Concolic Testing for Deep Neural Networks*

Youcheng Sun
University of Oxford, UK
youcheng.sun@cs.ox.ac.uk

Min Wu
University of Oxford, UK
min.wu@cs.ox.ac.uk

Wenjie Ruan
University of Oxford, UK
wenjie.ruan@cs.ox.ac.uk

Xiaowei Huang
University of Liverpool, UK
xiaowei.huang@liverpool.ac.uk

Marta Kwiatkowska
University of Oxford, UK
marta.kwiatkowska@cs.ox.ac.uk

Daniel Kroening
University of Oxford, UK
kroening@cs.ox.ac.uk

ABSTRACT

Concolic testing combines program execution and symbolic analysis to explore the execution paths of a software program. In this paper, we develop the first concolic testing approach for Deep Neural Networks (DNNs). More specifically, we utilise quantified linear arithmetic over rationals to express test requirements that have been studied in the literature, and then develop a coherent method to perform concolic testing with the aim of better coverage. Our experimental results show the effectiveness of the concolic testing approach in both achieving high coverage and finding adversarial examples.

KEYWORDS

neural networks, symbolic execution, concolic testing

ACM Reference Format:
Youcheng Sun, Min Wu, Wenjie Ruan, Xiaowei Huang, Marta Kwiatkowska, and Daniel Kroening. 2018. Concolic Testing for Deep Neural Networks. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3238147.3238172

1 INTRODUCTION

Deep neural networks (DNNs) have been instrumental in solving a range of hard problems in AI, e.g., the ancient game of Go, image classification, and natural language processing. As a result, many potential applications are envisaged. However, major concerns have been raised about the suitability of this technique for safety- and security-critical systems, where faulty behaviour carries the risk of endangering human lives or financial damages. To address these concerns, a (safety or security) critical system implemented with DNNs, or comprising DNN-based components, needs to be thoroughly tested and certified.

*Huang, Kroening, and Sun are supported by the DSTL project "Test Coverage Metrics for Artificial Intelligence". Kwiatkowska and Ruan are supported by EPSRC Mobile Autonomy Programme Grant (EP/M019918/1). Wu is supported by the CSC-PAG Oxford Scholarship.

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09.
https://doi.org/10.1145/3238147.3238172

The software industry relies on testing as a primary means to provide stakeholders with information about the quality of the software product or service under test [12]. So far, there have been only few attempts to apply software testing techniques to DNNs [15, 18, 23, 25, 30]. These are either based on concrete execution, e.g., Monte Carlo tree search [30] and gradient-based search [15, 18, 25], or symbolic execution in combination with solvers for linear arithmetic [23]. Together with these test-input generation algorithms, several test coverage criteria have been presented, including neuron coverage [18], a criterion that is inspired by MC/DC [23], and criteria to capture particular neuron activation values to identify corner cases [15]. None of these approaches implement *concolic testing* [8, 22], which combines concrete execution and symbolic analysis to explore the execution paths of a program that are hard to cover by techniques such as random testing.

We hypothesise that concolic testing is particularly well-suited for DNNs. The input space of a DNN is usually high dimensional, which makes random testing difficult. For instance, a DNN for image classification takes tens of thousands of pixels as input. Moreover, owing to the widespread use of the ReLU activation function for hidden neurons, the number of "execution paths" in a DNN is simply too large to be completely covered by symbolic execution. Concolic testing can mitigate this complexity by directing the symbolic analysis to particular execution paths, through concretely evaluating given properties of the DNN.

In this paper, we present the first concolic testing method for DNNs. The method is parameterised using test goals, which we express using Quantified Linear Arithmetic over Rationals (QLAR). For a given set $\mathfrak{R}$ of test goals, we gradually generate test cases to improve coverage by alternating between concrete execution and symbolic analysis. Given an unsatisfied test requirement r, it is transformed into its corresponding form $\delta(r)$ by means of a heuristic function δ. Then, for the current set $\mathcal{T}$ of test cases, we identify a pair (t, r) of test case $t \in \mathcal{T}$ and requirement r such that t is close to satisfying r according to an evaluation based on *concrete execution*. After that, *symbolic analysis* is applied to (t, r) to obtain a new concrete test case t' that satisfies r. The test case t' is then added to the existing test suite, i.e., $\mathcal{T} = \mathcal{T} \cup \{t'\}$. This process is iterated until we reach a satisfactory level of coverage.

Finally, the generated test suite $\mathcal{T}$ is passed to a *robustness oracle*, which determines whether $\mathcal{T}$ includes *adversarial examples*, i.e., test cases that have different classification labels when close to each other with respect to a distance metric. The lack of robustness has been viewed as a major weakness of DNNs, and the discovery of adversarial examples [24] and the robustness problem are studied

actively in several domains, including machine learning, automated verification, cyber security, and software testing.

Overall, the main contributions of this paper are threefold:

(1) We develop the first concolic testing method for DNNs.
(2) We evaluate the method with a broad range of test requirements, including Lipschitz continuity [1, 3, 20, 29, 30] and several coverage metrics [15, 18, 23]. We show experimentally that the new algorithm supports this broad range of properties in a coherent way.
(3) We implement the concolic testing method in the software tool *DeepConcolic*[1]. Experimental results show that DeepConcolic achieves high coverage on the test requirements and that it is able to discover a significant number of adversarial examples.

2 RELATED WORK

We briefly review existing efforts for assessing the robustness of DNNs and the state of the art in concolic testing.

2.1 Robustness of DNNs

Current work on the robustness of DNNs can be categorised as offensive or defensive. Offensive approaches focus on heuristic search algorithms (mainly guided by the forward gradient or cost gradient of the DNN) to find adversarial examples that are as close as possible to a correctly classified input. On the other hand, the goal of defensive work is to increase the robustness of DNNs. There is an arms race between offensive and defensive techniques.

In this paper we focus on defensive methods. A promising approach is automated verification, which aims to provide robustness guarantees for DNNs. The main relevant techniques include a layer-by-layer exhaustive search [11], methods that use constraint solvers [14], global optimisation approaches [20] and abstract interpretation [7, 16] to over-approximate a DNN's behavior. Exhaustive search suffers from the state-space explosion problem, which can be alleviated by Monte Carlo tree search [30]. Constraint-based approaches are limited to small DNNs with hundreds of neurons. Global optimisation improves over constraint-based approaches through its ability to work with large DNNs, but its capacity is sensitive to the number of input dimensions that need to be perturbed. The results of over-approximating analyses can be pessimistic because of false alarms.

The application of traditional testing techniques to DNNs is difficult, and work that attempts to do so is more recent, e.g., [15, 18, 23, 25, 30]. Methods inspired by software testing methodologies typically employ coverage criteria to guide the generation of test cases; the resulting test suite is then searched for adversarial examples by querying an oracle. The coverage criteria considered include *neuron coverage* [18], which resembles traditional statement coverage. A set of criteria inspired by MD/DC coverage [10] is used in [23]; Ma et al. [15] present criteria that are designed to capture particular values of neuron activations. Tian et al. [25] study the utility of neuron coverage for detecting adversarial examples in DNNs for the Udacity-Didi Self-Driving Car Challenge.

We now discuss algorithms for test input generation. Wicker et al. [30] aim to cover the input space by exhaustive mutation testing that has theoretical guarantees, while in [15, 18, 25] gradient-based search algorithms are applied to solve optimisation problems, and Sun et al. [23] apply linear programming. None of these considers concolic testing and a general formalism for describing test requirements as we do in this paper.

[1] https://github.com/TrustAI/DeepConcolic

2.2 Concolic Testing

By concretely executing the program with particular inputs, which includes random testing, a large number of inputs can often be tested easily. However, without guidance, the generated test cases may be restricted to a subset of the execution paths of the program and the probability of exploring execution paths that contain bugs can be extremely low. In symbolic execution [5, 26, 32] an execution path is encoded symbolically. Modern constraint solvers can determine feasibility of the encoding effectively, although performance still degrages as the size of the symbolic representation increases. Concolic testing [8, 22] is an effective approach to automated test input generation. It is a hybrid software technique that alternates between concrete execution, i.e., testing on particular inputs, and symbolic execution, a classical technique that treats program variables as symbolic ones [13].

Concolic testing has been applied routinely in software testing, and a wide range of tools is available, e.g., [4, 8, 22]. It starts by executing the program with a concrete input. At the end of the concrete run, another execution path must be selected heuristically. This new execution path is then encoded symbolically and the resulting formula is solved by a constraint solver, to yield a new concrete input. The concrete execution and the symbolic analysis interleave until a certain level of structural coverage is reached.

The key factor that affects the performance of concolic testing is the heuristics used to select the next execution path. While there are simple approaches such as random search and depth-first search, more carefully designed heuristics can achieve better coverage [4, 9]. Automated generation of search heuristics is an active area of research [6, 27].

