Automated Software Testing of Deep Neural Network Programs

Vahdat Pour, Maryam


master thesis

University of Calgary graduate students retain copyright ownership and moral rights for their thesis. You may use this material in any way that is permitted by the Copyright Act or through licensing that has been assigned to the document. For uses that are not allowable under copyright legislation or licensing, you are required to seek permission.

Downloaded from PRISM: https://prism.ucalgary.ca
Automated Software Testing of Deep Neural Network Programs

by

Maryam Vahdat Pour

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

GRADUATE PROGRAM IN ELECTRICAL ENGINEERING

CALGARY, ALBERTA
SEPTEMBER, 2020

© Maryam Vahdat Pour 2020
Abstract

Machine Learning (ML) models play an essential role in various applications. Specifically, in recent years, Deep neural networks (DNN) are leveraged in a wide range of application domains. Given such growing applications, DNN models’ faults can raise concerns about its trustworthiness and may cause substantial losses. Therefore, detecting erroneous behaviours in any machine learning system, specially DNNs is critical.

Software testing is a widely used mechanism to detect faults. However, since the exact output of most DNN models is not known for a given input data, traditional software testing techniques cannot be directly applied. In the last few years, several papers have proposed testing techniques and adequacy criteria for testing DNNs. This thesis studies three types of DNN testing techniques, using text and image input data.

In the first technique, I use Multi Implementation Testing (MIT) to generate a test oracle for finding faulty DNN models. In the second experiment, I compare the best adequacy metrics from the coverage-based criteria (Surprise Adequacy) and the best example from mutation-based criteria (DeepMutation) in terms of their effectiveness for detecting adversarial examples. Finally, in the last experiment, I applied three different test generation techniques (including a novel technique) to the DNN models and compared their performance if the generated test data are used to re-train the models.

The first experiment results indicate that using MIT as a test oracle can successfully detect the faulty programs. In the second study, the results indicate that although the mutation-based metric can show better performance in some experiments, it is sensitive to its parameters and requires hyper-parameter tuning. Finally, the last experiment shows a 17% improvement in terms of F1-score, when using the proposed approach in this thesis compared to the original models from the literature.
Acknowledgements

I would first like to thank my supervisor, Professor Hadi Hemmati, whose expertise was invaluable in formulating the research questions and methodology. Your insightful feedback pushed me to sharpen my thinking and brought my work to a higher level. I also would like to acknowledge my lab mates for their wonderful collaboration.

I would also like to thank my parents, whom I missed the most during these two years, for their wise counsel and sympathetic ear. You are always there for me. Finally, I could not have completed this dissertation without the support of my husband. He provided arousing discussions as well as happy distractions to rest my mind outside of my research.
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>ii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>iii</td>
</tr>
<tr>
<td>Table of Contents</td>
<td>iv</td>
</tr>
<tr>
<td>List of Tables</td>
<td>v</td>
</tr>
<tr>
<td>List of Figures</td>
<td>vi</td>
</tr>
<tr>
<td><strong>1 INTRODUCTION</strong></td>
<td>1</td>
</tr>
<tr>
<td><strong>2 BACKGROUND</strong></td>
<td>8</td>
</tr>
<tr>
<td>2.1 Deep Neural Networks</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Testing DNN Categories</td>
<td>9</td>
</tr>
<tr>
<td>2.3 Surprise Coverage</td>
<td>10</td>
</tr>
<tr>
<td>2.4 DeepMutation Score</td>
<td>12</td>
</tr>
<tr>
<td>2.5 Multiple Implementation Testing</td>
<td>14</td>
</tr>
<tr>
<td>2.6 Search-based Testing</td>
<td>14</td>
</tr>
<tr>
<td>2.7 Adversarial Samples</td>
<td>15</td>
</tr>
<tr>
<td>2.7.1 Generator Models</td>
<td>15</td>
</tr>
<tr>
<td>2.7.2 Refactoring</td>
<td>17</td>
</tr>
<tr>
<td><strong>3 RELATED WORK</strong></td>
<td>19</td>
</tr>
<tr>
<td><strong>4 METHODOLOGY</strong></td>
<td>24</td>
</tr>
<tr>
<td>4.1 Testing DNN: Test Oracle Generation</td>
<td>24</td>
</tr>
<tr>
<td>4.1.1 Problem Overview</td>
<td>24</td>
</tr>
<tr>
<td>4.1.2 Problem Design</td>
<td>25</td>
</tr>
<tr>
<td>4.1.3 Problem Experiment</td>
<td>29</td>
</tr>
<tr>
<td>4.1.4 Problem Discussion and Results</td>
<td>30</td>
</tr>
<tr>
<td>4.2 Testing DNN: Test Adequacy Evaluation</td>
<td>36</td>
</tr>
<tr>
<td>4.2.1 Problem Overview</td>
<td>36</td>
</tr>
<tr>
<td>4.2.2 Problem Design</td>
<td>37</td>
</tr>
<tr>
<td>4.2.3 Problem Experiment</td>
<td>40</td>
</tr>
<tr>
<td>4.2.4 Problem Discussion and Result</td>
<td>41</td>
</tr>
<tr>
<td>4.3 Testing DNN: Test Input Generation</td>
<td>46</td>
</tr>
<tr>
<td>4.3.1 Problem Overview</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2 Problem Design</td>
<td>46</td>
</tr>
<tr>
<td>4.3.3 Problem Experiment</td>
<td>51</td>
</tr>
<tr>
<td>4.3.4 Problem Discussion and Results</td>
<td>55</td>
</tr>
<tr>
<td><strong>5 Limitations of the study</strong></td>
<td>60</td>
</tr>
<tr>
<td>5.1 Testing DNN: Test Oracle Generation</td>
<td>60</td>
</tr>
<tr>
<td>5.2 Testing DNN: Test Adequacy Evaluation</td>
<td>60</td>
</tr>
<tr>
<td>5.3 Testing DNN: Test Input Generation</td>
<td>61</td>
</tr>
<tr>
<td><strong>6 CONCLUSION</strong></td>
<td>62</td>
</tr>
<tr>
<td>Bibliography</td>
<td>64</td>
</tr>
</tbody>
</table>
List of Tables

1.1 This table summarizes the three different testing techniques used in this thesis, as well as input data and the DNN model. ........................................... 5

2.1 Source level mutation operators (Global) ........................................... 12
2.2 Model level mutation operators ......................................................... 13

4.1 This table shows the calculated FID score for each implementation. These results are for the first input image, shown in Figure 4.5 .............................. 31
4.2 This table shows the calculated FID score for each implementation. These results are for the second input image, shown in 4.4 ...................... 32
4.3 Results of implementations with default parameters and inputs ............. 33
4.4 Sorting implementation’s FID considering lowest FID first, based on our three tables: Table 4.1) Input Image 1 - Set parameters; Table 4.2) Input Image 2 - Set parameters; Table 4.3) Default Input Image - Default Parameters .... 34
4.5 Number of adversarial examples per attack, per dataset ..................... 41
4.6 Pearson correlation between ”Percentage of Adversarial Examples” and ”The Adequacy Score (LCR and LSC) Increments Compared to the Original Test Data”. 44
4.7 Model’s original F1-score reported on the corresponding paper with default configurations on default dataset ................................................. 49
4.8 Code2vec F1-scores on original and re-trained models evaluating on method name’s prediction task ......................................................... 54
4.9 Code2vec F1-scores on original and re-trained models evaluating on method name’s prediction task ......................................................... 55
4.10 Code2seq F1-scores on original and re-trained models evaluating on method name’s prediction task ......................................................... 56
4.11 CodeBERT F1-score on original and re-trained models evaluating on code search task ............................................................................. 57
4.12 Code2seq Rouge-1 F1-score on original and re-trained models evaluating on code captioning task .................................................................. 57
# List of Figures and Illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>This figure shows the overall model structure of an Autoencoder.</td>
<td>26</td>
</tr>
<tr>
<td>4.2</td>
<td>This figure overviews the Variational Auto Encoder model in terms of its functionality.</td>
<td>27</td>
</tr>
<tr>
<td>4.3</td>
<td>First sample of model’s input image.</td>
<td>30</td>
</tr>
<tr>
<td>4.4</td>
<td>Second sample of model’s input image.</td>
<td>30</td>
</tr>
<tr>
<td>4.5</td>
<td>Generated images by all 10 implementations from $I_1$ to $I_{10}$.</td>
<td>31</td>
</tr>
<tr>
<td>4.6</td>
<td>This figure demonstrates the adding noise process for $I_1$ implementation.</td>
<td>34</td>
</tr>
<tr>
<td>4.7</td>
<td>This figure demonstrates the adding noise process for $I_2$ implementation.</td>
<td>34</td>
</tr>
<tr>
<td>4.8</td>
<td>MNIST dataset targeted the 5-layer-convolutional model with five different adversarial attacks.</td>
<td>42</td>
</tr>
<tr>
<td>4.9</td>
<td>MNIST dataset targeted the LeNet model with five different adversarial attacks.</td>
<td>43</td>
</tr>
<tr>
<td>4.10</td>
<td>CIFAR-10 dataset targeted 12-layer-convolutional model with five different adversarial examples.</td>
<td>43</td>
</tr>
<tr>
<td>4.11</td>
<td>CIFAR-10 dataset targeted GoogleNet model with five different adversarial examples.</td>
<td>44</td>
</tr>
<tr>
<td>4.12</td>
<td>The effect of changing mutation operator in LCR.</td>
<td>45</td>
</tr>
<tr>
<td>4.13</td>
<td>The effect of changing layers in LSC.</td>
<td>45</td>
</tr>
<tr>
<td>4.14</td>
<td>The effect of changing mutation rate in LCR.</td>
<td>45</td>
</tr>
<tr>
<td>4.15</td>
<td>This flowchart displays the 1-Time Mutation process flow used in third experiment.</td>
<td>51</td>
</tr>
<tr>
<td>4.16</td>
<td>This flowchart displays the 5-Time Mutation process flow used in third experiment.</td>
<td>52</td>
</tr>
<tr>
<td>4.17</td>
<td>This flowchart displays the Guided Mutation process flow used in third experiment.</td>
<td>53</td>
</tr>
<tr>
<td>4.18</td>
<td>Original Java sample code snippet from Java-Large dataset.</td>
<td>59</td>
</tr>
<tr>
<td>4.19</td>
<td>Generated code adversarial using 1-Time refactoring method. The underlined lines indicate the changes from original code snippet.</td>
<td>59</td>
</tr>
<tr>
<td>4.20</td>
<td>Generated code adversarial using 5-Time refactoring method. The underlined lines indicate the changes from original code snippet.</td>
<td>59</td>
</tr>
<tr>
<td>4.21</td>
<td>Generated code adversarial using Guided Mutation refactoring method. The underlined lines indicate the changes from original code snippet.</td>
<td>59</td>
</tr>
</tbody>
</table>
Chapter 1

INTRODUCTION

Machine Learning (ML) [1] algorithms are now used in a wide range of application domains in software engineering, such as prediction and estimation, generation and synthesis, transformation, and many other applications. Since these ML-based applications are growing every day, faults in these software systems can cause substantial problems. Software testing is a widely used mechanism to detect faults in software systems. Therefore, to prohibit critical losses due to faulty programs, ML-based applications need to be verified by software testing as well.

In general, there are two main challenges in software testing: 1) finding effective test inputs that result in detecting faults, and 2) providing a way to determine whether the system under test, given the input, behaves as expected (a passing test) or not (a failing test), which is called the “Test Oracle problem” [2]. Whether we are doing manual testing or automated testing, these challenges must be addressed, but typically they are harder to deal with when the test generation and evaluation are fully automated, which is the context of this thesis.

Regardless of the application type, the underlying technology, and the implemented algorithms, these challenges always need to be handled. However, ML-based software systems and especially those that leverage Deep Neural Networks, are harder and more challenging. Software systems implemented based on Deep Neural Networks (DNNs) are quite different from traditional software. In traditional software systems, developers implement the requirements logic explicitly (as a sequence of programming blocks; assignments, loops, and conditions, etc.) in a method or function. However, in deep learning systems, the logic is not hard-coded explicitly, but rather learnt from a training dataset using a deep neural network, represented as sets of weights fed into non-linear activation functions [3].

