Mayrhauser, R. Mraz, J. Walls, and P. Ocken. Domain based testing: increasing test case reuse. In *Proceedings 1994 IEEE International Conference on Computer Design: VLSI in Computers and Processors*, pages 484–491, 1994.

[129] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35(8):1798–1828, Aug 2013.

[130] Lawrence Cayton. Algorithms for manifold learning. *Univ. of California at San Diego Tech. Rep*, 12(1-17), 2005.

[131] Yoshua Bengio, Yann LeCun, et al. Scaling learning algorithms towards ai. *Large-scale kernel machines*, 34(5):1–41, 2007.

[132] Bin Dai and David P. Wipf. Diagnosing and enhancing VAE models. *CoRR*, abs/1903.05789, 2019, 1903.05789.

[133] Carl Doersch. Tutorial on variational autoencoders. *ArXiv*, 2016, 1606.05908.

[134] Anders B. L. Larsen, Søren K. Sønderby, and Ole Winther. Autoencoding beyond pixels using a learned similarity metric. *CoRR*, abs/1512.09300, 2015.

[135] Kihyuk Sohn, Honglak Lee, and Xinchen Yan. Learning structured output representation using deep conditional generative models. In *NIPS*, 2015.

[136] Alex Krizhevsky, Geoffrey Hinton, et al. Learning multiple layers of features from tiny images. Technical report, Citeseer, 2009.

[137] Udacity. Become a self-driving car engineer. `https://github.com/udacity/self-driving-car`, 2015.

[138] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks, 2020, 1905.11946.

[139] Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43(2):11:1–11:29, 2011.

[140] D Richard Kuhn, Raghu N Kacker, and Yu Lei. *Introduction to combinatorial testing*. CRC press, 2013.

[141] T. Byun, V. Sharma, S. Rayadurgam, S. McCamant, and M. P. E. Heimdahl. Toward rigorous object-code coverage criteria. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*, pages 328–338, 2017.

[142] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016.

[143] Wei Li. cifar-10-cnn: Play deep learning with cifar datasets. `https://github.com/BIGBALLON/cifar-10-cnn`, 2017.

[144] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[145] Min Lin, Qiang Chen, and Shuicheng Yan. Network in network, 2013, 1312.4400.

[146] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016.

[147] Matthew D. Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In David Fleet, Tomas Pajdla, Bernt Schiele, and Tinne Tuytelaars, editors, *Computer Vision – ECCV 2014*, pages 818–833, Cham, 2014. Springer International Publishing.

[148] Bin Dai and David P. Wipf. Twostagevae, 2019.

[149] Xi Chen, Yan Duan, Rein Houthooft, John Schulman, Ilya Sutskever, and Pieter Abbeel. Infogan: Interpretable representation learning by information maximizing generative adversarial nets. In D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, and R. Garnett, editors, *Advances in Neural Information Processing Systems 29*, pages 2172–2180. Curran Associates, Inc., 2016.

[150] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. In Francis Bach and David Blei, editors, *Proceedings of the 32nd International Conference on Machine Learning*, volume 37 of *Proceedings of Machine Learning Research*, pages 448–456, Lille, France, 07–09 Jul 2015. PMLR.

[151] Junhua Ding, Xiaojun Kang, and Xin-Hua Hu. Validating a deep learning framework by metamorphic testing. In *2017 IEEE/ACM 2nd International Workshop on Metamorphic Testing (MET)*, pages 28–34, 2017.

[152] Andrew Ilyas, Logan Engstrom, Anish Athalye, and Jessy Lin. Black-box adversarial attacks with limited queries and information. In Jennifer Dy and Andreas Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 2137–2146. PMLR, 10–15 Jul 2018.

[153] David Stutz, Matthias Hein, and Bernt Schiele. Disentangling adversarial robustness and generalization. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2019.

[154] Andrew Ng et al. Sparse autoencoder. *CS294A Lecture notes*, 72(2011):1–19, 2011.

[155] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[156] Junxian He, Daniel Spokoyny, Graham Neubig, and Taylor Berg-Kirkpatrick. Lagging inference networks and posterior collapse in variational autoencoders. *CoRR*, abs/1901.05534, 2019, 1901.05534.

[157] James Kennedy. *Particle Swarm Optimization*, pages 760–766. Springer US, Boston, MA, 2010.

[158] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, pages 6626–6637. Curran Associates, Inc., 2017.

[159] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9, 2015.

[160] Han Xiao, Kashif Rasul, and Roland Vollgraf. Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms, 2017.

[161] Connor Shorten and Taghi M Khoshgoftaar. A survey on image data augmentation for deep learning. *Journal of big data*, 6(1):1–48, 2019.

[162] Han Cai, Ligeng Zhu, and Song Han. Proxylessnas: Direct neural architecture search on target task and hardware. *CoRR*, abs/1812.00332, 2018, 1812.00332.