3 DEEP NEURAL NETWORKS

A (feedforward and deep) neural network, or DNN, is a tuple $\mathcal{N} = (L, T, \Phi)$ such that $L = \{L_k | k \in \{1, \ldots, K\}\}$ is a set of layers, $T \subseteq L \times L$ is a set of connections between layers, and $\Phi = \{\phi_k | k \in \{2, \ldots, K\}\}$ is a set of *activation functions*. Each layer L_k consists of s_k *neurons*, and the l-th neuron of layer k is denoted by $n_{k,l}$. We use $v_{k,l}$ to denote the value of $n_{k,l}$. Values of neurons in hidden layers (with $1 < k < K$) need to pass through a Rectified Linear Unit (ReLU) [17]. For convenience, we explicitly denote the activation value before the ReLU as $u_{k,l}$ such that

$$v_{k,l} = ReLU(u_{k,l}) = \begin{cases} u_{k,l} & \text{if } u_{k,l} \geq 0 \\ 0 & \text{otherwise} \end{cases} \quad (1)$$

ReLU is the most popular activation function for neural networks.

Except for inputs, every neuron is connected to neurons in the preceding layer by pre-defined weights such that $\forall 1 < k \leq K, \forall 1 \leq l \leq s_k$,

$$u_{k,l} = \sum_{1 \leq h \leq s_{k-1}} \{w_{k-1,h,l} \cdot v_{k-1,h}\} + b_{k,l} \quad (2)$$

where $w_{k-1,h,l}$ is the pre-trained weight for the connection between $n_{k-1,h}$ (i.e., the h-th neuron of layer $k-1$) and $n_{k,l}$ (i.e., the l-th neuron of layer k), and $b_{k,l}$ is the *bias*.

Finally, for any input, the neural network assigns a *label*, that is, the index of the neuron of the output layer having the largest value i.e., $label = \text{argmax}_{1 \le l \le s_K} \{v_{K,l}\}$.

Due to the existence of ReLU, the neural network is a highly non-linear function that approximates, e.g., the human perception ability. In this paper, we use variable x to range over all possible inputs in the input domain D_{L_1} and use $t, t_1, t_2, ...$ to denote concrete inputs. Given a particular input t, we say that the DNN $\mathcal{N}$ is instantiated and we use $\mathcal{N}[t]$ to denote this instance of the network.

- Given a network instance $\mathcal{N}[t]$, the activation values of each neuron $n_{k,l}$ of the network before and after ReLU are denoted as $u[t]_{k,l}$ and $v[t]_{k,l}$ respectively, and the final classification label is $label[t]$. We write $u[t]_k$ and $v[t]_k$ for $1 \le k \le s_k$ to denote the vectors of activations for neurons in layer k.
- When the input is given, the activation or deactivation of each ReLU operator in the DNN is determined.

We remark that, while for simplicity the definition focuses on DNNs with fully connected and convolutional layers, as shown in the experiments (Section 11) our method also applies to other popular layers, e.g., maxpooling, used in state-of-the-art DNNs.

4 FORMALIZING COVERAGE FOR DNNS

4.1 Activation Patterns

A software program has a set of concrete execution paths. Similarly, a DNN has a set of linear behaviours called *activation patterns* [23].

Definition 4.1 (Activation Pattern). Given a network $\mathcal{N}$ and an input t, the activation pattern of $\mathcal{N}[t]$ is a function $ap[\mathcal{N}, t]$ that maps the set of hidden neurons to $\{\mathsf{true}, \mathsf{false}\}$. We may write $ap[t]$ for $ap[\mathcal{N}, t]$ if $\mathcal{N}$ is clear from the context. For an activation pattern $ap[t]$, we use $ap[t]_{k,i}$ to denote whether the ReLU operator of the neuron $n_{k,i}$ is activated or not. Formally,

$$\begin{array}{lcl} ap[t]_{k,l} = \mathsf{false} & \equiv & u[t]_{k,l} < v[t]_{k,l} \\ ap[t]_{k,l} = \mathsf{true} & \equiv & u[t]_{k,l} = v[t]_{k,l} \end{array} \tag{3}$$

Intuitively, $ap[t]_{k,l} = \mathsf{true}$ if the ReLU of the neuron $n_{k,l}$ is activated, and $ap[t]_{k,l} = \mathsf{false}$ otherwise.

Given a DNN instance $\mathcal{N}[t]$, each ReLU operator's behaviour (i.e., each $ap[t]_{k,l}$) is fixed and this results in the particular activation pattern $ap[t]$, which can be encoded by using a Linear Programming (LP) model [23].

Computing a test suite that covers all activation patterns of a DNN is intractable owing to the large number of neurons in pratically-relevant DNNs. Therefore, we identify a subset of the activation patterns according to certain cover criteria, and then generate test cases that cover these activation patterns.

4.2 Quantified Linear Arithmetic over Rationals

We use a specific fragment of Quantified Linear Arithmetic over Rationals (QLAR) to express the requirements on the test suite for a given DNN. This enables us to give a single test generation algorithm (Section 8) for a variety of coverage criteria. We denote the set of formulas in our fragment by DR.

Definition 4.2. Given a network $\mathcal{N}$, we write $IV = \{x, x_1, x_2, ...\}$ for a set of variables that range over the inputs D_{L_1} of the network. We define $V = \{u[x]_{k,l}, v[x]_{k,l} \mid 1 \le k \le K, 1 \le l \le s_k, x \in IV\}$ to be a set of variables that range over the rationals. We fix the following syntax for DR formulas:

$$\begin{array}{lcl} r & ::= & Qx.e \mid Qx_1, x_2.e \\ e & ::= & a \bowtie 0 \mid e \wedge e \mid \neg e \mid |\{e_1, ..., e_m\}| \bowtie q \\ a & ::= & w \mid c \cdot w \mid p \mid a + a \mid a - a \end{array} \tag{4}$$

where $Q \in \{\exists, \forall\}$, $w \in V$, $c, p \in \mathbb{R}$, $q \in \mathbb{N}$, $\bowtie \in \{\le, <, =, >, \ge\}$, and $x, x_1, x_2 \in IV$. We may call r a requirement formula, e a Boolean formula, and a an arithmetic formula. We call the logic DR^+ if the negation operator $\neg$ is not allowed. *We use $\Re$ to denote a set of requirement formulas.*

The formula $\exists x.r$ expresses that there exists an input x such that r is true, while $\forall x.r$ expresses that r is true for all inputs x. The formulas $\exists x_1, x_2.r$ and $\forall x_1, x_2.r$ have similar meaning, except that they quantify over two inputs x_1 and x_2. The Boolean expression $|\{e_1, ..., e_m\}| \bowtie q$ is true if the number of true Boolean expressions in the set $\{e_1, ..., e_m\}$ is in relation $\bowtie$ with q. The other operators in Boolean and arithmetic formulas have their standard meaning.

Although V does not include variables to specify an activation pattern $ap[x]$, we may write

$$ap[x_1]_{k,l} = ap[x_2]_{k,l} \text{ and } ap[x_1]_{k,l} \neq ap[x_2]_{k,l} \tag{5}$$

to require that x_1 and x_2 have, respectively, the same and different activation behaviours on neuron $n_{k,l}$. These conditions can be expressed using the syntax above and the expressions in Equation (3). Moreover, some norm-based distances between two inputs can be expressed using our syntax. For example, we can use the set of constraints

$$\{x_1(i) - x_2(i) \le q,\ x_2(i) - x_1(i) \le q \mid i \in \{1, \ldots, s_1\}\} \tag{6}$$

to express $||x_1 - x_2||_\infty \le q$, i.e., we can constrain the Chebyshev distance L_∞ between two inputs x_1 and x_2, where $x(i)$ is the i-th dimension of the input x.

4.3 Semantics

We define the satisfiability of a requirement r over a test suite $\mathcal{T}$ containing a finite set of inputs.

Definition 4.3. Given a set $\mathcal{T}$ of test cases and a requirement r, the satisfiability relation $\mathcal{T} \models r$ is defined as follows.

- $\mathcal{T} \models \exists x.e$ if there exists some $t \in \mathcal{T}$ such that $\mathcal{T} \models e[x \mapsto t]$, where $e[x \mapsto t]$ is to substitute the occurrences of x with t.
- $\mathcal{T} \models \exists x_1, x_2.e$ if there exist two inputs $t_1, t_2 \in \mathcal{T}$ such that $\mathcal{T} \models e[x_1 \mapsto t_1][x_2 \mapsto t_2]$

The cases for $\forall$ formulas are similar to those for $\exists$ in the standard way. For the evaluation of Boolean expression e over an input t, we have

- $\mathcal{T} \models a \bowtie 0$ if $a \bowtie 0$
- $\mathcal{T} \models e_1 \wedge e_2$ if $\mathcal{T} \models e_1$ and $\mathcal{T} \models e_2$
- $\mathcal{T} \models \neg e$ if not $\mathcal{T} \models e$
- $\mathcal{T} \models |\{e_1, ..., e_m\}| \bowtie q$ if $|\{e_i \mid \mathcal{T} \models e_i, i \in \{1, ..., m\}\}| \bowtie q$

For the evaluation of arithmetic expression a over an input t,

- $u[t]_{k,l}$ and $v[t]_{k,l}$ have their values from the activations of the DNN, $c * u[t]_{k,l}$ and $c * v[t]_{k,l}$ have the standard meaning for c being the coefficient,
- p, $a_1 + a_2$, and $a_1 - a_2$ have the standard semantics.