Therefore, in terms of automated testing, the strategies for generating test inputs and oracles
may be different than the traditional automated testing approaches. For instance, most traditional software systems have an explicit expected output, that distinguishes the failing vs passing test cases. Deep learning software systems, however, are known as no-oracle problems [4]. Therefore, detecting erroneous behaviour in DNN-based software is different from catching those in traditional software. For example, existing test adequacy criteria for traditional software, such as statement, branch, and data-flow coverage, completely lose effect in testing DNNs.

According to a recent study on testing ML software systems, Liu et al. [5] have classified the testing and debugging tasks for machine learning systems into six groups: 1) Test oracle generation, 2) Test adequacy evaluation, 3) Test input generation, 4) Test prioritization and reduction, 5) Bug report analysis, and 6) Debug and repair.

In this thesis, I cover the first three testing categories: 1) Test oracle generation, 2) Test adequacy evaluation, and 3) Test input generation. DNN models are tested using the mentioned testing techniques by arranging three experiments.

First, I suggest a method to propose a test oracle for detecting faults in ML software. Second, I empirically evaluate existing test adequacy criteria for DNN testing. Finally, I contribute to generating test inputs for assessing DNN models, and retraining them via new train inputs using a novel method called Guided Mutation (GM). In the following, these three experiments are explained in a nutshell.

In the first experiment, I applied a test oracle technique to verify a Variational Auto Encoder (VAE) [6], a generative model to reduce input dimension, as the target DNN model. Multiple Implementation Testing (MIT) [7] is one of the methods that address the “No Oracle” problem. MIT compares different implementations of the same model, using the same test input. Given each model’s output, the test oracle is considered the most repeated output among all implementations. Any model output different (with an acceptable threshold) from the test oracle is perceived as a faulty program.

In the second experiment, I focused on the test adequacy evaluation. Among all test adequacy
criteria in traditional software testing, “code coverage” and “mutation testing” are two popular
techniques that are also adopted in machine learning and specifically DNN testing. Therefore, the
focus of this experiment is on coverage and mutation-based criteria.

In traditional software testing, code coverage determines the percentage of source code ele-
ments (e.g., statements, branches, conditions) that have been executed by a test suite \[8\]. The
higher coverage is an approximation of a higher probability of covered bugs. Unlike traditional
software, a DNN-based program does not implement the program logic through explicit state-
ments, branches, and conditions. Instead, the logic is being learned through a set of neurons in the
neural networks \[9\]. Thus recent works on DNN testing have introduced multiple new coverage
criteria based on “neuron coverage” to assess how well the input data have covered the DNN model
\[10\].

Pei et al. \[9\] were the first group to introduce Neuron Coverage (NC), as a testing metric for
DNNs. NC is defined as the ratio of the activated neurons for the given test inputs to the total
number of neurons in the DNN model. After proposing NC as a new metric, many researchers
showed interest in examining NC and introduced better coverage metrics. For instance, Kim et. al
\[11\] introduced a new concept for testing ML systems called Likelihood Surprise Coverage (LSC),
which calculates how surprised the model is through a test in comparison with the training input.

Mutation testing is one of the forms of white-box testing used in traditional software testing
and involves modifying a program by small changes \[12\]. Test cases detect and reject each mutated
version (mutant) by causing the mutant’s behaviour to differ from the original one. In mutation
testing, a metric is needed to figure out how good the test suites are in detecting faults. Hence, the
mutation score is described as the ratio of detected mutants against all seeded mutants.

In DNN testing, Ma et al. \[13\] proposed an approach called DeepMutation, which mutates
DNN models at the source-code or model-level, to make minor perturbations on the decision
boundary of a DNN. Based on this paper, a mutation score is defined as the ratio of test cases
that their results are changed on the mutant versus the original program, over the total number of
Test adequacy criteria such as coverage or mutation score are typically the metrics that people use to evaluate their test suite or testing strategy. However, in this study, the goal is to evaluate the adequacy criteria themselves. In traditional software testing, that would be done by assessing the metrics’ ability to detect real faults. In DNN-based testing, injecting faults and detecting them is related to the model specifications and its requirement. Therefore, classical evaluation metrics such as precision and recall might not be a good measure to evaluate the adequacy criteria itself based on the DNN model task.

In DNN-based testing, there is a category of faults that are quite important with respect to the system’s reliability. They are generated by adversarial attacks. Adversarial examples are specialized inputs created to fool a neural network, resulting in the misclassification of a given input. These inputs may not be distinguishable from the human eye, but the network cannot classify it correctly \cite{14}. Thus detecting samples of these attacks remains an essential job toward making DNNs robust, safe, and secure.

Both LSC \cite{11} and DeepMutation \cite{15} have been used to detect adversarial samples in recent literature. Therefore, my second experiment compares these two criteria with respect to their power for detecting adversarial attacks.

Finally, in the third experiment, I applied three different test generation techniques (including a novel technique) to the DNN models, and compare their performance if the generated test data are used to retrain the models.

In experiments 1 and 2, the inputs are images. However, in the third experiment, I used textual data. One of the new applications of DNNs is applying DNNs on source code within the domain of textual data. Recent years have seen massive progress in deep learning for source-code tasks, such as code search and comment generation. Methods for learning distributed representations produce low-dimensional vector representations for objects, referred to as embeddings. Code embedding defines as a vectorized representation of code snippets. By learning code embeddings, the appli-
Table 1.1: This table summarizes the three different testing techniques used in this thesis, as well as input data and the DNN model.

<table>
<thead>
<tr>
<th>Testing Category</th>
<th>DNN Model</th>
<th>Input Data</th>
<th>Input Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Test Oracle Generation</td>
<td>Variational Auto Encoder (VAE)</td>
<td>MNIST dataset</td>
<td>Image</td>
</tr>
<tr>
<td>Test Adequacy Evaluation</td>
<td>LeNet</td>
<td>MNIST dataset</td>
<td>Image</td>
</tr>
<tr>
<td></td>
<td>Conv5</td>
<td>MNIST dataset</td>
<td>Image</td>
</tr>
<tr>
<td></td>
<td>GoogleLeNet</td>
<td>CIFAR10 dataset</td>
<td>Image</td>
</tr>
<tr>
<td></td>
<td>Conv12</td>
<td>CIFAR10 dataset</td>
<td>Image</td>
</tr>
<tr>
<td>Test Case Generation</td>
<td>Code2seq</td>
<td>Java Code Snippet</td>
<td>Text (Code Snippet)</td>
</tr>
<tr>
<td></td>
<td>Code2vec</td>
<td>Java Code Snippet</td>
<td>Text (Code Snippet)</td>
</tr>
<tr>
<td></td>
<td>CodeBERT</td>
<td>Java Code Snippet</td>
<td>Text (Code Snippet)</td>
</tr>
</tbody>
</table>

cation of neural techniques covers a wide-range of programming-languages tasks. In this study, I evaluate the motivating tasks (code search, code captioning and method name’s prediction) experimenting three most known and recent tools called Code2Vec [16], Code2Seq [17] and CodeBert [18] in software engineering.

As discussed, detecting adversarial samples is one of the main topics in DNN testing. Thus to test code embedding techniques, one needs to define adversarial samples for code. Though text adversarial sample generation strategies exist, they are not the best fit for source code. Alipour [19] and Thomas [20] worked on generating adversarial sample for testing source-code based DNNs, in particular. Their key idea is to use source code refactoring as a way to change the code. Their goal was to show that they can fool the original DNN using these refactoring operators. In experiment 3, I introduce a new adversarial generation method for source code and compare it with their approaches in terms of the ability to fool the DNN models. Moreover, I retrain the models using the adversarial examples and show how much improvement can be provided using my approach compared to the alternatives.

My proposed adversarial sample generator is based on a Guided Mutation (GM) algorithm. It iteratively mutates the code using refactoring operators to maximize a fitness function (which is based on the DeepMutation++ mutation score).
This thesis seeks to answer six research questions experimenting with three different aspects of testing DNN models. Table 1.1 sums up all three experiments in terms of the testing aspect, the DNN model and the dataset specifications. Following is this thesis contribution in summary:

- **Experiment 1:** In the first experiment, I apply multi-implementation testing on a DNN model called Variational Auto Encoder, to see if it can be used as a test oracle, and faults can be detected as a result. My investigations are willing to answer the following research question:

  \[RQ1: \text{How can multi-implementation testing adequately approximate a test oracle for DNN models?}\]

- **Experiment 2:** In the second experiment, I empirically evaluated two different test adequacy metrics for testing DNNs called Surprise Coverage (SC) and Mutation Score (MS) to detect adversarial samples. In this experiment, I seek to answer the following research questions:

  \[RQ2: \text{Which of the two metrics (MS or SC) is more sensitive in detecting adversarial examples?}\]

  \[RQ3: \text{How can changing the metrics’ parameters change the metrics’ sensitivity?}\]

- **Experiment 3:** In the last experiment, I test DNN models’ application in source code, with generating adversarial examples. I applied two existing methods, as well as introduce a new method to generate adversarial examples for source code inputs. Later, I retrain the model to see if any improvement occurs given adversarial examples to the training model. In the last experiment, I explore answering the following research questions:

  \[RQ4: \text{How does adversarial test input change the model evaluation score?}\]

  \[RQ5: \text{How does re-training the model impact the F1-score?}\]
**RQ6:** Does the choice of evaluation task have any effect on improving the model’s performance?

The rest of this thesis is organized as follows. Chapter 2 discusses the background for all three experiments in this thesis. Chapter 3 presents related work, Chapter 4 sets out the methodology, and Chapter 5 addresses threats to validity. Finally, Chapter 6 concludes the thesis.
Chapter 2

BACKGROUND

In this chapter, I first explain what are DNN models, and then discuss the different aspects of testing them. Next, I go through the definition of Multi-implementation testing as a testing metric in general, and then introduce Surprise Coverage and Mutation Score as two metrics to evaluate test quality for DNN models. Finally, I define adversarial examples and the different methods to generate them.

2.1 Deep Neural Networks

Artificial intelligence and machine learning, a subset of AI, play an essential part in everyday life in recent years. Deep Neural Networks (DNNs) are also one of the interesting topics that has been involved into different domains such as self driving cars and disease diagnosis. DNN does not only work according to the algorithm but identifies and extracts the relevant high-level features from raw inputs without any human guidance [9].

A deep learning system defines as any software system that includes a Deep Neural Network (DNN) component. This software system may have DNN components interacting with traditional software components. The developers of DNN components can indirectly influence the rules learned by a DNN by modifying the training data, features, and the model’s architectural details [3].

Deep neural networks compose many parametric functions to build representations of an input. Practically, a DNN is made of several successive layers of neurons building up to an output layer. A neuron is an individual computing unit inside a DNN that applies an activation function on its inputs and passes the result to other connected neurons. Input, output, and hidden layers are the
three main layers that each DNN has.

Each connection between the neurons in a DNN is bound to a weight parameter. Training a model defines as learning the weights of the connections by minimizing a cost function. Each layer of the network transforms the information in its input to a higher-level representation of the data.

2.2 Testing DNN Categories

In this section, I briefly describe the following three aspects (out of six) classified by [5] for testing ML systems:

- Test input generation
- Test oracle generation
- Test adequacy evaluation

**Test input generation**

In the area of testing, researchers have developed many techniques and tools to target test input generation [21]. There are a number of these techniques in the literature nowadays, which differ in the way they generate inputs, the strategy they use to explore the behaviour of the model under test, and the specific heuristics they use. Test input generation defines as the process of creating a set of test data for testing the adequacy of software or a model.

**Test oracle generation**

Test oracle generation describes finding a rule so that it enables the judgment of whether a bug exists. A research literature survey found several potential categorizations for test oracles [2]. Metamorphic Relations and Multiple Implementation are the two examples to generate test oracle.

**Test adequacy evaluation**

Test adequacy evaluation aims to estimate an existing test’s ability in terms of detecting faults. Code coverage and mutation testing are the two most popular test adequacy evaluation techniques.
2.3 Surprise Coverage

Pei et al. [9] published the first coverage testing criterion for DNN models by introducing Neuron Coverage (NC), as the ratio of the activated neurons for the given test inputs to the total number of neurons in the DNN. An activated neuron is a neuron such that the sum of its weights and bias makes it to fire, which means that the amount is more than a specific threshold. Therefore, activation values are defined as the threshold that each neuron requires to be activated. Later, Kim et al. [11] introduced a new concept for testing ML systems. They offer Surprise Adequacy (SA), which calculates how surprised the model is through a test input compared with the training input. SA aims to define an adequacy criterion that measures behavioural differences observed in a given set of inputs relative to the training data.

