Concolic Testing for Deep Neural Networks *

Youcheng Sun[1], Min Wu[1], Wenjie Ruan[1], Xiaowei Huang[2], Marta Kwiatkowska[1], and Daniel Kroening[1]

[1]University of Oxford, UK
{youcheng.sun; min.wu; wenjie.ruan}@cs.ox.ac.uk
{marta.kwiatkowska; daniel.kroening}@cs.ox.ac.uk
[2]University of Liverpool, UK
xiaowei.huang@liverpool.ac.uk

Abstract

Concolic testing combines program execution and symbolic analysis to explore the execution paths of a software program. This paper presents the first concolic testing approach for Deep Neural Networks (DNNs). More specifically, we formalise coverage criteria for DNNs that have been studied in the literature, and then develop a coherent method for performing concolic testing to increase test coverage. Our experimental results show the effectiveness of the concolic testing approach in both achieving high coverage and finding adversarial examples.

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 damage. To address these concerns, a (safety or security) critical system comprising DNN-based components needs to be validated thoroughly.

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 [1]. So far, there have been only few attempts to test DNNs systematically [2–6]. These are either based on concrete execution, e.g., Monte Carlo tree search [2] or gradient-based search [3, 4, 6], or symbolic execution in combination

*Kwiatkowska and Ruan are supported by EPSRC Mobile Autonomy Programme Grant (EP/M019918/1). Wu is supported by the CSC-PAG Oxford Scholarship.

with solvers for linear arithmetic [5]. Together with these test-input generation algorithms, several test coverage criteria have been presented, including neuron coverage [3], a criterion that is inspired by MC/DC [5], and criteria to capture particular neuron activation values to identify corner cases [6]. None of these approaches implement *concolic testing* [7, 8], 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 a set of coverage requirements, which we express using Quantified Linear Arithmetic over Rationals (QLAR). For a given set $\mathfrak{R}$ of coverage requirements, we incrementally generate a set of test inputs to improve coverage by alternating between concrete execution and symbolic analysis. Given an unsatisfied test requirement r, we identify a test input t within our current test suite such that t is close to satisfying r according to an evaluation based on *concrete execution*. After that, *symbolic analysis* is applied to obtain a new test input t' that satisfies r. The test input t' is then added to the test suite. This process is iterated until we reach a satisfactory level of coverage.

Finally, the generated test suite is passed to a *robustness oracle*, which determines whether the test suite includes *adversarial examples* [9], i.e., pairs of test cases that disagree on their classification labels when close to each other with respect to a given distance metric. The lack of robustness has been viewed as a major weakness of DNNs, and the discovery of adversarial examples 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 coverage requirements, including Lipschitz continuity [2, 10–13] and several structural coverage metrics [3, 5, 6]. We show experimentally that our 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 and that it is able to discover a significant number of adversarial examples.

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

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 [14], methods that use constraint solvers [15], global optimisation approaches [13] and abstract interpretation [16, 17] 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 [2]. 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., [2–6]. 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* [3], which resembles traditional statement coverage. A set of criteria inspired by MD/DC coverage [18] is used in [5]; Ma et al. [6] present criteria that are designed to capture particular values of neuron activations. Tian et al. [4] 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. [2] aim to cover the input space by exhaustive mutation testing that has theoretical guarantees, while in [3, 4, 6] gradient-based search algorithms are applied to solve optimisation problems, and Sun et al. [5] apply linear programming. None of these consider concolic testing and a general means for modeling test coverage requirements as we do in this paper.

2.2 Concolic Testing

By concretely executing the program with particular inputs, which includes random testing, a large number of inputs can be tested at low cost. 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 [19–21], an execution path is encoded symbolically. Modern constraint solvers can determine feasibility of the encoding effectively, although performance still degrades as the size of the symbolic representation increases. Concolic testing [7, 8] is an effective approach to automated test input generation. It is a hybrid software testing technique that alternates between concrete execution, i.e., testing on particular inputs, and symbolic execution, a classical technique that treats program variables as symbolic values [22].

Concolic testing has been applied routinely in software testing, and a wide range of tools is available, e.g., [7, 8, 23]. 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 alternate until a desired 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 [23, 24]. Automated generation of search heuristics for concolic testing is an active area of research [25, 26].