Similarly to Definition 4.3, we can define the semantics based on a satisfiability relation $X \models r$ where $X \subseteq D_{L_1}$ is a (maybe continuous) input subspace. Note that, while $\mathcal{T}$ is finite, X may contain an infinite number of inputs. The relation $X \models r$ largely follows that of $\mathcal{T} \models r$ by replacing $\mathcal{T}$ with X. We have the following proposition.

PROPOSITION 4.4. *Given a DR$^+$ requirement r, a test suite $\mathcal{T}$ and a subspace $X \subseteq D_{L_1}$, if all test cases in $\mathcal{T}$ are also in X, we have that $X \models r$ implies $\mathcal{T} \models r$ but not vice versa.*

4.4 Test Criteria

Now we can define test criteria with respect to a set of test requirements and a set of test cases.

Definition 4.5 (Test Criterion). Given a network $\mathcal{N}$, a set $\mathfrak{R}$ of test requirements expressed as DR formulas, and a test suite $\mathcal{T}$, the test criterion $M(\mathfrak{R}, \mathcal{T})$ is as follows:

$$M(\mathfrak{R}, \mathcal{T}) = \frac{|\{r \in \mathfrak{R} \mid \mathcal{T} \models r\}|}{|\mathfrak{R}|} \tag{7}$$

Intuitively, it computes the percentage of the test requirements that are satisfied by test cases in $\mathcal{T}$. Similarly, we may define $M(\mathfrak{R}, X)$, called the true test criterion over X, for the consideration of the test requirement $\mathfrak{R}$ over all possible inputs in X. For $\mathcal{T} \subseteq X \subseteq D_{L_1}$, we have that

$$M(\mathfrak{R}, \mathcal{T}) \leq M(\mathfrak{R}, X) \leq M(\mathfrak{R}, D_{L_1}) \leq 1.0 \tag{8}$$

when all requirements in $\mathfrak{R}$ are DR$^+$ formulas.

4.5 Computational Complexity

We study the computational complexity of the test requirements satisfaction problem. For the testing, it is in polynomial time with respect to the number of test cases in the test suite.

THEOREM 4.6. *Given a network $\mathcal{N}$, a DR requirement formula r, and a test suite $\mathcal{T}$, the checking of $\mathcal{T} \models r$ can be done in polynomial time with respect to the size of $\mathcal{T}$.*

However, the general verification problem is NP-complete with respect to the number of hidden neurons.

THEOREM 4.7. *Given a network $\mathcal{N}$, a DR requirement formula r and a subspace $X \subseteq D_{L_1}$, the checking of $X \models r$ is NP-complete. This conclusion also holds for DR$^+$ requirements.*

5 CONCRETE REQUIREMENTS

In this section, we use DR$^+$ formulas to express several important requirements for DNNs, including Lipschitz continuity [1, 3, 20, 29, 30] and test criteria [15, 18, 23]. The test criteria we consider have syntactical similarity with structural test coverage metrics in software testing. Lipschitz continuity is semantic, specific to DNNs, and shown to be closely related to the theoretical understanding of convolutional DNNs [29] and the robustness of both DNNs [20, 30] and Generative Adversarial Networks [1]. These requirements have been studied in the literature using different formalisms and approaches.

Each test coverage criterion gives rise to a set of test requirements. The degree to which these requirements are satisfied is used as a metric for the confidence in the safety of the DNN under test. In the following, we discuss the three test criteria from [15, 18, 23], respectively. We use $||t_1 - t_2||_q$ to denote the distance between two inputs t_1 and t_2 with respect to a distance metric $||\cdot||_q$. The metric $||\cdot||_q$ can be, e.g., a norm-based metric such as the L_0-norm (Hamming distance), L_2-norm (Euclidean distance), and L_∞-norm (Chebyshev distance), or a structural similarity distance, such as SSIM [28]. In the following, we fix a distance metric and simply write $||t_1 - t_2||$. Section 11 will give the metrics we use for the experiments.

We may consider requirements for a set of input subspaces. Given a real number b, we can generate a finite set $\mathcal{S}(D_{L_1}, b)$ of subspaces of D_{L_1} such that for all inputs $x_1, x_2 \in D_{L_1}$, if $||x_1 - x_2|| \leq b$ then there exists a subspace $X \in \mathcal{S}(D_{L_1}, b)$ such that $x_1, x_2 \in X$. The subspaces can be overlapping. Usually, every subspace $X \in \mathcal{S}(D_{L_1}, b)$ can be represented with a box constraint, e.g., $X = [l, u]^{s_1}$, and therefore $t \in X$ can be expressed with a Boolean expression as follows.

$$\bigwedge_{i=1}^{s_1} x(i) - u \leq 0 \wedge x(i) - l \geq 0 \tag{9}$$

5.1 Lipschitz Continuity

Lipschitz continuity has been shown in [20, 24] to hold for a large class of DNNs, including, e.g., image classification DNNs.

Definition 5.1 (Lipschitz Continuity). A network $\mathcal{N}$ is regarded *Lipschitz continuous* if there exists a real constant $c \geq 0$ such that, for all $x_1, x_2 \in D_{L_1}$:

$$||v[x_1]_1 - v[x_2]_1|| \leq c * ||x_1 - x_2|| \tag{10}$$

The value c is called the *Lipschitz constant*, and the smallest c is called the *best Lipschitz constant*, denoted as c_{best}. Recall that $v[x]_1$ denotes the vector of activations for neurons at the input layer.

Since the computation of c_{best} is an NP-hard problem and a smaller c can significantly improve the performance of verification algorithms [20, 30, 31], it is interesting to know whether a given number c is a legitimate Lipschitz constant, either for the entire input space D_{L_1} or for some subspace $X \in \mathcal{S}(D_{L_1}, b)$. The testing of Lipschitz continuity can be guided by having the following requirements.

Definition 5.2 (Lipschitz Requirements). Given a real $c > 0$ and an integer $b > 0$, the set $\mathfrak{R}_{Lip}(b, c)$ of Lipschitz requirements is

$$\begin{aligned}\{\exists x_1, x_2.(||v[x_1]_1 - v[x_2]_1|| - c * ||x_1 - x_2|| > 0) \\ \wedge x_1, x_2 \in X \mid X \in \mathcal{S}(D_{L_1}, b)\}\end{aligned} \tag{11}$$

Intuitively, for each $X \in \mathcal{S}(D_{L_1}, b)$, this requirement expresses the existence of two inputs x_1 and x_2 such that $\mathcal{N}$ breaks the Lipschitz constant c. Given a number c, the true test criteria $M(\mathfrak{R}_{Lip}(b, c), D_{L_1})$ may be impossible to satisfy fully, because there may exist $r \in \mathfrak{R}_{Lip}(b, c)$ such that $D_{L_1} \not\models r$. Thus, the goal for a test case generation algorithm is to produce $\mathcal{T}$ that satisfy the criteria as much as possible.

5.2 Neuron Coverage

Neuron Coverage (NC) [18] is an adaptation of statement coverage in conventional software testing. It is defined as follows.

Definition 5.3. Neuron coverage for a DNN $\mathcal{N}$ requires a test suite $\mathcal{T}$ such that, for any hidden neuron $n_{k,i}$, there exists test case $t \in \mathcal{T}$ such that $ap[t]_{k,i} = \mathsf{true}$.

This can be expressed with the following requirements in $\mathfrak{R}_{NC}$, each of which expresses that there is an input x that activates the neuron $n_{k,i}$, i.e., $ap[x]_{k,i} = \mathsf{true}$.

Definition 5.4 (NC Requirements). The set $\mathfrak{R}_{NC}$ of requirements is

$$\{\exists x. ap[x]_{k,i} = \mathsf{true} \mid 2 \le k \le K-1, 1 \le i \le s_k\} \tag{12}$$

5.3 Modified Condition/Decision Coverage (MC/DC)

In [23], a family of four test criteria are proposed, inspired by MC/DC coverage in conventional software testing. Here, we work with the Sign-Sign Coverage (SSC). According to [23], each neuron $n_{k+1,j}$ can be seen as a decision such that these neurons in the previous layer (i.e., the k-th layer) are conditions that define its activation value, as in Equation (2). Adapting MC/DC to DNNs, it means that each condition neuron must be shown to independently affect the outcome of the decision neuron. In particular, the SSC observes the change of a decision or condition neuron, if the sign of its activation, which is either positive or negative, changes.