SA proposes two measures to calculate the surprise: Likelihood-based Surprise Adequacy (LSA) and Distance-based Surprise Adequacy (DSA).

LSA calculates the probability of density function using Kernel Density Estimation (KDE) [22]. Since SA wants to measure the surprise of the individual input $x$, LSA is defined as a metric that increases when probability density decreases.

$$\text{LSA}(x) = -\log(f(x))$$  \hspace{1cm} (2.1)

where $f(x)$ is the density function, defined as:

$$\hat{f}(x) = \frac{1}{|A_{N_L}(T)|} \sum_{x_i \in T} K_H(\alpha_{N_L}(x) - \alpha_{N_L}(x_i))$$  \hspace{1cm} (2.2)

$N_L$ is a neuron in a selected layer, $A_{N_L}(x)$ yields a set of activation traces, $H$ is a bandwidth matrix, and $K$ is a Gaussian kernel function. $\alpha_{N_L}(x)$ denotes a vector of activation values.

LSA($x$) calculates 1) the probability of a variable to be in a specific area and 2) how probable it is for a neuron to be activated for training and testing input, to compare their activation difference.

DSA, on the other hand, uses Euclidean Distance of test input to the border of two classes as the measure of surprise. If the distance between test input one and border is more than the test
input two and border, then the first input is more surprised, since it could not decide well, to which
class it belongs. Hence, DSA is defined as follows:

$$DSA(x) = \frac{dist_a}{dist_b}$$  \hspace{1cm} (2.3)

where \(dist_a\) and \(dist_b\) are defined as follow:

$$dist_a = ||\alpha_N(x) - \alpha_N(x_i)||$$  \hspace{1cm} (2.4)

$$dist_b = ||\alpha_N(x_a) - \alpha_N(x_b)||$$  \hspace{1cm} (2.5)

and,

$$x_a = \text{argmin} ||\alpha_N(x) - \alpha_N(x_i)||$$  \hspace{1cm} (2.6)

$$x_b = \text{argmin} ||\alpha_N(x_a) - \alpha_N(x_i)||$$  \hspace{1cm} (2.7)

Kim et al. [11] evaluate their method to see whether it is possible to detect adversarial examples
based on SA values, by training an adversarial example classifier (a logistic regression) on SA
values. They only apply DSA for classification tasks, for which it can be more effective than LSA.

Finally, they show that LSA values correctly capture DNN systems’ behaviour since the DSA-
based classifier successfully detects adversarial examples. They also have shown that by retraining
the model using adversarial examples, the accuracy would increase.

Later, they calculated the input diversity, and they called it: Surprise Coverage (SC). Since
both LSA and DSA are defined in continuous spaces, they use bucketing to discretize the space
of surprise and define Likelihood-based Surprise Coverage (LSC). LSC for a set of inputs \(X\) is
defined as follows:

$$LSC(X) = \frac{\left| \{b_i | \exists x \in X : SA(x) \in (U \frac{i-1}{n}, U \frac{i}{n}] \} \right|}{n}$$  \hspace{1cm} (2.8)
Table 2.1: Source level mutation operators (Global)

<table>
<thead>
<tr>
<th>Fault Type</th>
<th>Target</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Repetition</td>
<td>Data</td>
<td>Duplicates training data</td>
</tr>
<tr>
<td>Label Error</td>
<td>Data</td>
<td>Falsify data results</td>
</tr>
<tr>
<td>Data Missing</td>
<td>Data</td>
<td>Remove selected data</td>
</tr>
<tr>
<td>Data Shuffle</td>
<td>Data</td>
<td>Shuffle selected data</td>
</tr>
<tr>
<td>Noise Perturb.</td>
<td>Data</td>
<td>Add noise to training data</td>
</tr>
<tr>
<td>Layer Removal</td>
<td>DNN model</td>
<td>Remove a layer</td>
</tr>
<tr>
<td>Layer Addition</td>
<td>DNN model</td>
<td>Add a Layer</td>
</tr>
<tr>
<td>Activation Function Removal</td>
<td>DNN model</td>
<td>Remove activation function</td>
</tr>
</tbody>
</table>

For a given upper bound of $U$ and buckets $B = b_1, b_2, ..., b_n$ that divide $(0, U]$ into $n$ SA segments.

In the second experiment, I use Likelihood-based Surprise Coverage (LSC) as the surprise coverage metric. The LSC is considered instead of DSA because as mentioned in the baseline paper, DSA is only used for the classifiers, and not for calculating the coverage.

2.4 DeepMutation Score

Ma et al. [13] have proposed a testing technique called DeepMutation, which adopts mutation testing on DNN systems to measure the quality of test data. They first introduced a source-level mutation testing technique that mutates the training data and the target DNN model. Table 2.1 shows the source-level mutation testing operators for DL systems.

Secondly, they designed a set of model-level mutation operators to inject faults and create mutants without training the model. Table 2.2 demonstrates the model-level operators. Eventually, after having mutant models and data, each test input is evaluated on the set of mutant models.

Suppose a $k$-classification problem and let $C = \{c_1, ..., c_k\}$ be all the $k$ classes of input data. For a test data point $t' \in T'$, $t'$ kills $c_i \in C$ of mutant $m' \in M'$ if the following conditions are satisfied: (1) $t'$ is correctly classified as $c_i$ by the original DL model $M$, and (2) $t'$ is not classified as $c_i$ by $m'$. Mutation score for DL systems is defined as follows, where $KilledClasses(T',m')$ is the set of
### Table 2.2: Model level mutation operators.

<table>
<thead>
<tr>
<th>Mutation Operator</th>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gaussian Fuzzing</td>
<td>Weight</td>
<td>Fuzz eights by Gaussian Distribution</td>
</tr>
<tr>
<td>Weight Shuffling</td>
<td>Neuron</td>
<td>Shuffle selected weights</td>
</tr>
<tr>
<td>Neuron Effect Block</td>
<td>Neuron</td>
<td>Block a neuron effect on following layers</td>
</tr>
<tr>
<td>Neuron Activation Inverse</td>
<td>Neuron</td>
<td>Invert the activation status of a neuron</td>
</tr>
<tr>
<td>Neuron Switch</td>
<td>Neuron</td>
<td>Switch two neurons of a same layer</td>
</tr>
<tr>
<td>Layer Deactivation</td>
<td>Layer</td>
<td>Deactivate the effects of a layer</td>
</tr>
<tr>
<td>Layer Addition</td>
<td>Layer</td>
<td>Add a layer in neuron network</td>
</tr>
<tr>
<td>Activation Function Removal</td>
<td>Layer</td>
<td>Remove activation function</td>
</tr>
</tbody>
</table>

classes of $m'$ killed by test data in $T'$ [13]:

$$
MutationScore(T', M') = \frac{\sum_{m' \in M'} |KilledClasses(T', m')|}{|M'| \times |C'|} \quad (2.9)
$$

Wang et al. [15] propose an approach to detect adversarial samples using model mutation. Their approach is an integration of DeepMutation testing [13] and statistical hypothesis testing [23]. They define the problem as to how efficiently the model can decide whether $f(x)$ is a normal sample or an adversarial sample, given an input sample $x$ to a DNN model $f$.

Their approach is based on “sensitivity”, measured by Label Change Rate (LCR). With mutated DNN models based on the given DNN model, the mutated DNN models are more likely to label an adversarial sample different from the label generated by the original DNN model. Given an input sample $x$ (either regular or adversarial) and a DNN model $f$, they first mutate the model using a set of the model mutation operators used in DeepMutation, to obtain a set of mutated models.

Once obtaining such a set of mutated models $F$, they then obtain the label $f_i(x)$ of the input sample $x$ on every mutated model $f_i \in F$. LCR is defined on a sample $x$ as follows:

$$
LCR(x) = \frac{|\{f_i | f_i \in F, f_i(x) \neq f(x)\}|}{|F|} \quad (2.10)
$$

Intuitively, $LCR(x)$ measures how sensitive an input sample $x$ is on a DNN model’s mutations. They observed that the adversarial examples’ LCR values are significantly higher than those of the
standard samples based on the results. A practical implication of the observation is that given an input sample \( x \), LCR could detect whether \( x \) is likely to be normal or adversarial.

Since Wang et al. in their study [15] integrated the mutation models and operators, and their metric is quite similar to the DeepMutation. However, in our experiment, I used the label change rate (LCR) to test the adequacy criteria for mutation by focusing on adversarial examples.

### 2.5 Multiple Implementation Testing

Multiple-implementation testing [24][25] is a technique for addressing no oracle problems [4]. It is based on the insight that multiple implementations of the same functionality may be available to leverage. Some of these implementations can contain different faults leading to unexpected behaviours for particular inputs. However, the same output across a majority of the executed implementations is likely to be correct. Such a majority can be determined by a predefined percentage threshold denoted as \( \alpha \) for a given input when the percentage of the implementations sharing the same output is greater than \( \alpha \); the result is considered the majority output. It can be used as a proxy for the expected output.

### 2.6 Search-based Testing

Search-based software engineering (SBSE) applies metaheuristic search techniques such as Genetic Algorithms (GA) and Hill Climbing (HC) to software engineering [26]. This involves defining a search space or a set of possible solutions. Usually, the search space is too large, and therefore, a metaheuristic approach is suggested to explore the possible solutions. Then, the fitness function is used as a metric to measure the quality of potential solutions [27][28].

Search-Based Software Testing (SBST) applies optimizing search techniques (GA, for example) to solve software testing problems. SBST is used to prioritize test cases, generate test data, optimize software test oracles, and verify software models.
Many activities in software engineering can be stated as optimization problems. In a genetic algorithm, a set of candidate solutions is evolved toward better solutions to an optimization problem. The evolution starts from a set of randomly generated individuals, called population. The fittest ones are transferred from the current generation to the next one by calculating each individual’s fitness. The rest of the population can be mutated to generate new individuals [29]. The algorithm terminates when a maximum number of generations has been reached.

2.7 Adversarial Samples

DNNs have shown a vulnerable behaviour to strategically modified samples, named adversarial examples. These samples are specialized inputs created to confuse a neural network, resulting in the misclassification of a given input. They are generated with some imperceptible perturbations that can fool the DNNs to give false predictions.

There are several ways to generate adversarial samples for both text and image data inputs. Thus detecting examples of these attacks remains an essential job toward making DNNs robust, safe, and secure.

The Fast Gradient Sign Method (FGSM) [30], Jacobian-based Saliency Map Attack (JSMA) [31], and C&W [32] are some famous methods for generating adversarial samples for both text and image [33].

2.7.1 Generator Models

In the following, I describe the main idea behind each adversary generator model.

**FGSM**

The Fast Gradient Sign Method (FGSM) is designed by adding up the gradient of the cost function concerning the input. Notice that FGSM does not guarantee that the adversarial perturbation is minimal.
JSMA
Jacobian-based Saliency Map Attack (JSMA) is a greedy algorithm that changes one pixel during each iteration. This pixel change increases the probability of having the target label, to attack a model with minimal perturbation. It misleads the target model into classifying it with a particular label by adversarial sample.

C&W
Carlini et al. proposed a group of attacks based on three distance metrics. The key idea is to solve an optimization problem that minimizes the perturbation while maximizing the probability of the target class label.

Deepfool
The idea of DeepFool is to make the original samples cross the decision boundary with minimal perturbations.

Blackbox
The idea is to train a substitute model to mimic the target model’s behaviours with data augmentation. It then applies one of the existing attack algorithms, e.g., FGSM and JSMA, to generate adversarial samples for the substitute model.

All these methods are widely used for models with text and image input [33]. In this work, I also used mentioned adversarial methods to generate adversarial examples for the image input. However, for the third experiment, since all these methods change the input structure, using them to generate adversarial code snippets is not practical. A small unsupervised change might result in a useless code snippet with no specific meaning. Therefore, I define refactoring as a methodology to generate code adversaries.
2.7.2 Refactoring

In software engineering, code refactoring is a way to change a code snippet, without changing its functionality. It is used to improve an existing code by making it more readable, understandable, and clean. Refactoring also helps to add new features more convenient, building large applications easier, and detecting bugs faster.

In this study, to generate adversarial samples for source code, I used refactoring operators. This is a good choice since a refactored code snippet is semantically the same and must produce the same output for most DNNs. However, the syntactical changes often easily result in a different DNN output. Following is a list of ten refactoring operators to generate semantically-equivalent code snippets, in Java.