Consequently, the test requirements for SSC are defined by the following set.

Definition 5.5 (SSC Requirements). Given a pair $\alpha = (n_{k,i}, n_{k+1,j})$ of neurons, the singleton set $\mathfrak{R}_{SSC}(\alpha)$ of requirements is as follows:

$$\{\exists x_1, x_2.\ ap[x_1]_{k,i} \neq ap[x_2]_{k,i} \wedge ap[x_1]_{k+1,j} \neq ap[x_2]_{k+1,j} \wedge \bigwedge_{1 \le l \le s_k, l \neq i} ap[x_1]_{k,l} - ap[x_2]_{k,l} = 0\} \tag{13}$$

and we have

$$\mathfrak{R}_{SSC} = \bigcup_{2 \le k \le K-2, 1 \le i \le s_k, 1 \le j \le s_{k+1}} \mathfrak{R}_{SSC}((n_{k,i}, n_{k+1,j})) \tag{14}$$

That is, for each pair $(n_{k,i}, n_{k+1,j})$ of neurons at two adjacent layers k and $k+1$ respectively, we need two inputs x_1 and x_2 such that the sign change of $n_{k,i}$ independently affects the sign change of $n_{k+1,j}$. Other neurons at layer k are required to maintain their signs between x_1 and x_2 to ensure the independent affection. The idea of SS Cover (and all other test criteria in [23]) is to ensure that not only the presence of a feature needs to be tested but also the effects of less complex features on a more complex feature must be tested.

5.4 Neuron Boundary Cover

The Neuron Boundary Cover (NBC) [15] aims to cover neuron activation values that exceed pre-specified bounds. It can be formulated as follows.

Definition 5.6 (Neuron Boundary Cover Requirements). Given two sets of bounds $h = \{h_{k,i} | 2 \le k \le K-1, 1 \le i \le s_k\}$ and $l = \{l_{k,i} | 2 \le k \le K-1, 1 \le i \le s_k\}$, the set $\mathfrak{R}_{NBC}(h, l)$ of requirements is

$$\{\exists x.\ u[x]_{k,i} - h_{k,i} > 0,\ \ \exists x.\ u[x]_{k,i} - l_{k,i} < 0 \mid 2 \le k \le K-1, 1 \le i \le s_k\} \tag{15}$$

where $h_{k,i}$ and $l_{k,i}$ are the upper and lower bounds on the activation value of a neuron $n_{k,i}$.

6 OVERALL DESIGN

This section describes the overall design of the concolic testing approach for requirements expressed using our formalism. The method alternates between concretely evaluating a DNN's activation patterns and symbolically generating new inputs. The concolic testing pseudocode is in Algorithm 1 and the corresponding workflow is depicted in Figure 1.

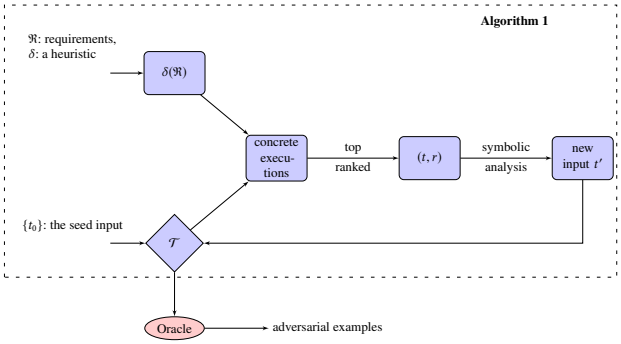


Figure 1: Overview of our concolic testing method.

Algorithm 1 takes as inputs a DNN $\mathcal{N}$, an input t_0, a heuristic δ, and a set $\mathfrak{R}$ of requirements, and produces a test suite $\mathcal{T}$. In the algorithm, t is the latest test case generated, and is initialised as the input t_0. For every test requirement $r \in \mathfrak{R}$, it is removed from $\mathfrak{R}$ whenever satisfied by $\mathcal{T}$, i.e., $\mathcal{T} \models r$.

Algorithm 1 Concolic Testing Algorithm for DNNs

INPUT: $\mathcal{N}, \mathfrak{R}, \delta, t_0$
OUTPUT: $\mathcal{T}$

```
1:  T ← {t0} and S = {}
2:  t ← t0
3:  while ℜ ≠ ∅ do
4:    for each r ∈ ℜ do
5:      if T ⊨ r then ℜ ← ℜ \ {r}
6:    while true do
7:      t, r ← requirement_evaluation(T, δ(ℜ))
8:      t′ ← symbolic_analysis(t, r)
9:      if soundness_check(t′) = true then
10:       T ← T ∪ {t′}
11:       break
12:     else
13:       S ← S ∪ {(t, r)}
14:       if S = T × ℜ then return T
15: return T
```

The function *requirement_evaluation* (Line 7), whose details are given in Section 7, aims to find a pair (t, r) [2] of input and requirement

[2]For some requirements, we might return two inputs t_1 and t_2. Here, for simplicity, we describe the case for a single input. The generalisation to two inputs is straightforward.

which, according to our concrete evaluation, are the most promising in finding a new test case t' to satisfy the requirement r. The heuristic δ is a transformation function mapping a quantified formula r with operator $\exists$ into an optimisation formula $\delta(r)$ with operator $\arg opt$. In the evaluation, concrete executions are applied.

After obtaining (t, r), the *symbolic_analysis* (Line 8), whose details are in Section 8, is applied to have a new concrete input t'. Then a function *soundness_check* (Line 9), whose details are given in Section 9, is applied to check if the new input is sound or not. The set S maintains a set of (t, r) pairs on which our symbolic analysis cannot find a sound new input.

The algorithm has two termination conditions. When all test requirements in $\mathfrak{R}$ have been satisfied, i.e., $\mathfrak{R} = \emptyset$, or no further requirement in $\mathfrak{R}$ can be satisfied, i.e., $S = \mathcal{T} \times \mathfrak{R}$, the algorithm terminates and returns the current test suite $\mathcal{T}$.

As shown in Figure 1, after the generation of test suite $\mathcal{T}$ by Algorithm 1, $\mathcal{T}$ will run through an oracle, i.e., *robustness_oracle* in Section 9, in order to evaluate the robustness of the DNN.

7 REQUIREMENT EVALUATION

This section presents our approach for Line 7 of Algorithm 1. Given a set of requirements $\mathfrak{R}$ that have not been satisfied, a heuristic δ, and the current set $\mathcal{T}$ of test cases, the goal is to select a concrete input $t \in \mathcal{T}$ together with a requirement $r \in \mathfrak{R}$, both of which will be used later in a symbolic approach to find the next concrete input t' (to be given in Section 8). The selection of t and r is done by concrete executions.

The general idea of obtaining (t, r) is as follows. For all requirements $r \in \mathfrak{R}$, we transform r into $\delta(r)$ by utilising operators $\arg opt$ for $opt \in \{\max, \min\}$ that will be evaluated by concretely executing tests in $\mathcal{T}$. As $\mathfrak{R}$ may contain more than one requirement, we return the pair (t, r) such that

$$r = \arg\max_{r}\{val(t, \delta(r)) \mid r \in \mathfrak{R}\}. \tag{16}$$

Note that, when evaluating $\arg opt$ formulas (e.g., $\arg\min_x a : e$), if an input $t \in \mathcal{T}$ is returned, we may need the value $(\min_x a : e)$ as well. We use $val(t, \delta(r))$ to denote such a value for the returned input t and the requirement formula r.

The formula $\delta(r)$ expresses an optimisation objective together with a set of constraints. We will give several examples later in Section 7.1. In the following, we extend the semantics in Definition 4.3 to work with formulas with $\arg opt$ operators for $opt \in \{\max, \min\}$, including $\arg opt_x a : e$ and $\arg opt_{x_1,x_2} a : e$. Intuitively, $\arg\max_x a : e$ ($\arg\min_x a : e$, resp.) is to find the input x among those satisfying Boolean formula e to maximise (minimise) the value of the arithmetic formula a. Formally,

- the evaluation of $\arg\min_x a : e$ on $\mathcal{T}$ returns an input $t \in \mathcal{T}$ such that, $\mathcal{T} \models e[x \mapsto t]$ and for all $t' \in \mathcal{T}$ such that $\mathcal{T} \models e[x \mapsto t']$ we have $a[x \mapsto t] \leq a[x \mapsto t']$.
- the evaluation of $\mathcal{T} \models \arg\min_{x_1,x_2} a : e$ on $\mathcal{T}$ returns two inputs $t_1, t_1 \in \mathcal{T}$ such that, $\mathcal{T} \models e[x_1 \mapsto t_1][x_2 \mapsto t_2]$ and for all $t'_1, t'_2 \in \mathcal{T}$ such that $\mathcal{T} \models e[x_1 \mapsto t'_1][x_2 \mapsto t'_2]$ we have $a[x_1 \mapsto t_1][x_2 \mapsto t_2] \leq a[x_1 \mapsto t'_1][x_2 \mapsto t'_2]$.

The cases for $\arg\max$ formulas are similar to those for $\arg\min$, by replacing $\leq$ with $\geq$. Similarly to Definition 4.3, the semantics is for a set $\mathcal{T}$ of test cases and we can adapt it to work with an input subspace $X \subseteq D_{L_1}$. We remark that in concrete execution the evaluation is on $\mathcal{T}$.

7.1 Heuristics

For the several requirements r discussed in Section 5, we present the formula $\delta(r)$ used in this paper. We remark that, since δ is a heuristic, there exist other definitions. The following definitions work well in our experiments.

7.1.1 Lipschitz Continuity. When a Lipschitz requirement r as in Equation (11) is unsatisfiable on $\mathcal{T}$, we transform it into $\delta(r)$ as follows:

$$\arg\max_{x_1,x_2} .||v[x_1]_1 - v[x_2]_1|| - c * ||x_1 - x_2|| : x_1, x_2 \in X \tag{17}$$

According to the semantics in Definition 4.3, the aim is to find the best t_1 and t_2 from $\mathcal{T}$ to make the evaluation of $||v[t_1]_1 - v[t_2]_1|| - c * ||t_1 - t_2||$ as large as possible. As described, we also need to compute $val(t_1, t_2, r) = ||v[t_1]_1 - v[t_2]_1|| - c * ||t_1 - t_2||$.

7.1.2 Neuron Cover. When a requirement r in Equation (12) is unsatisfiable on $\mathcal{T}$, we transform it into the following requirement $\delta(r)$:

$$\arg\max_{x} c_k \cdot u_{k,i}[x] : \mathsf{true} \tag{18}$$

According to the semantics, the requirement will return the input $t \in \mathcal{T}$ that has the maximal value for $c_k \cdot u_{k,i}[x]$.

The coefficient c_k is a layer-wise constant. It is based on the following observation. With the propagation of signals in the DNN, activation values at each layer can be of different magnitudes. For example, if the minimum activation value of neurons at layer k and $k+1$ are -10 and -100 respectively, then even when a neuron $u[x]_{k,i} = -1 > -2 = u[x]_{k+1,j}$, we may still regard $n_{k+1,j}$ as being closer to be activated than $u_{k,i}$ is. Consequently, we define a layer factor c_k for each layer which normalises the average activation valuations of neurons at different layers into the same magnitude level. It is estimated by sampling a large enough input dataset.

7.1.3 SS Cover. In the SS Cover, given a decision neuron $n_{k+1,j}$, the concrete evaluation aims to select one of its condition neurons $n_{k,i}$ at layer k such that, for the test case to be generated, the signs of $n_{k,i}$ and $n_{k+1,j}$ can be negated and the rest of $n_{k+1,j}$'s condition neurons reserve their respective signs. This is achieved by having the following $\delta(r)$:

$$\arg\max_{x} -c_k \cdot |u[x]_{k,i}| : \mathsf{true} \tag{19}$$

Intuitively, given the decision neuron $n_{k+1,j}$, Equation (19) selects the condition that is closest to the change of activation sign (i.e., smallest $|u[x]_{k,i}|$).

7.1.4 Neuron Boundary Cover. We transform the requirement r in Equation (20) into the following $\delta(r)$ when it is not satisfiable on $\mathcal{T}$; it selects the neuron that is closest to either the higher or lower boundary.

$$\begin{array}{l} \arg\max_x c_k \cdot (u[x]_{k,i} - h_{k,i}) : \mathsf{true} \\ \arg\max_x c_k \cdot (l_{k,i} - u[x]_{k,i}) : \mathsf{true} \end{array} \tag{20}$$

8 SYMBOLIC GENERATION OF NEW CONCRETE INPUTS

This section presents our approach for Line 8 of Algorithm 1. That is, given a concrete input t and a requirement r, we need to find the next concrete input t' by symbolic analysis. This new t' will be added into the test suite, i.e., Line 10 of Algorithm 1. The symbolic analysis techniques to be considered include the linear programming method in [23], a global optimisation method for the L_0 norm in [21], and a new optimisation algorithm to be introduced below. We regard optimisation algorithms as symbolic analysis methods because, similarly to constraint solving methods, they work with a set of test cases in a single run.

Thanks to the use of *DR*, for each symbolic analysis method, its application to different test criteria can be formulated under a unified logic framework. To ease the presentation, the following description may, for each algorithm, focus on a few requirements, but we remark that all algorithms can work with all the requirements given in Section 5.

8.1 Symbolic Analysis using Linear Programming

As explained in Section 4, given an input x, the DNN instance $\mathcal{N}[x]$ maps to an activation pattern $ap[x]$ that can be modeled using Linear Programming (LP). In particular, the following linear constraints [23] yield a set of inputs that exhibit the same ReLU behaviour as x:

$$\{\mathbf{u_{k,i}} = \sum_{1\le j\le s_{k-1}} \{w_{k-1,j,i}\cdot \mathbf{v_{k-1,j}}\} + b_{k,i} \mid k\in[2,K], i\in[1..s_k]\} \quad (21)$$

$$\begin{array}{l}\{\mathbf{u_{k,i}} \ge 0 \wedge \mathbf{u_{k,i}} = \mathbf{v_{k,i}} \mid ap[x]_{k,i} = \mathsf{true}, k\in[2,K), i\in[1..s_k]\}\\ \cup\{\mathbf{u_{k,i}} < 0 \wedge \mathbf{v_{k,i}} = 0 \mid ap[x]_{k,i} = \mathsf{false}, k\in[2,K), i\in[1..s_k]\}\end{array} \quad (22)$$

Real valued variables in the LP model are emphasized in **bold**.

- The activation value of each neuron is encoded by the linear constraint in (21), which is a symbolic version of the Equation (2) that calculates a neuron's activation.
- Given a particular input x, the activation pattern (Definition 4.1) $ap[x]$ is known, with $ap[x]_{k,i}$ being either true or false that represents the ReLU's activation or not for the neuron $n_{k,i}$. Following the definition of ReLU in (1), for every neuron $n_{k,i}$, the linear constraints in (22) encode its ReLU's activation (when $ap[x]_{k,i} = \mathsf{true}$) or deactivation (when $ap[x]_{k,i} = \mathsf{false}$).

The linear model (denoted as C for generic purposes) given by (21) and (22) represents an input set that results in the same activation pattern as encoded. Consequently, the symbolic analysis for finding a new input t' from a pair (t, r) of input and requirement is equivalent to finding a new activation pattern. *Note that, to make sure that the obtained test case is meaningful, in the LP model an objective is added to minimize the distance between t and t′.* Thus, the use of LP requires that a distance metric be linear, e.g., L_∞-norm in (6).

8.1.1 Neuron Coverage. The symbolic analysis of neuron coverage takes the input test case t and requirement r, let us say, on the activation of neuron $n_{k,i}$, and it shall return a new test t' such that the test requirement is satisfied by the network instance $\mathcal{N}[t']$. As a result, given $\mathcal{N}[t]$'s activation pattern $ap[t]$, we can build up a new activation pattern ap' such that

$$\{ap'_{k,i} = \neg ap[t]_{k,i} \wedge \forall k_1 < k : \bigwedge_{0\le i_1\le s_{k_1}} ap'_{k_1,i_1} = ap[t]_{k_1,i_1}\} \quad (23)$$

This activation pattern specifies the following conditions.

- $n_{k,i}$'s activation sign is negated: this ensures the aim of the symbolic analysis to activate $n_{k,i}$.
- In the new activation pattern ap', the neurons before layer k preserve their activation signs as in $ap[t]$. Though there may exist various activation patterns that make $n_{k,i}$ activated, for the use of LP modeling one particular combination of activation signs must be pre-determined.
- Other neurons are irrelevant, as the sign of $n_{k,i}$ is only affected by the activation values of those neurons in previous layers.

Finally, the new activation pattern ap' defined in (23) is encoded by the LP model C using (21) and (22), and if there exists a feasible solution, then it will become the new test case t', which makes the DNN satisfy the requirement r.

8.1.2 SS Coverage. When it comes to SS Coverage, to satisfy the requirement r we need to find a new test case such that, with respect to the input t, activation signs of $n_{k+1,j}$ and $n_{k,i}$ are negated, while other signs of other neurons at layer k are kept the same as in the case of input t. To achieve this, the following activation pattern ap' is built up for the LP modeling.

$$\begin{array}{l}\{ap'_{k,i} = \neg ap[t]_{k,i} \wedge ap'_{k+1,j} = \neg ap[t]_{k+1,j}\\ \wedge\forall k_1 < k : \bigwedge_{1\le i_1\le s_{k_1}} ap'_{k_1,i_1} = ap[t]_{k_1,i_1}\}\end{array}$$

8.1.3 Neuron Boundary Coverage. In case of the neuron boundary coverage, the symbolic analysis aims to find an input t' such that the neuron $n_{k,i}$'s activation value exceeds either its higher bound $h_{k,i}$ or its lower bound $l_{k,i}$. To achieve this, while preserving the DNN activation pattern as $ap[t]$, we add one of the following constraints into the LP program.