- **Local Variable Renaming**: Renames the name of a variable using synonym word.
- **Argument Renaming**: Renames the name of an argument using synonym word.
- **Method Name Renaming**: Renames the name of a method using synonym word.
- **API Renaming**: Renames the name of a API using synonym word for the local variable.
- **Local Variable Adding**: Adds a local variable to the code
- **Argument Adding**: Adds an argument to the code
- **Add Print**: Adds print to a random line of the code
- **For Loop Enhance**: Replaces for loops with while loops or vice versa.
- **IF Loop Enhance**: Replace IF condition with equivalent logic
- **Return Optimal**: Change return variable where possible.
Although the source code functionality has not been changed using the mentioned refactoring techniques, if the DNN results change, the refactored code is now considered a new input data input called an adversarial input in this thesis.
Chapter 3

RELATED WORK

A testing approach related to the multiple implementations is Differential testing [34]. In differential testing, developers generate tests to show the differences in behaviour between two versions of a program, if there are any differences. Therefore, with choosing a specific implementation as a reference implementation, a developer is not doing multiple-implementation testing but does differential testing or testing against the reference implementation. In multiple-implementation testing, there is no reference implementation, and all implementations are considered equally.

Another approach for testing Machine Learning applications is metamorphic testing. Murphy and Kaiser [4] proposed based on metamorphic, random testing with considering parameters. Analyzing the problem domain helps them to find similar classes to guide the testing as mentioned above techniques. Metamorphic testing has been applied to specific ML algorithms such as kNN and NB [35] for supervised learning. Metamorphic testing requires high human cost and skill to formally specify metamorphic properties; however, my multiple-implementation testing approach does not require any formal specifications.

Murphy et al., which focused on testing without an oracle in [36], present an approach called Automated Metamorphic System Testing. This paper applied the automation of metamorphic testing by checking the application’s properties after execution at the system level. A person who tests the application can set up metamorphic tests without much manual intervention with minimal impact on the user. They also present an approach called Heuristic Metamorphic Testing, which seeks to reduce false positives and address some cases of non-determinism.

Before Murphy’s paper based on Automated Metamorphic System Testing [36], Gotlieb and Botella [37] coined the term ”automated metamorphic testing” to describe how the process can be conducted automatically. However, their work focuses more on the automatic creation of in-
put data that would reveal violations of metamorphic properties and not automatically check that those properties hold after execution. Also, they do not describe any mechanism for addressing performance concerns or ensuring that the user does not see the program’s additional invocation.

In the Neural Network field, Tian et al. [38] proposed an approach to test Deep-Neural-Networks (DNN) driven autonomous cars. They created a test oracle to leverage metamorphic relations between the car behaviours for different input images to the DNN model. However, defining relations may have different results. Their approach was allowing variations within some error ranges.

Pei et al. [9] proposed an approach for automatically testing DNNs. Their approach goes through multiple deep learning software with similar functionalists. They try to generate test cases from main inputs that can cause incorrect behaviours. On the contrary, my approach uses multiple implementations to find faults using the same input image to find the implementation with deviation.

Multiple-implementation testing also has been used for non-ML application domains, like in detecting faults in XACML implementations [24], web input validates [25], and cross-browser issues [39]. In this thesis, I detect faults in DNN models by using multiple-implementation testing.

Xie et al. [40] approach, derives a test input’s proxy oracle from the majority-voted output and reports those test inputs whose outputs are different from the majority-voted outputs as failing tests. They evaluate their approach to k-Nearest Neighbour (KNN) and Naive Bayes (NB). However, the focus of this thesis is testing DNN models, and not traditional ML models.

Pei et al. [9] published the first coverage criteria in 2017, called DeepXplore. In this paper, they introduced Neuron Coverage as a measure of DNN test case quality. In each iteration, the neuron output value is compared with a user-specified threshold, and if it is larger than the threshold, it is considered an activated neuron. Neuron coverage is defined as the ratio of the activated neurons for the given test inputs and the total number of neurons in the DNN.

Ma et al. [10] published the extended version of the neuron coverage concept in 2018, called
DeepGauge. It proposes a range of finer-grained adequacy criteria described as k-Multisection Neuron Coverage (KMNC), Neuron Boundary Coverage (NBC), and Strong Neuron Activation Coverage (SNAC). K-Multisection Neuron Coverage measures the ratio of activation value buckets that have been covered across all neurons. Neuron Boundary Coverage measures the ratio of neurons that are activated beyond the ranges observed during training. Strong neuron activation coverage also measures how the given test inputs have covered many corner cases. They also presented layer-level coverage criteria, which consider the top hyperactive neurons and their combinations to determine the behaviours of a DNN. In their follow-up work [41] [42], they proposed combinatorial testing coverage. Combinatorial activation status is checked by calculating the fraction of activated neurons in each layer.

Sun et al. [43] proposed four test coverage criteria inspired by the MC/DC [44] also in 2018. MC/DC observes the change of a Boolean variable. However, their approach is to observe any change of neuron in sign, value, or distance to capture the test inputs’ changes.

Sekhon and Fleming [45] defined coverage criteria that look for all pairs of neurons in the same layer and consecutive layers, having all possible value combinations. They also examine existing testing methods for deep neural networks to find opportunities to improve testing methods. Their proposed approach tries to capture all possible parts of the deep neural network’s logic.

Du et al. [46] introduce the first RNN-based testing criteria called DeepCruiser. Their criteria are based on the state and traces of the transition system to capture state transition behaviours. Li et al. [47] published a paper discussing the limitations of coverage criteria for deep networks caused by differences between neural networks and human-written programs. Their experiments with natural inputs found no strong correlation between the number of miss-classified inputs in a test set and its primary coverage.

Huang et. al. [48] published another RNN-based paper called testRNN. This paper evaluates the robustness of a network using three new long short-term memory (LSTM) structural test coverage metrics.
Kim et al. [11] proposed an approach that calculates the input diversity, and they called it as Surprise Coverage (SC). The SC first indicates the inputs’ boundary and can be compared with other coverage criteria mentioned in the paper.

Traditional software testing explains mutation testing as an evaluating metric to measure the quality of a test suite in detecting faults via injecting them into the source code [8] [49]. In testing DNNs, the system’s behaviour depends on data and model structure, as well as a learning network.

Mutation score is a metric to figure out how good the test suits are detecting faults. In traditional testing, the mutation score is described as the ratio of detected faults against all injected faults. Ma et al. [13] proposed a paper called DeepMutation, which mutates DNNs at the source level or model level, to make minor perturbation on the decision boundary of a DNN. Based on this paper, a mutation score is defined as the ratio of test cases that their results are changed against the total number of test cases.

Shen et al. [50] proposed five mutation operators for DNNs and evaluated properties of mutation on the MINST dataset. They pointed out that domain-specific mutation operators are needed to enhance mutation analysis.

Wang et al. [15] propose an alternative approach to detect adversarial samples at run time. They propose a measure of ”sensitivity” and empirically show statistical hypothesis testing and model mutation testing that standard samples and adversarial samples have distinguishable sensitivity. They show that their approach detects adversarial samples generated by state-of-the-art attacking methods efficiently and accurately.

My second experiment fits in the third class of the above. It serves as an empirical study to compare the effectiveness of the best techniques from those two categories in detecting adversarial examples.

Recent years have seen massive progress in deep learning for source-code tasks. Allamanis et al. [51] have done a survey discussed the contrast programming languages against natural languages and consider how these similarities and differences drive the design of probabilistic models.
Generating code adversarial is also a novel area that has not been touched a lot.

Alipour [19] and Thomas [20] worked on generating adversarial sample for testing source-code based DNNs, specifically [16] and Code2Seq [17].

Alipour et al. [19] apply semantics preserving program transformations to produce new programs using refactoring methods on which they expect models to keep their original predictions and report the prediction change rate. However, they have not retrained the model using adversarial test input to see if any improvement happens. Additionally, I have introduced a new method for generating adversarial methods and report the F1 score of the evaluation part.

On the other hand, Thomas et al. [20] focused on the number of changes applied to each test input. As stated, they have tried different amounts of K, which is defined as changing each data input, and reported that the K = 5 is the best based on their experiment on the code2seq model. In our study, I include K = 5 for creating adversarial samples and introduce a guided version of applying K modifications on a test input.

My last experiment proposes a new generating adversarial sample technique, which first generates adversarial examples using three different methods and then retrains the model using those generated samples.
Chapter 4

METHODOLOGY

In this chapter, I describe the methodologies used for testing deep neural networks through three experiments: 1) Testing DNN: Test Oracle Generation, 2) Testing DNN: Test Adequacy Evaluation, and 3) Testing DNN: Test Input Generation. In each section, I go through the problem overview, design, experiment, results and discussion.

4.1 Testing DNN: Test Oracle Generation

In this experiment, I first describe the problem, then design it and, at last, discuss the results.

4.1.1 Problem Overview

In this experiment, I investigate if multi-implementation testing can be considered as one of the oracles to test DNN models.

Objective

In this experiment, I run ten different implementations of the Variational Auto Encoders (VAE) algorithms with a same test inputs and configurations. Considering all model outputs, I define the majority-voted output of these implementations as the test oracle. Finally, I evaluate whether the test oracle can be useful in detecting faulty implementations. Faults in these implementations define as any reason that causes the model to generate low-quality outputs.

In particular, this approach includes three main steps: First, I collect multiple implementations of VAE algorithm from Github. Second, I run test inputs on all implementations to define a test oracle based on the majority-voted output. Finally, I evaluate if the test oracle is a good measure to find faulty implementations.
Research Question

At the end of this experiment, I am willing to answer the following research question:

• **RQ1**: How can multi-implementation testing adequately approximate a test oracle for DNN models?

4.1.2 Problem Design

In this section, I discuss DNN model under study, configurations and datasets used in the experiment.

**Dataset**

I used MNIST [52], a free dataset with more than 30,000 images that includes handwritten English numbers from existing online repositories.

For the test input, I created two images as a test input, a collection of different handwritten numbers for each number from 0 to 9. These test inputs help us to see how good the different implementations can generate the image. I use these test inputs for the model to measure the quality of test inputs. Figure 4.3 and 4.4 are two test input samples used to answer RQ1 in this experiment.

**DNN Model Under Study**

In this experiment, I used VAE algorithm as a representative of the DNN model for testing. I collected multiple implementations of the VAE algorithm by searching VAE and Variational Auto Encoder keywords in GitHub repository. After collecting 25 implementations by October 25\(^{th}\) 2018\(^1\) only 10 implementations were replicable. All implementations were written in the Python language.

Before going through the definition of the VAE itself, I go through the autoencoder definition. As described in [53] and [54], autoencoders are an unsupervised learning technique that leverages neural networks for the task of representation learning. The design of neural network architecture

\(^1\)https://github.com/MaryamVP/VAE-Algorithms
is such that a bottleneck in the network is imposed to compress the original input’s knowledge representation. If the input features were independent of one another, this compression and subsequent reconstruction would be challenging. However, if data has some structures like correlations between input features, this structure can be learned and leveraged when forcing the input through its bottleneck.

As visualized in Figure 4.1, an unlabeled dataset can be framed as a supervised learning problem tasked with outputting $\hat{x}$, a reconstruction of the original input $x$. The bottleneck is a crucial attribute of the network design. Without an information bottleneck, the network could quickly learn to memorize the input values by passing them along through the network. A bottleneck constrains the amount of information that can traverse the full network, forcing a learned input data compression.

![Figure 4.1: This figure shows the overall model structure of an Autoencoder.](image)

The most straightforward architecture for constructing an autoencoder is to constrain the number of nodes present in the network’s hidden layer(s), limiting the amount of information that can flow through it. By penalizing the network according to the reconstruction error, the model
can learn the essential input data attributes and how to best reconstruct the original input from an “encode” state. Ideally, this encoding learns and describes the latent attributes of the input data.

In an autoencoder, the encoder includes some layers that can be fully connected, which will take the input and compress it down to a smaller representation with less dimension than the input. This representation is what is called the bottleneck or the hidden layer. Then from the bottleneck, it tries to reconstruct the input by using a Generative Adversarial Network. Using a variational autoencoder takes the opposite approach. The distribution that’s being followed by the latent vectors is not the question, and target distribution is what the model is seeking to reach.

In just three years, Variational Auto Encoders (VAEs) has emerged as one of the most popular approaches to unsupervised learning of complicated distributions. VAEs are appealing because they are built on top of standard function approximations (neural networks), and can be trained with stochastic gradient descent [6].

Beyond the mathematical explanation of VAEs [55], the idea of VAE, in a nutshell, is that instead of mapping inputs to fixed vectors, map them to distribution.

As shown in Figure 4.2, the only difference between VAE and autoencoder is that the standard bottleneck is replaced by two separate vectors, one representing the mean and the other representing the standard deviation of the distribution. Therefore, when a vector is needed to feed through the decoder network, a sample from distribution is taken and fed through the decoder.

Figure 4.2: This figure overviews the Variational Auto Encoder model in terms of its functionality.
Model Configuration

In all implementations, I set the following parameters for all implementations of the VAE algorithm:

- Number of epochs = 50
- Number of Latent Layers = 20
- Batch size = 128
- Number of Neurons: 100
- Input Image Size: 488 x 488 inches

Where Number of epochs indicates the number of iteration for each algorithm, Number of latent layers indicates the bottleneck layers as representing vector, Batch size is a number of batch members to make predictions on that batch of examples at once. Number of neurons indicates the number of nodes in each layer other than bottleneck, and Input image size indicates size of each image for training. I used all the configurations based on the most frequent number used for each feature in all the implementations.

Evaluation Measurement

To evaluate the generated images, I used a measurement called Fréchet Inception Distance [56].

Fréchet Inception Distance (FID) is one of the metrics for automatically evaluating the quality of image generative models [57] [58] [56]. FID is a measurement that compares two images pixel by pixel. The image size and blurriness can affect the score, which is a float number showing the similarity of images. That being said, comparing one image with itself would have a 0.00 FID score. In this experiment, I use FID to compare input images with the generated images by all implementations. Following is a complete definition of the FID score.
The Inception Score (IS) \cite{salimans_towards_2016} uses the quality of the generated images and their diversity as two criteria. I want the conditional probability $P(y|x)$ to be highly predictable in the generated images. So I use an inception network to classify the generated images and predict $P(y|x)$, the label of $x$. It reflects the quality of the images.

Next, I need to measure the diversity of images. $P(y)$ is the marginal probability computed as:

$$
\int_z P(y|x = G(z)) dz
$$

If the generated images are diverse, the data distribution for $y$ should be uniform. To combine these two criteria, I compute their KL-divergence and use the equation below to calculate IS.

$$
IS(G) = \exp(E_{x \sim p_d}D_{KL}(p(y|x) || p(y)))
$$

In FID, I use the inception network to extract features from an intermediate layer. Then, I model the data distribution for these features using a multivariate Gaussian distribution with mean $\mu$ and covariance $\Sigma$. The FID between the real images $x$ and generated images $g$ is computed as:

$$
FID(x, g) = ||\mu_x - \mu_g||^2 + Tr(\Sigma_x + \Sigma_g - 2(\Sigma_x \Sigma_g)^{1/2})
$$

Where $Tr$ sums up all the diagonal elements. Lower FID values mean better image quality and diversity. The distance increases with simulated missing modes. FID is more robust to noise than IS. If the model only generates one image per class, the distance will be high. So FID is a better measurement for image diversity. FID has some rather high bias but low variance. By computing the FID between a training dataset and a testing dataset, I should expect the FID to be zero since both are real images. However, running the test with different batches of training sample shows non-zero FID.

4.1.3 Problem Experiment

After having all implementations, I test all 10 implementations with the two test inputs. Each implementation took around five hours to generate the results in terms of run time.
the quality of the generated images, I calculated the generated image distance from the original image with the FID threshold of 120. This heuristic threshold is evaluated manually since there is no other previous work of multiple-implementation testing on VAEs.

I also run all the models given their sets of configurations and inputs, to see if the FID score of the generated images is related to the configuration or to the DNN model.

Finally, having the test oracle, I compare different implementations manually to see if a the difference between FID scores of the original image and generated image indicates the existence of any fault.

4.1.4 Problem Discussion and Results

In the following, I go through the answer to this experiment’s research question by discussing the results.

Table 4.1 shows the results on each implementation for VAE algorithm, where \( I_i \) is the \( i^{th} \) implementation. Figure 4.5 shows the generated images by all implementations for the Figure 4.3 input.

Table 4.2 also shows the results of all implementations based on the second input shown in
Table 4.1: This table shows the calculated FID score for each implementation. These results are for the first input image, shown in Figure 4.5.

<table>
<thead>
<tr>
<th>Implementation ID</th>
<th>FID Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>158.89</td>
</tr>
<tr>
<td>$I_2$</td>
<td>229.28</td>
</tr>
<tr>
<td>$I_3$</td>
<td>117.49</td>
</tr>
<tr>
<td>$I_4$</td>
<td>67.5</td>
</tr>
<tr>
<td>$I_5$</td>
<td>205.37</td>
</tr>
<tr>
<td>$I_6$</td>
<td>62.43</td>
</tr>
<tr>
<td>$I_7$</td>
<td>107.74</td>
</tr>
<tr>
<td>$I_8$</td>
<td>95.05</td>
</tr>
<tr>
<td>$I_9$</td>
<td>54.38</td>
</tr>
<tr>
<td>$I_{10}$</td>
<td>135.01</td>
</tr>
</tbody>
</table>

Figure 4.4 shows the FID of each implementation with its default parameters. Despite the other two tables that use the parameters mentioned in Section 4.1.2 in this table, parameters are set the same as the default for each implementation while cloning. Also, the input image is also the default input image for the implementation. The higher the number of epochs, the more time for compiling was needed for each implementation.

In the next step, I combined our implementation FID scores in Table 4.4 by sorting all of them based on their FIDs that can be found in Table 4.1, 4.2, and 4.3 which represents first input, second
Table 4.2: This table shows the calculated FID score for each implementation. These results are for the second input image, shown in 4.4.

<table>
<thead>
<tr>
<th>Implementation ID</th>
<th>FID Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>I₁</td>
<td>166.23</td>
</tr>
<tr>
<td>I₂</td>
<td>257.67</td>
</tr>
<tr>
<td>I₃</td>
<td>125.11</td>
</tr>
<tr>
<td>I₄</td>
<td>84.55</td>
</tr>
<tr>
<td>I₅</td>
<td>193.49</td>
</tr>
<tr>
<td>I₆</td>
<td>76.21</td>
</tr>
<tr>
<td>I₇</td>
<td>115.89</td>
</tr>
<tr>
<td>I₈</td>
<td>102.9</td>
</tr>
<tr>
<td>I₉</td>
<td>68.21</td>
</tr>
<tr>
<td>I₁₀</td>
<td>143.70</td>
</tr>
</tbody>
</table>

input and different parameters and inputs respectively. It is sorted as the lowest FID comes first.

Finally, I set a threshold of $\alpha$ for finding the majority voted implementations. Since there is no previous work for setting the $\alpha$, I chose $\alpha = 120$ based on my observation and comparing the quality of the generated images. The bold items in Table 4.4 show the implementations that are greater than the threshold.

RQ1: How can multi-implementation testing adequately approximate a test oracle for DNN models?

Comparing sorted implementations in Table 4.4 can show that in sorted implementations with both first and second input, $I₁₀, I₁, I₅$ and $I₂$ are the algorithms that are larger than $\alpha = 120$, which means they have the most distance from the original input. In contrast, for the last column, which is default input with default parameters of each implementation, $I₁$ and $I₂$ and $I₁₀$, were the ones that are greater than the threshold.

The results show that while keeping the parameters similar, there is no change in the sorted FID scores. It means that although the amount of FID score might vary based on different input images for each implementation, the quality of generated images does not change if the parameters are all set similarly. On the other hand, the last column shows that I can have different results with
changing the parameters and the input image. Therefore, to answer research question 1, I need to analyze $I_5$, $I_1$, $I_2$ and $I_{10}$ implementations to see if I find any faults. Faults in these implementations define as any reason that causes the model to generate low-quality outputs.

As $I_5$ has not produced images as good as the other algorithms while setting specific parameters rather than choosing default parameters, I can conclude that changing a specific parameter might cause the difference. As shown in Table 4.3, the default number of hidden layers for this implementation is 50, whereas the number of hidden layers set for all models equals 20.

Now, to get closer to the answer of RQ1, I need to compare the implementations of $I_1$, $I_2$ and $I_{10}$ to figure out the reasons why these are not generating images as good as other implementations. After analyzing $I_1$, I figured out that compared with other implementations, what makes it have different results is the noise initially added to the input image. After adding noise, the algorithm tries to generate the image and try to ignore these noises. Some auto encoders add noise to an image to train the neural network so that the model learns how to ignore them. So, by giving the noisy image and asking the algorithm to generate the real image, it tries to learn to ignore noises.

Figure 4.6 shows the input, noisy input and the generated image in epoch one and ten for implementation $I_1$. By looking at the generated image by $I_1$ in Figure 4.6, I can see that adding noise can cause generating worse images. $I_2$ implementation also made the same approach for
Table 4.4: Sorting implementation’s FID considering lowest FID first, based on our three tables: Table [4.1] Input Image 1 - Set parameters; Table [4.2] Input Image 2 - Set parameters; Table [4.3] Default Input Image - Default Parameters

<table>
<thead>
<tr>
<th>Table 4.1</th>
<th>Table 4.2</th>
<th>Table 4.3</th>
</tr>
</thead>
<tbody>
<tr>
<td>I_9</td>
<td>I_9</td>
<td>I_9</td>
</tr>
<tr>
<td>I_6</td>
<td>I_6</td>
<td>I_5</td>
</tr>
<tr>
<td>I_4</td>
<td>I_4</td>
<td>I_7</td>
</tr>
<tr>
<td>I_8</td>
<td>I_8</td>
<td>I_4</td>
</tr>
<tr>
<td>I_7</td>
<td>I_7</td>
<td>I_8</td>
</tr>
<tr>
<td>I_3</td>
<td>I_3</td>
<td>I_6</td>
</tr>
<tr>
<td>I_{10}</td>
<td>I_{10}</td>
<td>I_3</td>
</tr>
<tr>
<td>I_1</td>
<td>I_1</td>
<td>I_{10}</td>
</tr>
<tr>
<td>I_5</td>
<td>I_5</td>
<td>I_1</td>
</tr>
<tr>
<td>I_2</td>
<td>I_2</td>
<td>I_2</td>
</tr>
</tbody>
</table>

Figure 4.6: This figure demonstrates the adding noise process for $I_1$ implementation.

generating the images. Figure 4.7 shows the process of generating images with noise for $I_2$. It
worth mentioning that there might be some different types of adding noise to the input image,
which is not in the scope of this study, but they can affect the quality of the generated images.

Figure 4.7: This figure demonstrates the adding noise process for $I_2$ implementation.

The last implementation that has greater FID than the threshold is $I_{10}$. In this implementation,
the algorithm assigns labels to the images. Since each data under a specific label has its distribution,
it is easy to sample data with a specific label. In this study, the input data is not a single number.
Hence, labelling an image did not add anything to the generated image and made it generate worse
results than expected.

Regarding Table 4.4, I can realize that in the first two columns that the parameters are set similar, there is no change in the order of the FID score of the implementations. Hence, the input does not affect the results. However, comparing these two columns with the third one shows that the results may vary based on the parameters.

**RQ1 Conclusion:**
To make a conclusion based on the discussions mentioned above, I can state that using MIT can help detect any faults that cause different outputs on the implementations for DNN models. Hence, the answer to the first research question is that MIT can help detect any fault or difference in the implementation of DNN models. In this experiment, I also mentioned two reasons why a VAE implementation may not give us proper results.
4.2 Testing DNN: Test Adequacy Evaluation

In the second experiment, I first describe the problem, then go through the design and experiment, and finally discuss the results.

4.2.1 Problem Overview

In this experiment, test adequacy criteria for DNN models are evaluated through a comparison between a coverage-based and mutation-based testing techniques.

Objective

In recent years, multiple test adequacy metrics have been proposed for testing DNN programs. However, there is no direct comparison study between two main groups of adequacy criteria (namely coverage and mutation-based criteria) to detect adversarial examples. In this experiment, I proposed an approach for comparing Surprise Coverage and Model Mutation adequacy metrics to determine which one can have higher increment by adding adversarial examples to the original test set. In this experiment, sensitivity is defined as the increment in the metric given a portion of adversarial examples. Therefore, the higher increment in the score of these metrics, shows higher sensitivity to the adversarial attacks.