- If $u[x]_{k,i} - h_{k,i} > l_{k,i} - u[x]_{k,i}$: $\mathbf{u_{k,i}} > h_{k,i}$;
- otherwise: $\mathbf{u_{k,i}} < l_{k,i}$.

8.2 Symbolic Analysis using Global Optimisation

The symbolic analysis for finding a new input can also be implemented by solving the global optimisation problem in [21]. That is, by specifying the test requirement as an optimisation objective, we apply global optimisation to find a test case that makes the test requirement satisfied. Readers are referred to [21] for the details of the algorithm.

- For Neuron Coverage, the objective is thus to find a t' such that the specified neuron $n_{k,i}$ has $ap[t']_{k,i} = \mathsf{true}$.
- In case of SS Coverage, given the neuron pair $(n_{k,i}, n_{k+1,j})$ and the original input t, the optimisation objective becomes

$$\begin{array}{l}ap[t']_{k,i} \ne ap[t]_{k,i} \wedge ap[t']_{k+1,j} \ne\\ ap[t]_{k+1,j} \wedge \bigwedge_{i'\ne i} ap[t']_{k,i'} = ap[t]_{k,i}\end{array}$$

- Regarding the Neuron Boundary Coverage, depending on whether the higher bound or lower bound for the activation of $n_{k,i}$ is considered, the objective of finding a new input

t' can be one of the two forms: 1) $u[t']_{k,i} > h_{k,i}$ or 2) $u[t']_{k,i} < l_{k,i}$.

8.3 Lipschitz Test Case Generation

Given a requirement in Equation (11) for a subspace X, we let $t_0 \in \mathbb{R}^n$ be the representative point of the subspace X to which t_1 and t_2 belong. The optimisation problem is to generate two inputs t_1 and t_2 such that

$$\begin{aligned} ||v[t_1]_1 - v[t_2]_1||_{D_1} - c * ||t_1 - t_2||_{D_1} > 0 \\ \text{s.t. } ||t_1 - t_0||_{D_2} \leq \Delta,\ ||t_2 - t_0||_{D_2} \leq \Delta \end{aligned} \tag{24}$$

where $|| * ||_{D_1}$ and $|| * ||_{D_2}$ denote certain norm metrics such as the L_0-norm, L_2-norm or L_∞-norm, and Δ intuitively represents the radius of a norm ball (for L_1, L_2-norm) or the size of a hypercube (for L_∞-norm) centered on t_0. Δ is a hyper-parameter of the algorithm.

The above problem can be efficiently solved by a novel *alternating compass search* scheme. Specifically, we alternately optimise the following two optimisation problems through relaxation [19], i.e., maximizing the lower bound of the original Lipschitz constant instead of directly maximizing the Lipschitz constant itself. To do so we formulate the original non-linear proportional optimisation as a linear problem when both norm metrics $|| * ||_{D_1}$ and $|| * ||_{D_2}$ are L_∞-norm.

8.3.1 *Stage One.*

We solve

$$\begin{aligned} \min_{t_1} F(t_1, t_0) = -||v[t_1]_1 - v[t_0]_1||_{D_1} \\ \text{s.t. } ||t_1 - t_0||_{D_2} \leq \Delta \end{aligned} \tag{25}$$

The above objective enables the algorithm to search for an optimal t_1 in the space of a norm ball or hypercube centered on t_0 with radius Δ, such that the norm distance of $v[t_1]_1$ and $v[t_0]_1$ is as large as possible. From the constraint, we know that $\sup_{||t_1 - t_0||_{D_2} \leq \Delta} ||t_1 - t_0||_{D_2} = \Delta$. Thus a smaller $F(t_1, t_0)$ essentially leads to a larger Lipschitz constant, considering that $\mathbf{Lip}(t_1, t_0) = -F(t_1, t_0)/||t_1 - t_0||_{D_2} \geq -F(t_1, t_0)/\Delta$, i.e., $-F(t_1, t_0)/\Delta$ is the lower bound of $\mathbf{Lip}(t_1, t_0)$. Therefore, the searching trace of minimizing $F(t_1, t_0)$ will generally lead to an increase of the Lipschitz constant.

To solve the above the problem we use the compass search method [2], which is efficient, derivative-free, and guaranteed to have first-order global convergence. Because we aim to find an input pair to break the predefined Lipschitz constant c instead of finding the largest Lipschitz constant, along each iteration, when we get $\bar{t}_1$, we check whether $\mathbf{Lip}(\bar{t}_1, t_0) > c$. If it holds, we find an input pair $\bar{t}_1$ and t_0 that satisfies the test requirement; otherwise, we continue the compass search until convergence or a satisfiable input pair is generated. If Equation (25) is convergent and we can find an optimal t_1 as

$$t_1^* = \arg\min_{t_1} F(t_1, t_0) \text{ s.t. } ||t_1 - t_0||_{D_2} \leq \Delta$$

but we still cannot find a satisfiable input pair, we perform Stage Two optimisation.

8.3.2 *Stage Two.*

We solve

$$\begin{aligned} \min_{t_2} F(t_1^*, t_2) = -||v[t_2]_1 - v[t_1^*]_1||_{D_1} \\ \text{s.t. } ||t_2 - t_0||_{D_2} \leq \Delta \end{aligned} \tag{26}$$

Similarly, we use derivative-free compass search to solve the above problem and check whether $\mathbf{Lip}(t_1^*, t_2) > c$ holds at each iterative optimisation trace $\bar{t}_2$. If it holds, we return the image pair t_1^* and $\bar{t}_2$ that satisfies the test requirement; otherwise, we continue the optimisation until convergence or a satisfiable input pair is generated. If Equation (26) is convergent at t_2^*, and we still cannot find such a input pair, we modify the objective function again by letting $t_1^* = t_2^*$ in Equation (26) and continue the search and satisfiability checking procedure.

8.3.3 *Stage Three.*

If the function $\mathbf{Lip}(t_1^*, t_2^*)$ stops increasing in Stage Two, we treat the whole search procedure as convergent and fail to find an input pair that can break the predefined Lipschitz constant c. In this case, we return the best input pair we can find, i.e., t_1^* and t_2^*, and the largest Lipschitz constant $\mathbf{Lip}(t_1^*, t_2)$. Note that the returned constant is smaller than c.

In summary, the proposed method is an alternating optimisation scheme based on the compass search. Basically, we start from the given t_0 to search for an image t_1 in a norm ball or hypercube, where the optimisation trajectory on the norm ball space is denoted as $S(t_0, \Delta(t_0)))$ such that $Lip(t_0, t_1) > c$ (this step is symbolic execution); if we cannot find it, we modify the optimisation objective function by replacing t_0 with t_1^* (the best concrete input found in this optimisation trace) to initiate another optimisation trajectory on the space, i.e., $S(t_1^*, \Delta(t_0))$. This process is repeated until the optimisation trace gradually covers the whole norm ball space $S(\Delta(t_0))$.

9 ORACLE

First of all, we provide details to Line 10 of Algorithm 1 about the soundness checking.

Definition 9.1 (Soundness Checking). Given a set O of correctly classified inputs (e.g., the training dataset) and a real number b, a test case $t' \in \mathcal{T}$ passes the soundness checking if

$$\exists t \in O: \quad ||t - t'|| \leq b \tag{27}$$

Intuitively, it says that the test case t is sound if it is close to some of the correctly classified inputs in O. Given a test case $t' \in \mathcal{T}$, we can write $O(t')$ for the input $t \in O$ which has the smallest distance to t' among all inputs in O.

When checking the quality of the generated test suite $\mathcal{T}$, we use the following oracle.

Definition 9.2 (Robustness Oracle). Given a set O of correctly classified inputs, a test case t' passes the robustness oracle if

$$\arg\max_j v[t']_{K,j} = \arg\max_j v[O(t')]_{K,j} \tag{28}$$

Intuitively, the role of this oracle is to check the robustness of the DNN on input $O(t')$: if t' cannot pass the oracle then it serves as evidence of the DNN lacking in robustness.

10 A SUMMARY OF COVERAGE-BASED DNN TESTING

We briefly summarise the similarities and differences between our concolic testing method, named *DeepConcolic*, and other existing coverage-driven DNN testing methods: DeepXplore [18], DeepTest [25], DeepCover [23], and DeepGauge [15]. The details are presented in Table 1, where NC, SSC, and NBC are short for Neuron Cover, SS Cover, and Neuron Boundary Cover, respectively. In addition to the concolic nature of DeepConcolic, we observe the following differences.

Table 1: Comparison between different coverage-based DNN testing methods

	DeepConcolic	DeepXplore [18]	DeepTest [25]	DeepCover [23]	DeepGauge [15]
Coverage criteria	NC, SSC, NBC etc.	NC	NC	MC/DC	NBC etc.
Test generation	concolic	dual-optimisation	greedy search	symbolic execution	gradient descent methods
DNN inputs	single	multiple	single	single	single
Image inputs	single/multiple	multiple	multiple	multiple	multiple
Distance metric	L_∞, L_0-norm	L_1-norm	Jaccard distance	L_∞-norm	L_∞-norm

- DeepConcolic is generic, using a unified language DR to express test requirements and a small set of algorithms to compute a class of requirements; the other methods are *ad hoc* tests for specific requirements.
- Comparing with DeepXplore, which needs a set of DNNs to explore multiple gradient directions, the other methods, including DeepConcolic, need a single DNN only.
- In contrast to the other methods, DeepConcolic can achieve good coverage by starting from a single input; the other methods need a non-trivial set of inputs.
- Until now, there is no conclusion on the best distance metric. DeepConcolic can be parameterized with a desired norm distance metric $||\cdot||$.

Moreover, from the workflow given in Figure 1, we can see that DeepConcolic features a clean separation between the generation of test cases and the oracle. This is well aligned with the traditional approach to test case generation. The other methods essentially use the oracles of Section 9 as part of their objectives to guide the generation of test cases.

11 EXPERIMENTAL RESULTS

The concolic testing approach presented in this paper has been implemented in a software tool we have named DeepConcolic[3]. In this section, we compare it with the latest DNN testing tools and evaluate its performance for different test requirements. The experiments are run on a machine with 24 cores Intel(R) Xeon(R) CPU E5-2620 v3 @ 2.40GHz and 125G memory. When testing a DNN, if the DeepConcolic testing does not finish within 12 hours, it is forced to terminate. All coverage results are collected by running the testing repeatedly at least 10 times.

11.1 Comparison with DeepXplore

This section compares the use of DeepConcolic and DeepXplore [18] for two state-of-the-art DNNs on the MNIST and CIFAR-10 datasets, respectively. We remark that DeepXplore has been tested on further datasets.

For each tool, we start neuron cover testing from a randomly sampled image input. Note that, since DeepXplore requires more than one DNN, we designate our trained DNN as the target model and utilise the other two default models provided by DeepXplore. Table 2 gives the neuron cover reports from the two tools. We observe that DeepConcolic yields much higher neuron coverage than DeepXplore in any of its three modes of operation ('light', 'occlusion', and 'blackout'). On the other hand, DeepXplore is much faster and terminates in seconds.

Table 2: Neuron coverage of DeepConcolic and DeepXplore

	DeepConcolic		DeepXplore		
	L_∞-norm	L_0-norm	light	occlusion	blackout
MNIST	97.60%	95.91%	80.77%	82.68%	81.61%
CIFAR-10	84.98%	98.63%	77.56%	81.48%	83.25%

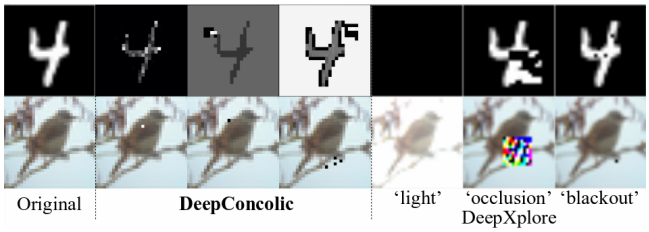
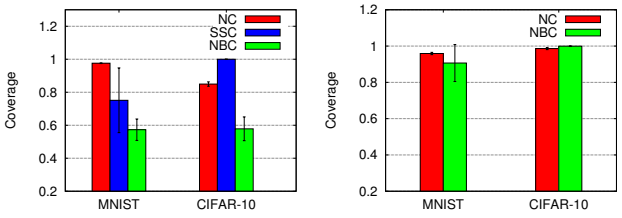


Figure 2: Adversarial images of DNNs, with L_∞-norm for MNIST (top row) and L_0-norm for CIFAR-10 (bottom row), generated by DeepConcolic and DeepXplore, the latter with image constraints 'light', 'occlusion', and 'blackout'.

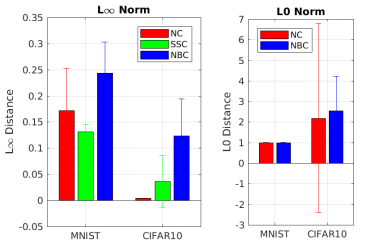
Figure 2 exhibits several adversarial examples found by DeepConcolic (with L_∞-norm and L_0-norm) and DeepXplore. It is worth noting that, although DeepConcolic does not impose particular domain-specific constraints on the original image as DeepXplore does, the concolic testing procedure generates test cases that resemble "human perception". For example, based on the L_∞-norm, it produces adversarial examples (Figure 2, top row) that gradually reverse the black and white colours. For the L_0-norm, DeepConcolic generates adversarial examples similar to those of DeepXplore under the 'blackout' constraint, which is essentially pixel manipulation.



(a) L_∞-norm (b) L_0-norm

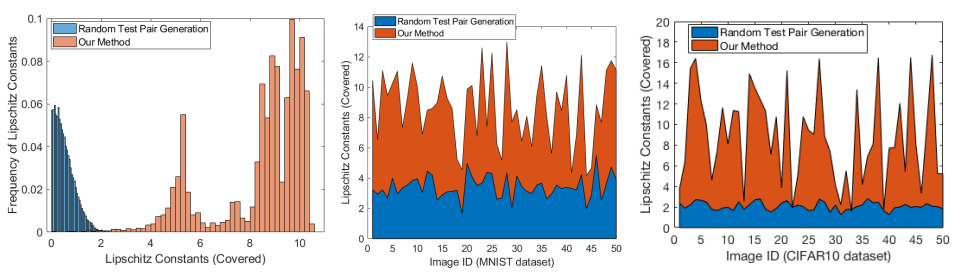
Figure 3: The coverage results of different criteria.

[3]The implementation and all data in this section are available online at https://github.com/TrustAI/DeepConcolic



(a) (b)

Figure 4: (a) Distance of NC, SSC, and NBC on MINIST and CIFAR-10 datasets based on L_∞ norm; (b) Distance of NC and NBC on the two datasets based on L_0 norm.



(a) (b) (c)

Figure 5: (a) Lipschitz Constant Coverage generated by 1,000,000 randomly generated test pairs and our concolic testing method for input image-1 on MNIST DNN; (b) Lipschitz Constant Coverages generated by random testing and our method for 50 input images on MNIST DNN; (c) Lipschitz Constant Coverage generated by random testing and our method for 50 input images on CIFAR-10 DNN.

11.2 Testing Results on Different Test Criteria

This section presents the results of applying DeepConcolic to evaluate the test criteria NC, SSC, and NBC. DeepConcolic starts the NC testing with one single seed input. For SSC and NBC, to improve the performance, an initial set of 1000 images are sampled. Furthermore, for experimental purposes, we only test a subset of the neurons for SSC and NBC. A distance upper bound of 0.3 (L_∞-norm) and 100 pixels (L_0-norm) is set up for collecting adversarial examples.

The full coverage report, including the average coverage and standard derivation, is given in Figure 3. Table 3 contains the adversarial example results. We have observed that the overhead for the symbolic analysis based on global optimisation in Section 8.2 is too high. Thus, the SSC result with L_0-norm is excluded.

Table 3: Adversarial examples by different test criteria, distance metrics, and DNN models

	L_∞-norm				L_0-norm			
	MNIST		CIFAR-10		MNIST		CIFAR-10	
	adversary /test suite	minimum distance	adversary /test suite	minimum distance	adversary /test suite	minimum distance	adversary /test suite	minimum distance
NC	13.93%	0.0039	0.79%	0.0039	0.53%	1	5.59%	1
SSC	0.02%	0.1215	0.36%	0.0039	–	–	–	–
NBC	0.20%	0.0806	7.71%	0.0113	0.09%	1	4.13%	1

Overall, DeepConcolic achieves high coverage and, according to the robustness check in Definition 9.2, detects a significant portion of adversarial examples, with the cover of corner-case activation values (i.e., NBC) sometimes being harder to achieve.

Concolic testing is able to find adversarial examples with the minimum possible distance: that is, $\frac{1}{255} \approx 0.0039$ for the L_∞ norm and 1 pixel for the L_0 norm. Figure 4 gives the average distance of adversarial examples (from one DeepConcolic run), which often falls into a reasonably small distance range. Remarkably, for the same network, the number of adversarial examples found under the NC can be quite different when the distance metric is changed. This observation suggests that, when designing test criteria for DNNs, they need to be examined using different distance metrics.