The main reason to choose these two categories of criteria is that they are the most studied metrics in both traditional and DNN testing domain.

Research Questions

In our evaluation, I investigate the following research questions:

- RQ2: Which of the two metrics (Surprise Coverage or Label Change Rate) is more sensitive in detecting adversarial examples?
- RQ3: How can changing the metrics’ parameters change the metrics’ sensitiveness?
4.2.2 Problem Design

In this section, I will explain the details of our experiment design. The code and details are available publicly in Github[2].

**Dataset**

I have evaluated the DNN models on the MNIST dataset, an extensive database of handwritten digits commonly used for training various image processing systems citelecun-mnisthandwrittendigit-2010. I have also used the CIFAR-10[59] dataset, a collection of frequently used images to train machine learning and computer vision algorithms. These two datasets have also been used in both original papers that proposed SA and DeepMutation. So our choice will not be biased toward one metric.

**Adversarial Attacks**

Since different attacks may have different behaviour (some harder to detect and some easier for a given test adequacy metric), I need to evaluate the two adequacy criteria under several attacks. I chose FGSM, JSMA, C&W, Deepfool, and Blackbox testing as adversarial attacks for RQ1 and RQ2, which were used in both original papers of Surprise Adequacy and Model Mutation[15][11].

**DNN Model Under Study**

In Surprise Adequacy paper[11], the targeted deep learning models for MNIST and CIFAR-10 datasets are 5-layer convolution networks and 12-layer convolution networks respectively, while the deep learning models in model mutation paper[13] is LeNet and GoogleNet respectively. To be unbiased, I selected one model from each paper, per dataset as follows:

- MNIST: LeNet and 5-layer convolution networks (with 98.6% and 95.3% accuracy respectively)

• CIFAR-10: *GoogleNet* and *12-layer convolution networks* (with 90.5% and 80.7% accuracy respectively)

**Model Configuration**

To answer each research question, a separate configuration is needed, as below. Settings for RQ2 are:

- **Surprise Coverage Configuration:** I set the default activation threshold for LSC metric to $10^5$, same as the original paper [11], as well as setting the deepest hidden layer in each model when computing LSC. That is because Kim et al. [11] could not find any correlation with the depth of the layer [15], and their experiment is also on the deepest hidden layer.

- **Mutation Model Configuration:** Two configurations should be set in the model mutation method. First, the mutation operator that generates mutated models, and second, mutation rate. For this experiment, I choose Gaussian Fuzzing (GF) since it was more effective than the others in the original paper. By generating ten model mutants for each model, I address its internal randomness as well. I also choose the lowest mutation rate of 0.01 for the MNIST dataset, as there is no suggestion in the paper.

In RQ3, the goal is to measure the sensitivity of the metrics concerning parameter changes. Therefore, several configurations should be set. Each configuration changes one aspect of the experiment compared to RQ2 and keeps the rest the same. Following are the elements I am controlling in RQ3:

- **Choice of Dataset and Model.**

As described, I use MNIST dataset and *LeNet* and *5-layer convolution networks* models for experimenting RQ2. For answering how sensitive the metrics are in
RQ3, I change the MNIST dataset to CIFAR10, and change the models to GoogleNet and 12-layer convolution networks. I also use different mutation rates for CIFAR-10 models (0.007), as suggested by the original LCR paper.

- **Choice of Surprise Coverage Layer**: One of LSC’s essential hyper-parameters is the layer chosen for measuring the coverage. In RQ2, I decided on the deepest hidden layer, as suggested and used in the original paper. In RQ3, however, I use the first hidden layer. Note that I do not repeat the experiment with all multiple attacks and models since I found interesting observations in the first random case (i.e., MNIST dataset, LeNet Model, and FGSM attack).

- **Choice of Mutation Rate**: In RQ2, I use the lowest mutation rate for MNIST and highest for CIFAR10, according to the options examined in the original LCR paper [15]. It is because there is no explicit suggestion in the paper on how to choose the rate. In RQ3, I change the rate on MNIST from the lowest (0.01) to the highest (0.05). I expect that this change increases the increments as it was observed in the original paper. I used the MNIST dataset and LeNet Model and picked JSMA as a random attack.

- **Choice of Mutation Operator**: In the original LCR paper, three mutation operators out of six mentioned in the DeepMutation paper have been implemented and evaluated, since the other operators are not suitable for the study. Again, since there is no explicit winner among these three (thus no suggestion by the paper), I randomly picked one in RQ2 (GF) and examined the other two, Neuron Switch (NS) and Weight Shuffling (WS) operators, in RQ3. I used the same setup as the previous study, MNIST dataset, LeNet Model, and JSMA attack since both evaluate LCR hyper-parameters.
**Evaluation Measurement**

For the Surprise Adequacy criterion, I have used Likelihood-based Surprise Coverage (LSC) as the coverage metric and not DSA. DSA is only used to classify the adversarial and original examples, not for calculating the neurons’ coverage. For DeepMutation, the only choice is the Label Change Rate (LCR), as explained in Section 2.

4.2.3 Problem Experiment

To best see the adequacy metric’s effect in detecting adversarial examples, I do not follow the design of previous papers that add all adversarial samples from one attack at once and observe the difference in the metric values [15] [11]. Instead, I incrementally add the adversarial examples (1% at a time) to the original test dataset and observe the change of LCR and LSC compared to the original data.

Table 4.5 shows the number of adversarial test inputs for each attack using MNIST and CIFAR-10 datasets. I begin with a set of original (non-adversarial) test inputs for each experiment, the same size as the generated adversarial attacks. I choose this set randomly out of all original test data from either dataset.

For the original test inputs \( O \), I add 1% of adversarial inputs \( A \) to the original input in each iteration and then calculate the new test set’s adequacy criteria. That means that I am updating the set of test inputs in each iteration and measuring the new LCR and LSC scores. This way, I can see the effectiveness of LCR and LSC scores at detecting adversarial examples in fine grain. The set of test input \( T_i \) is updated as follows, for \( i = \{1, 2, ..., 100\} \):

\[
T_i = T_{i-1} + A_i
\]  

(4.1)

Where \( A_i \) is 1% of remaining adversarial \( A \) in each iteration and \( T_i \) is total test inputs in \( i^{th} \) iteration. Note that \( T_0 \) and \( A_0 \) equal to the number of original input and the number of attacks. It is worth to mention that since I am using 1% of adversarial attacks each time, I round the number
Table 4.5: Number of adversarial examples per attack, per dataset

<table>
<thead>
<tr>
<th>Adversarial Attacks</th>
<th>MNIST</th>
<th>CIFAR-10</th>
</tr>
</thead>
<tbody>
<tr>
<td>Black-Box</td>
<td>2000</td>
<td>1000</td>
</tr>
<tr>
<td>Deepfool</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>FGSM</td>
<td>2000</td>
<td>2000</td>
</tr>
<tr>
<td>JSMA</td>
<td>1000</td>
<td>2000</td>
</tr>
<tr>
<td>C&amp;W</td>
<td>700</td>
<td>1000</td>
</tr>
</tbody>
</table>

Of attacks down to the closest hundreds, in Table I. For instance, the total generated attacks using C&W for MNIST is 730, but I use 700 to choose seven attacks in each iteration.

To address the randomness of this experiment design, I repeat the process ten times for each attack on each dataset. The median values are calculated to report LSC and LCR increments.

4.2.4 Problem Discussion and Result

In this section, I discuss the results of the experiments under each research question.

RQ2: Which of the two metrics (MS or SC) is more sensitive in detecting adversarial examples?

To evaluate LCR and LSC, I calculated the scores on the two models (LeNet and 5-layer convolution networks), using the MNIST dataset and five different attacks.

First, I calculated the correlation between "Percentage of Adversarial Examples" and "The Adequacy Score (LCR and LSC) Increments Compared to the Original Test Data", using Bivariate Pearson Correlation [60] [61][62]. As Table 4.6 shows (note that Table 4.6 contains correlation values for all 40 experiments in both RQs), all the correlations are very high (larger than 90%), which suggest that both LCR and LSC are potentially sensitive in distinguishing the adversarial examples. Again, sensitivity is defined as any increment or decrement on the score of the test adequacy. However, the correlation values alone do not reveal the magnitude of difference when an adversarial sample is measured through LCR and LSC. In other words, in practice, the way such metrics can be useful is to return a very different (much higher or lower) score when applied on the
adversarial samples compared to the normal tests. Therefore, a metric can be highly correlated (increasing with the same pattern as adversarial tests are added) but not very practical. What happens if the increments are very low.

Figures 4.14 and 4.9 depicts increments of a score (LCR and LSC) when some random adversarial samples are added to the test set (at every step, 1% extra adversarial samples are added). By looking at the graphs, I observe that for all adversarial attacks on both DNN models, the mutation-based metric (LCR) has by far higher increments than the coverage-based metric (LSC).

Figure 4.8: MNIST dataset targeted the 5-layer-convolutional model with five different adversarial attacks.

**RQ2 Conclusion:**

Therefore, the conclusion for RQ2 is that, given the suggested parameters, LCR is more sensitive than LSC in detecting adversarial examples because the higher increment in the metric can be seen in the experiment results.

**RQ3: How can changing the metrics’ parameters change the metrics’ sensitiveness?**

To answer RQ3, I analyze the question from the following aspects:

**Choice of Dataset and Model** As shown in Figure 4.10 and 4.11, except for the Black-box attack on GoogleNet, where LCR is as low as LSC, the other nine attack-models show the very
same pattern of LCR dominating the LSC results. Therefore, given our data, I conclude that the RQ2 results are robust concerning the change of dataset and models.

**Choice of the Surprise Coverage Layer** As shown in Figure 4.13, surprisingly, changing the layer resulted in a higher likelihood surprise coverage (LSC) increments, but also made LSC better than LCR. This one example was enough to illustrate that the choice of adequacy metric hyper-parameter (here the layer in LSC) is important in detecting adversarial. In other words, with proper tuning, one may or may not use LSC compared to LCR.
Figure 4.11: CIFAR-10 dataset targeted GoogleNet model with five different adversarial examples.

Table 4.6: Pearson correlation between "Percentage of Adversarial Examples" and "The Adequacy Score (LCR and LSC) Increments Compared to the Original Test Data".

<table>
<thead>
<tr>
<th>Model</th>
<th>Black-Box</th>
<th>Deepfool</th>
<th>FGSM</th>
<th>JSMA</th>
<th>C&amp;W</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SC</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeNet</td>
<td>0.987</td>
<td>0.929</td>
<td>0.99</td>
<td>0.978</td>
<td>0.947</td>
</tr>
<tr>
<td>Conv5</td>
<td>0.994</td>
<td>0.982</td>
<td>0.98</td>
<td>0.985</td>
<td>0.986</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>0.988</td>
<td>0.956</td>
<td>0.963</td>
<td>0.969</td>
<td>0.972</td>
</tr>
<tr>
<td>Conv12</td>
<td>0.98</td>
<td>0.989</td>
<td>0.98</td>
<td>0.99</td>
<td>0.986</td>
</tr>
<tr>
<td><strong>Mutation</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LeNet</td>
<td>0.985</td>
<td>0.984</td>
<td>0.984</td>
<td>0.982</td>
<td>0.984</td>
</tr>
<tr>
<td>Conv5</td>
<td>0.984</td>
<td>0.984</td>
<td>0.985</td>
<td>0.984</td>
<td>0.984</td>
</tr>
<tr>
<td>GoogLeNet</td>
<td>0.978</td>
<td>0.984</td>
<td>0.985</td>
<td>0.984</td>
<td>0.982</td>
</tr>
<tr>
<td>Conv12</td>
<td>0.983</td>
<td>0.989</td>
<td>0.98</td>
<td>0.9985</td>
<td>0.98</td>
</tr>
</tbody>
</table>

**Choice of Mutation Rate** Figure 4.14 shows that improving the mutation rate can result in more increments in the Label Change Rate (LCR). LCR increases and it does not change the pattern of LCR having higher increments than LSC. However, changing the mutation rate can change the increment values that support the previous experiment claim that with proper tuning, one may or may not use LCR.

**Choice of Mutation Operator** Figure 4.12 shows the results. As shown, the Neuron Switch (NS) operator is not even visible in the diagram, since its increments are negative. For Weight Shuffling (WS) operator, the increments are positive, but they are not as high as the default opera-
tor GF. Again, the LCR’s effectiveness seems to be sensitive to the choice of the operator.