11.3 Results for Lipschitz Constant Testing

This section reports experimental results for the Lipschitz constant testing on DNNs. We test Lipschitz constants ranging over $\{0.01 : 0.01 : 20\}$ on 50 MNIST images and 50 CIFAR-10 images respectively. Every image represents a subspace in D_{L_1} and thus a requirement in Equation (11).

11.3.1 Baseline Method. Since this paper is the first to test Lipschitz constants of DNNs, we compare our method with random test case generation. For this specific test requirement, given a predefined Lipschitz constant c, an input t_0 and the radius of norm ball (e.g., for L_1 and L_2 norms) or hypercube space (for L_∞-norm) Δ, we randomly generate two test pairs t_1 and t_2 that satisfy the space constraint (i.e., $||t_1 - t_0||_{D_2} \leq \Delta$ and $||t_2 - t_0||_{D_2} \leq \Delta$), and then check whether $\mathbf{Lip}(t_1, t_2) > c$ holds. We repeat the random generation until we find a satisfying test pair or the number of repetitions is larger than a predefined threshold. We set such threshold as $N_{rd} = 1,000,000$. Namely, if we randomly generate 1,000,000 test pairs and none of them can satisfy the Lipschitz constant requirement $> c$, we treat this test as a failure and return the largest Lipschitz constant found and the corresponding test pair; otherwise, we treat it as successful and return the satisfying test pair.

11.3.2 Experimental Results. Figure 5 (a) depicts the Lipschitz Constant Coverage generated by 1,000,000 random test pairs and our concolic test generation method for image-1 on MNIST DNNs. As we can see, even though we produce 1,000,000 test pairs by random test generation, the maximum Lipschitz converage reaches only 3.23 and most of the test pairs are in the range $[0.01, 2]$. Our concolic method, on the other hand, can cover a Lipschitz range of $[0.01, 10.38]$, where most cases lie in $[3.5, 10]$, which is poorly covered by random test generation.

Figure 5 (b) and (c) compare the Lipschitz constant coverage of test pairs from the random method and the concolic method on both MNIST and CIFAR-10 models. Our method significantly outperforms random test case generation. We note that covering a large Lipschitz constant range for DNNs is a challenging problem since most image pairs (within a certain high-dimensional space) can produce small Lipschitz constants (such as 1 to 2). This explains the reason why randomly generated test pairs concentrate in a range of less than 3. However, for safety-critical applications such as self-driving cars, a DNN with a large Lipschitz constant essentially indicates it is more vulnerable to adversarial perturbations [20, 21].

As a result, a test method that can cover larger Lipschitz constants provides a useful robustness indicator for a trained DNN. We argue that, for safety testing of DNNs, the concolic test method for Lipschitz constant coverage can complement existing methods to achieve significantly better coverage.

12 CONCLUSIONS

In this paper, we propose the first concolic testing method for DNNs. We implement it in a software tool and apply the tool to evaluate DNN robustness, through coverage testing for Lipschitz continuity and several other test criteria. Experimental results confirm that the combination of concrete execution and symbolic analysis serves as a viable approach for DNN testing.

REFERENCES

[1] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein GAN. *arXiv preprint arXiv:1701.07875* (2017).

[2] Charles Audet and Warren Hare. 2017. *Derivative-Free and Blackbox Optimization*. Springer.

[3] Radu Balan, Maneesh Singh, and Dongmian Zou. 2017. Lipschitz Properties for Deep Convolutional Networks. *arXiv preprint arXiv:1701.05217* (2017).

[4] Jacob Burnim and Koushik Sen. 2008. Heuristics for Scalable Dynamic Test Generation. In *Automated Software Engineering, ASE. 23rd International Conference on*. IEEE, 443–446.

[5] Cristian Cadar, Daniel Dunbar, Dawson R Engler, and others. 2008. KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.. In *OSDI*, Vol. 8. 209–224.

[6] Sooyoung Cha, Seongjoon Hong, Junhee Lee, and Hakjoo Oh. 2018. Automatically Generating Search Heuristics for Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 1244–1254.

[7] Timon Gehr, Matthew Mirman, Dana Drachsler-Cohen, Petar Tsankov, Swarat Chaudhuri, and Martin Vechev. 2018. AI2: Safety and Robustness Certification of Neural Networks with Abstract Interpretation. In *Security and Privacy (SP), 2018 IEEE Symposium on*.

[8] Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. 213–223.

[9] Patrice Godefroid, Michael Y Levin, David A Molnar, and others. 2008. Automated Whitebox Fuzz Testing. In *NDSS*, Vol. 8. 151–166.

[10] Kelly Hayhurst, Dan Veerhusen, John Chilenski, and Leanna Rierson. 2001. *A Practical Tutorial on Modified Condition/Decision Coverage*. Technical Report. NASA.

[11] Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. 2017. Safety Verification of Deep Neural Networks. In *International Conference on Computer Aided Verification*. Springer, 3–29.

[12] Cem Kaner. 2006. Exploratory Testing. In *Quality Assurance Institute Worldwide Annual Software Testing Conference*.

[13] Raghudeep Kannavara, Christopher J Havlicek, Bo Chen, Mark R Tuttle, Kai Cong, Sandip Ray, and Fei Xie. 2015. Challenges and Opportunities with Concolic Testing. In *Aerospace and Electronics Conference (NAECON), 2015 National*. IEEE, 374–378.

[14] Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. 2017. Reluplex: An Efficient SMT Solver for Verifying Deep Neural Networks. In *International Conference on Computer Aided Verification*. Springer, 97–117.

[15] Lei Ma, Felix Juefei-Xu, Jiyuan Sun, Chunyang Chen, Ting Su, Fuyuan Zhang, Minhui Xue, Bo Li, Li Li, Yang Liu, and others. 2018. DeepGauge: Comprehensive and Multi-Granularity Testing Criteria for Gauging the Robustness of Deep Learning Systems. *arXiv preprint arXiv:1803.07519* (2018).

[16] Matthew Mirman, Timon Gehr, and Martin Vechev. 2018. Differentiable Abstract Interpretation for Provably Robust Neural Networks. In *International Conference on Machine Learning*. 3575–3583.

[17] Vinod Nair and Geoffrey E Hinton. 2010. Rectified Linear Units Improve Restricted Boltzmann Machines. In *International Conference on Machine Learning*. 807–814.

[18] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. DeepXplore: Automated Whitebox Testing of Deep Learning Systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 1–18.

[19] T. Roubicek. 1997. *Relaxation in Optimization Theory and Variational Calculus*. Berlin: Walter de Gruyter.

[20] Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. Reachability Analysis of Deep Neural Networks with Provable Guarantees. *The 27th International Joint Conference on Artificial Intelligence (IJCAI)* (2018).

[21] Wenjie Ruan, Min Wu, Youcheng Sun, Xiaowei Huang, Daniel Kroening, and Marta Kwiatkowska. 2018. Global Robustness Evaluation of Deep Neural Networks with Provable Guarantees for L0 Norm. *arXiv preprint arXiv:1804.05805v1* (2018).

[22] Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. *ACM SIGSOFT Software Engineering Notes* 30, 5 (2005), 263–272.

[23] Youcheng Sun, Xiaowei Huang, and Daniel Kroening. 2018. Testing Deep Neural Networks. *arXiv preprint arXiv:1803.04792* (2018).

[24] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. 2014. Intriguing Properties of Neural Networks. In *International Conference on Learning Representations (ICLR)*.

[25] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. 2018. DeepTest: Automated Testing of Deep-Neural-Network-Driven Autonomous Cars. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 303–314.

[26] Willem Visser, Corina S PÇŐsÇŐreanu, and Sarfraz Khurshid. 2004. Test Input Generation with Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 29, 4 (2004), 97–107.

[27] Xinyu Wang, Jun Sun, Zhenbang Chen, Peixin Zhang, Jingyi Wang, and Yun Lin. 2018. Towards Optimal Concolic Testing. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 291–302.

[28] Zhou Wang, Eero P Simoncelli, and Alan C Bovik. 2003. Multiscale Structural Similarity for Image Quality Assessment. In *Signals, Systems and Computers, Conference Record of the Thirty-Seventh Asilomar Conference on*.

[29] Thomas Wiatowski and Helmut Bölcskei. 2018. A Mathematical Theory of Deep Convolutional Neural Networks for Feature Extraction. *IEEE Transactions on Information Theory* 64, 3 (2018), 1845–1866.

[30] Matthew Wicker, Xiaowei Huang, and Marta Kwiatkowska. 2018. Feature-Guided Black-Box Safety Testing of Deep Neural Networks. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 408–426.

[31] Min Wu, Matthew Wicker, Wenjie Ruan, Xiaowei Huang, and Marta Kwiatkowska. 2018. A Game-Based Approximate Verification of Deep Neural Networks with Provable Guarantees. *arXiv preprint arXiv:1807.03571* (2018).

[32] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. 2005. Symstra: A Framework for Generating Object-Oriented Unit Tests using Symbolic Execution. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems (LNCS)*, Vol. 3440. Springer, 365–381.