**RQ3 Conclusion:**

Therefore, RQ3 results conclude that both metrics are sensitive to their parameters. Their performance can be affected by a wrong parameter setup, to the point that it would not be predictive anymore. Therefore, hyper-parameter tuning is essential for testing tools and metrics in this domain.
4.3 Testing DNN: Test Input Generation

This section goes through the third experiment’s design, as well as results and discussions.

4.3.1 Problem Overview

In the third experiment, I investigate the third aspect of testing DNNs, test input generation. The goal is to generate test inputs (adversarial samples) for DNN models that find issues with the model (e.g., misclassification) and then fix the model by re-training the models with those generated test inputs.

**Objective**

In this experiment, first, I generate adversarial examples using three different methodologies. Next, I examine the model’s performance on three downstream tasks in software engineering, and finally re-train the models using the generated samples, and test whether there is any improvement in the F1-score.

**Research Questions**

In this experiment, I seek to answer three following research questions:

- RQ4) How does adversarial test input change the model evaluation score?

- RQ5) How does re-training the model impact the F1-score?

- RQ6) Does the choice of evaluation task have any effect on improving the model’s performance?

4.3.2 Problem Design

In this experiment, the focus is to generate adversarial examples to test DNN models used in software engineering tasks such as code summarization and comment generation. Most software engineering tasks require code snippets as the input, requiring code embeddings to feed the DNNs.
In this thesis, I test three well-known DNN models for generating code embeddings, and evaluate their trained model on three different downstream tasks discussed in Section 4.3.2.

**DNN Model Under Study**

In software engineering, there are three well-known tools for embedding code inputs called Code2vec [16], Code2seq [17] and CodeBERT [18]. All three models suggest a method for embedding code snippets since a wide variety of software engineering applications like code summarization, documentation, and retrieval require them. Code2vec presents a neural model for representing snippets of code as continuously distributed vectors, where Code2seq suggests a transformation to generate natural language sequences from source code snippets. Finally, both models evaluate their method by predicting the method’s name based on the given body.

Code2vec models the source code to the AST paths. An AST path is defined as a path between nodes in the AST, starting from one terminal, ending in another terminal, and passing through an intermediate non-terminal in the path, a common ancestor of both terminals. Both source and destination terminals, along with the AST path, are mapped into an embedding vector, which is learned jointly with other network parameters during training. Each terminal and the path is then concatenated to a single context vector called path-context. It is also an attention vector to score each path-context to a single code vector representing the method’s body.

The Code2seq model uses an encoder-decoder architecture to encode paths node-by-node and create labels as sequences at each step. The encoder represents a method’s body as a set of AST paths where each path is compressed to a fixed-length vector using a bi-directional LSTM, which encodes paths node-by-node. The decoder uses attention to select relative paths while decoding and predicts sub-tokens of target sequence at each step when generating the method’s name. In contrast, code2vec uses monolithic path embeddings and only generates a single label at a time.

CodeBERT also learns general-purpose representations that support downstream software engineering applications such as natural language code search and code documentation generation. It is a bi-modal pre-trained model for natural language (NL) and programming language (PL) like
Dataset

In this study, I use original raw java files as input and generate adversarial examples using original ones. In the following, I would first explain the original dataset details and then talk about the adversarial input generation.

**Original Java Files:** Both Code2vec and Code2seq tools support Java and C# languages as the source code snippet input. Code2seq also supports Python language. CodeBERT supports the mentioned languages, in addition to JavaScript, PHP, Ruby, and Go.

Code2vec published dataset is in a pre-processed format, but in this study, I needed the raw files to generate adversarial examples. It also splits the dataset to train, validation, and test sets by a single Java file, whereas Code2seq splits them by project folders. Fortunately, the datasets, accompanied by Code2seq, contained the raw java files. CodeBERT also provided pre-processed training and validation datasets and provided the code for pre-processing the test dataset. Therefore, I used the Java-Large dataset in the Code2seq Github page for all three models, including 9000 Java projects for training, 200 for validation, and 300 for testing. This dataset contains about 16M examples.

Since Code2vec splits the dataset by a single Java file, I first moved all the java files in a single folder and then used it for the training. Overall, I had about 1.9M java files. The compressed size is about 4GB, and the extracted size is about 20GB.

**Adversary Java Files:**

As discussed in Section 2.7, there are several methods to generate adversarial examples for text and image input. However, they are not usable when it comes to the code snippet. An alternate method to generate adversarial examples for code snippets without losing critical information or any change to the code structure is code refactoring. For this study, I used ten refactoring methods to generate adversarial examples, as listed in Section 2.7.

Applying applicable refactoring methods to the raw java files in both training and testing
Table 4.7: Model’s original F1-score reported on the corresponding paper with default configurations on default dataset.

<table>
<thead>
<tr>
<th>Tool</th>
<th>F1-score</th>
<th>Downstream task</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code2vec</td>
<td>58.4</td>
<td>Method name’s prediction</td>
</tr>
<tr>
<td>Code2seq</td>
<td>59.19</td>
<td>Method name’s prediction</td>
</tr>
<tr>
<td>CodeBERT</td>
<td>74.84</td>
<td>Code Search</td>
</tr>
</tbody>
</table>

datasets, the total number of original programs is 1,798,419 training files, 44,140 validation files, and 59,404 test files. I generated the same amount of adversarial Java files given the amount of the original data, using three different methods below:

1. **1-Time Mutation:** Apply refactoring operator on a Java method once.

2. **K-Time Mutation:** Apply five different refactoring operator on a Java method.

3. **Guided Mutation:** Apply refactoring operators on the Java method by guiding the test input using a mutation score for 100 iterations.

Note that mutation in these three techniques defines applying refactoring methods on each java file to generate an adversarial example of the corresponding java file. Section 4.3.3 describes each generative model in details.

**Model Configuration**

Code2vec, Code2seq, and CodeBERT tools are all publicly available to replicate. Table 4.7 shows the F1 score reported in each paper for each tool. While the trained model for Code2seq and CodeBERT meets the performance reported in their corresponding paper, Code2vec could not reach the F1 score reported in the original paper because of lacking available datasets publicly. Additionally, because of transferring all the java files in a single folder, the F1 score for Code2vec is also different from the reported one in code2seq paper using the Java-Large dataset.

In the experiment, I set the number of epochs as 20 and kept other configurations, as suggested in the corresponding papers. For the GM experiment, I set the mutation rate as 0.05, as studies
show that it is a good mutation rate for a genetic algorithm problem [63].

**Evaluation Measurement**

In this study, I evaluate the trained model on three different downstream tasks: Method Name Prediction, Code Captioning and Code Search.

- **Method Name Prediction**: Predict method’s name given the method’s body. The evaluation metric is F1-score over sub-tokens.

- **Code Captioning**: Predict a full natural language sentence given a short Java code snippet. The target sequence length in this task is about ten on average. The model is evaluated using ROUGE-N and ROUGE-L F1-Score.

- **Code Search**: Given a natural language as the input, the objective of code search is to find the most semantically related code from a collection of codes. The evaluation metric is F1-score.

ROUGE, or Recall-Oriented Understudy for Gisting Evaluation [64] is a set of metrics and a software package used for evaluating automatic summarization and machine translation software in natural language processing. The metrics compare an automatically produced summary or translation against a human-produced summary or translation. The following five evaluation metrics are available.

- **ROUGE-N**: Overlap of N-grams [65] between the system and reference summaries.

  \( \text{ROUGE-1} \) refers to the overlap of unigram (each word) between the system and reference summaries.

  \( \text{ROUGE-2} \) refers to the overlap of bigrams between the system and reference summaries.
4.3.3 Problem Experiment

In this section, all three adversary input generation is described.

**1-Time Mutation:** As shown in Figure 4.15 in the 1-Time Mutation method, I refactor all the original files just once. Precisely speaking, first, the model extracts each Java method and then randomly selects a refactoring operator from the pool of operators to apply on the method. Note that some of the random refactoring techniques might not be applicable to a Java method. For instance, if the specific method does not contain any loop, the randomly chosen Loop Enhance method cannot be applied there. Therefore, I iterate the process until I make sure that the method is refactored.

Once all methods are extracted and refactored, the adversarial Java files are generated. Figure 4.19 is a 1-time refactored sample, using Argument Adding refactoring operator.

**K-Time Mutation:** K-time approach is similar to the 1-time approach, with refactoring a single method for K times. After extracting each Java method, a randomly selected refactoring operator
is applied, and this process would be repeated for $K$ times per method. In other words, each Java method is refactored $K$ times with $K$ randomly chosen operators, as shown in Figure 4.16. Again, some of the random refactoring techniques might not be applicable. Therefore, I iterate the process with different operators to make sure the method is refactored. In this thesis, I consider $K = 5$, as Thomas et al. [20] evaluated the different $K$s, and suggested that $K = 5$ has the best F1 score. Figure 4.20 shows the code snippet sample generated by the 5-time adversarial technique. As specified, Local Variable Renaming, Argument Adding, For Loop Enhance, Add Print and Method Name Renaming refactoring operators have been used in this sample code snippet.

**Guided Mutation:**

This experiment introduces and proposes a Guided Mutation (GM) methodology to generate adversarial inputs for code embeddings. Guided mutation method is an evolutionary strategy inspired by the Genetic Algorithm (GA) in terms of the structure, but without applying crossover on the input generation. Once the genetic representation and the fitness function are defined, a GA proceeds to initialize a population of solutions and then improve it through repetitive application of the mutation and crossover. As discussed, in the GM method, I only apply mutation and not
crossover on the inputs. The reason is that changing the code snippets using crossover may cause many wrong (not even compilable) code snippets, let alone functionality preserving code.

Elitism in GA involves copying a small proportion of the fittest candidates, unchanged, into the next generation. It can sometimes have a dramatic impact on performance by ensuring that the EA does not waste time re-discovering previously discarded partial solutions. Candidate solutions that are preserved unchanged through elitism remain eligible for selection as parents when developing the next generation’s remainder.

As shown in Figure 4.17, following are the steps to generate adversarial samples using GM methodology:

1. Calculate mutation score for the current generation.
2. Choose elite candidates based on the highest mutation score and copy them into the next generation.
3. Mutate the remaining candidates with specified mutation rates.
4. Repeat step one until meeting the stopping criterion (reaching 100 iterations).
DeepMutation++ [66] is a mutation testing framework to analyze the test data quality. Therefore, in this thesis, I use the mutation score metric in DeepMutation++ to evaluate the quality of test data that I generate using GM method.

DeepMutation [13] first introduced eight model-level operators for feed-forward neural network (FNN) models to generate model mutants. Later, DeepMutation++ proposes nine new operators specialized for recurrent neural networks (RNN). It supports static mutant generation to analyze the test data as a whole and dynamic mutant generation to detect the vulnerable segments of a test input. Table 4.8 shows the 9 operators defined by DeepMutation++.

DeepMutation++ first leverages the provided mutation operators to generate a set of high-quality DNN mutants with a user-specified quality threshold. After making a certain number of mutants, DeepMutation++ analyzes the behaviour differences of original DNN and the generated DNN mutants against the provided test inputs. Finally, it outputs test data quality by giving the mutation score given the specific model.

Given an input $t$, a DNN $m$ and its mutant $m'$, they define $t$ is killed by $m'$ if the outputs are inconsistent at $t$, i.e., $m(t) \neq m'(t)$. Given a set pf mutant DNNs, $M$, they define the mutation score as:

$$MS(t,m,M) = \frac{|\{m' | m' \in M \land m(t) \neq m'(t)\}|}{|M|} \quad (4.2)$$
### Table 4.9: Code2vec F1-scores on original and re-trained models evaluating on method name’s prediction task

<table>
<thead>
<tr>
<th>Code2vec - Method Name’s Prediction</th>
<th>Test</th>
<th>Original</th>
<th>1-time</th>
<th>5-time</th>
<th>GM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Train</strong></td>
<td><strong>Original</strong></td>
<td>35.9251</td>
<td>34.7597</td>
<td>33.8735</td>
<td>35.4537</td>
</tr>
<tr>
<td></td>
<td><strong>1-time</strong></td>
<td>36.4816</td>
<td>35.9213</td>
<td>35.7913</td>
<td>38.7394</td>
</tr>
<tr>
<td></td>
<td><strong>5-time</strong></td>
<td>36.4008</td>
<td>35.4896</td>
<td>36.1684</td>
<td>37.0826</td>
</tr>
<tr>
<td></td>
<td><strong>GM</strong></td>
<td>36.1465</td>
<td>47.3447</td>
<td>46.5198</td>
<td>53.1309</td>
</tr>
</tbody>
</table>

In this experiment, I used the mutation score as the fitness function to evaluate the test inputs for the GM model. I use all nine operators for the RNN models and created ten mutants using each operator. Therefore, overall I have 90 mutants for the model. Figure 4.21 displays adversarial examples generated by GM using Local Variable Renaming and IF Loop Enhance operators.

**Retraining**

Retraining refers to re-running the process that generated the previously selected model on a new data training set. The features, model algorithm, and hyper-parameter search space should all remain the same.

In this experiment, I retrain the model using generated adversarial sets of examples to experiment if the retrained model has learned better or not. The learned model with newly generated data is expected to be more robust while testing it with adversarial test sets.

### 4.3.4 Problem Discussion and Results

In this section, I first report the results, and then discuss them per research question.

Table 4.9, 4.10 and 4.11 indicates the results for code2vec and code2seq experiments respectively. The columns represent the original test data as well as adversarial ones, and the rows show the original and retrained models using adversarial train data.

In the following, I discuss the results over research questions:

**RQ4: How does adversarial test input change the model evaluation score?**

As shown in Tables 4.9, 4.10 and 4.11, columns state the testing result on the down streaming
Table 4.10: Code2seq F1-scores on original and re-trained models evaluating on method name’s prediction task

<table>
<thead>
<tr>
<th>Code2seq - Method Name’s Prediction</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>Train</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>42.7167</td>
</tr>
<tr>
<td>1-time</td>
<td>39.5952</td>
</tr>
<tr>
<td>5-time</td>
<td>39.9891</td>
</tr>
<tr>
<td>GM</td>
<td>41.1948</td>
</tr>
</tbody>
</table>

To answer this research question, I first discuss the test results for the trained model using the original dataset. As shown in Tables 4.9, testing the original trained model on the original test cases has an F1-Score of 35.92. This score decreases given adversarial test inputs. This result shows that given adversarial test inputs, the original model loses score due to unseen dataset.

Same scenario happens for Table 4.10 and 4.11. The test evaluation of adversarial examples have a reduced score than the original test data.

**RQ4 Conclusion:**

Therefore, the answer to this question is that testing the original model using adversarial examples reduce the accuracy and F1-Score of the model.

**RQ5: How does re-training the model impact the F1-score?**

This section discusses the impacts of retraining the model on the F1-score with the three different adversarial examples generated for test and train data.

In all tables, retraining the model using adversarial examples cause the score reduction for the original test dataset. It is expected since the model might be over-fitted through the learning process, and then, the results might not be as good as the trained model using the original dataset.

As shown in all three mentioned tables, comparing the 1-time, 5-time and GM results after retraining results in better scores than the original model testing with adversarial examples. This
Table 4.11: CodeBERT F1-score on original and re-trained models evaluating on code search task

<table>
<thead>
<tr>
<th>CodeBERT - Code Search</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>Train</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>81.36</td>
</tr>
<tr>
<td>1-time</td>
<td>80.16</td>
</tr>
<tr>
<td>5-time</td>
<td>80.53</td>
</tr>
<tr>
<td>GM</td>
<td>81.68</td>
</tr>
</tbody>
</table>

Table 4.12: Code2seq Rouge-1 F1-score on original and re-trained models evaluating on code captioning task

<table>
<thead>
<tr>
<th>Code2seq - Code Captioning</th>
<th>Test</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Original</td>
</tr>
<tr>
<td>Train</td>
<td></td>
</tr>
<tr>
<td>Original</td>
<td>52.0947</td>
</tr>
<tr>
<td>1-time</td>
<td>40.9169</td>
</tr>
<tr>
<td>5-time</td>
<td>34.6725</td>
</tr>
<tr>
<td>GM</td>
<td>36.1937</td>
</tr>
</tbody>
</table>

concludes that with adversarial examples retraining the model, the model can have better test scores overall. As it is seen, retraining the model using GM test data has the most F1-score through all the models and tests.

**RQ5 Conclusion:**

Therefore, I conclude that retraining the model improves the model’s performance on the same downstream task.

**RQ6: Does the choice of evaluation task have any effect on improving the model’s performance?**

In the last experiment, I used three different downstream tasks to evaluate our models. Code2vec and Code2seq both have been assessed on name prediction, Code2seq also been assessed on code captioning, and finally, CodeBERT evaluated on code search task.

As shown in Table 4.9, the F1-score on method name’s prediction for Code2vec has been improved by 17.27%. Similarly, we can see a 13.52% improvement for Code2seq on the method
name’s prediction. Code2seq for code captioning task also had a noticeable improvement, as shown in Table 4.12.

Although there is a small increment in Table 4.11 after retraining the model with the three adversarial examples, the improvement indicates that apart from the evaluation task, retraining the model using adversarial samples improves the model performance.

**RQ6 Conclusion:**
Retraining a code embedding model using adversarial samples improves the model performance, despite the downstream task evaluating the model.
Figure 4.18: Original Java sample code snippet from Java-Large dataset.

```java
private static JoinPoint currentJoinPoint() {
    MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
    if (!(mi instanceof ProxyMethodInvocation)) {
        throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
    }
    ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
    JoinPoint jp = (JoinPoint) pmi.getAttribute(JOIN_POINT_KEY);
    if (jp == null) {
        jp = new MethodInvocationProceedingJoinPoint(pmi);
        pmi.setUserAttribute(JOIN_POINT_KEY, jp);
    }
    return jp;
}
```

Figure 4.19: Generated code adversarial using 1-Time refactoring method. The underlined lines indicate the changes from original code snippet.

```java
private static JoinPoint currentJoinPoint(String currentJoinPoint) {
    MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
    if (!(mi instanceof ProxyMethodInvocation)) {
        throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
    }
    ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
    JoinPoint jp = (JoinPoint) pmi.getAttribute(JOIN_POINT_KEY);
    if (jp == null) {
        jp = new MethodInvocationProceedingJoinPoint(pmi);
        pmi.setUserAttribute(JOIN_POINT_KEY, jp);
    }
    return jp;
}
```

Figure 4.20: Generated code adversarial using 5-Time refactoring method. The underlined lines indicate the changes from original code snippet.

```java
private static JoinPoint currentJoinPoint(String newJoinPoint) {
    MethodInvocation mi = ExposeInvocationInterceptor.currentInvocation();
    if (!(mi instanceof ProxyMethodInvocation)) {
        throw new IllegalStateException("MethodInvocation is not a Spring ProxyMethodInvocation: "+ mi);
    }
    ProxyMethodInvocation pmi = (ProxyMethodInvocation) mi;
    JoinPoint jp = (JoinPoint) pmi.getAttribute(JOIN_POINT_KEY);
    for (int i = 0; i <= 5; i++) {
        if (jp == null) {
            jp = new MethodInvocationProceedingJoinPoint(pmi);
            pmi.setUserAttribute(JOIN_POINT_KEY, jp);
        }
    }
    return jp;
}
```

Figure 4.21: Generated code adversarial using Guided Mutation refactoring method. The underlined lines indicate the changes from original code snippet.

```java
private static JoinPoint currentJoinPoint() {
    ProcessInvocation returningName = ExposeInvocationInterceptor.currentInvocation();
    if (!(returningName instanceof ProxyProcessInvocation)) {
        throw new IllegalStateException("ProcessInvocation is not a Spring ProxyProcessInvocation: "+ returningName);
    }
    ProxyProcessInvocation declaringVariable = (ProxyProcessInvocation) returningName;
    JoinPoint aspectInstancefactory = (JoinPoint) declaringVariable.getAttribute(JOIN_POINT_KEY);
    if (aspectInstancefactory == null \&\& returningName == returningName) {
        aspectInstancefactory = new MethodInvocationProceedingJoinPoint(declaringVariable);
        declaringVariable.setUserAttribute(JOIN_POINT_KEY, aspectInstancefactory);
    }
    return aspectInstancefactory;
}
```
Chapter 5

Limitations of the study

In this chapter, I discuss the limitations of the thesis and threats to the validity of the experiments, per study.

5.1 Testing DNN: Test Oracle Generation

One of the limitations of this experiment is that I only evaluate the Oracle-based method on the VAE algorithm. Since DNNs have many applications, I would apply more DNN models to test the oracle problem in future work.

Another limitation is about the effectiveness of multiple implementation testing when there are not deviation-free implementations, i.e., those whose results are the same as the majority output across all test cases. I will work on this in future work as well.

In addition to the mentioned limitations, in this methodology, I cannot say how good the other implementations are, especially since there is no specific threshold. It cannot show that the other implementations generate images with very high quality. Also, comparing implementations based on their default parameters and input images cannot guarantee that the only reason for having a difference in FIDs is different parameters. Each algorithm might use some input images with some specifications that are not considered in the other implementations.

5.2 Testing DNN: Test Adequacy Evaluation

DNN programs can come in many different shapes (different datasets, models, and different configurations of the models). Besides, DNN-based testing criteria also have multiple parameters. Therefore, although I examined the effect of some of the variations in this study, I did not cover an
exhaustive combination of possibilities. However, the point is that I managed to observe the lack of robustness concerning some of these changes.

Moreover, I replicated the two baseline papers as much as possible. I used the same models, dataset, attacks, mutation operator, and mutation rate options. The only part that was slightly different was that I used some new combinations of configurations (e.g., applying LSC on a model from the LCR paper and vice versa) to compare both metrics fairly.

Regarding conclusion validity, I calculated the correlations between LCR/LSC and the other adversarial examples before evaluating the techniques using the graphs’ raw values. I also addressed the randomness of the results by repeating each experiment 10 times (whenever there was a random component in the algorithm) with different random seeds and reported them.

5.3 Testing DNN: Test Input Generation

DNN models can be applied in many different downstream tasks. Although this thesis covers three downstream tasks, more applications like comment generation and code summarization may need to be tested through this experiment in the future.

Another limitation that the experiment faces is using only Java language code snippets as the input. It worth adding more programming languages like Python and C# to see if the performance of the refactoring-based adversarial samples is dependant on the programming language or not.
Chapter 6

CONCLUSION

Providing robust, safe, and secure Deep Neural Networks (DNN) is one of the main challenges of the current machine learning system. Software testing is the most practical tool in the industry to achieve those goals in traditional software systems.

Regarding the importance of the testing DNNs, several papers have proposed testing techniques and adequacy criteria for testing DNNs. This thesis consists of three different testing techniques. In the first technique, I use multi implementation testing to generate test oracle. In the second experiment, I compare two adequacy metrics in terms of their effectiveness for detecting adversarial examples. The last experiment applies three different test generation techniques and compares their performance if the generated test data are used to re-train the models.

The first experiment proposed an approach of multiple-implementation testing for a DNN model called Variational Auto Encoder. The evaluations on VAE exhibited that the majority-voted oracle, produced by multiple-implementation testing, is an adequate proxy of a test oracle. The second experiment compares two main categories of testing techniques that help DNNs become more robust, particularly adversarial examples. I compare two leading adequacy metrics (Surprise Adequacy and DeepMutation) from Coverage and Mutation-based DNN testing categories regarding their effectiveness in detecting adversarial examples. In the last experiment, I applied three different test generation techniques to evaluate the DNN models used in code embedding tasks.

First experiment shows that considering the measurement (FID) threshold equals 120, there were three implementations out of 10 that showed a different result. By comparing their implementation manually, I recognized that two implementations could not produce good results since their implementation added noise to the input image at the beginning of the training. The other implementation used labelling for the input image, but since our inputs were not a single number,
labelling does not help generators to produce better images.

Second experiment shows that though both adequacy metrics from Coverage and Mutation-based DNN testing categories can be highly effective. Additionally, they express that their performance is very sensitive to the hyper-parameter values, and thus tuning is crucial for DNN testing tools.

And finally, last experiment’s results indicate that re-training the models using adversarial examples can improve the model performance by up to 18%, compared to the original state of the art.

In the conclusion of this thesis, from the three conducted studies, I can say that testing DNN models are effective in software engineering, as it is used in a variety of software systems. However, suitable testing techniques must be chosen based on the context, and this requires careful study and cautious experiments.
Bibliography