Table 1: Comparison with different coverage-driven DNN testing methods

	DeepConcolic	DeepXplore [3]	DeepTest [4]	DeepCover [5]	DeepGauge [6]
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_∞ and L_0-norm	L_1-norm	Jaccard distance	L_∞-norm	L_∞-norm

2.3 Comparison with Related Work

We briefly summarise the similarities and differences between our concolic testing method, named *DeepConcolic*, and other existing coverage-driven DNN testing methods: DeepXplore [3], DeepTest [4], DeepCover [5], and DeepGauge [6]. The details are presented in Table 1, where NC, SSC, and NBC are short for Neuron Coverage, SS Coverage, and Neuron Boundary Coverage, respectively. In addition to the concolic nature of DeepConcolic, we observe the following differences.

- DeepConcolic is generic, and is able to take coverage requirements as input; the other methods are *ad hoc*, and are tailored to specific requirements.

- DeepXplore requires 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, DeepConcolic features a clean separation between the generation of test inputs and the test oracle. This is a good fit for traditional test case generation. The other methods use the oracle as part of their objectives to guide the generation of test inputs.

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) [27]. For convenience, we explicitly denote the activation value before the ReLU as $u_{k,l}$ such that

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

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 $\forall 1 < k \leq K, \forall 1 \leq l \leq s_k$,

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

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 that has the largest value, i.e., $label = \text{argmax}_{1 \leq l \leq s_K} \{v_{K,l}\}$.

Due to the existence of ReLU, the neural network is a highly non-linear function. 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 $1 \leq k \leq s_k$ 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 10) our method also applies to other popular layers, e.g., maxpooling, used in state-of-the-art DNNs.

4 Test 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* [5].

Definition 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 $\{\mathit{true}, \mathit{false}\}$. We 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,*

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

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 [5].

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 coverage criteria, and then generate test inputs that cover these activation patterns.

4.2 Formalizing Test Coverage Criteria

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

Definition 2 *Given a network $\mathcal{N}$, we write $IV = \{x, x_1, x_2, ...\}$ for a set of variables that range over the all inputs D_{L_1} of the network. We define $V = \{u[x]_{k,l}, v[x]_{k,l} \mid 1 \leq k \leq K, 1 \leq l \leq s_k, x \in IV\}$ to be a set of variables that range over the rationals. We fix the following syntax for DR formulas:*

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

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

The formula $\exists x.e$ expresses that there exists an input x such that e is true, while $\forall x.e$ expresses that e is true for all inputs x. The formulas $\exists x_1, x_2.e$ and $\forall x_1, x_2.e$ 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 in the syntax above using 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

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

to express $||x_1 - x_2||_\infty \leq q$, 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 vector x.

Semantics

We define the satisfiability of a coverage requirement r by a test suite $\mathcal{T}$.

Definition 3 *Given a set $\mathcal{T}$ of test inputs and a coverage requirement r, the satisfiability relation $\mathcal{T} \models r$ is defined as follows.*

- *$\mathcal{T} \models \exists x.e$ if there exists some test $t \in \mathcal{T}$ such that $\mathcal{T} \models e[x \mapsto t]$, where $e[x \mapsto t]$ denotes the expression e in which the occurrences of x are replaced by t.*

- $\mathcal{T} \models \exists x_1, x_2.e$ *if there exist two tests* $t_1, t_2 \in \mathcal{T}$ *such that* $\mathcal{T} \models e[x_1 \mapsto t_1][x_2 \mapsto t_2]$

The cases for $\forall$ *formulas are similar. 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* $v[t]_{k,l}$ *derive their values from the activation patters of the DNN for test* t*, and* $c \cdot u[t]_{k,l}$ *and* $c \cdot v[t]_{k,l}$ *have the standard meaning where* c *is a coefficient,*
- p*,* $a_1 + a_2$*, and* $a_1 - a_2$ *have the standard semantics.*

Note that $\mathcal{T}$ is finite. It is trivial to extend the definition of the satisfaction relation to an infinite subspace of inputs.

Complexity

Given a network $\mathcal{N}$, a DR requirement formula r, and a test suite $\mathcal{T}$, checking $\mathcal{T} \models r$ can be done in time that is polynomial in the size of $\mathcal{T}$. Determining whether there exists a test suite $\mathcal{T}$ with $\mathcal{T} \models r$ is NP-complete.

4.3 Test Coverage Metrics

Now we can define test coverage criteria by providing a set of requirements on the test suite. The coverage metric is defined in the standard way as the percentage of the test requirements that are satisfied by the test cases in the test suite $\mathcal{T}$.

Definition 4 (Coverage Metric) *Given a network* $\mathcal{N}$*, a set* $\mathfrak{R}$ *of test coverage requirements expressed as DR formulas, and a test suite* $\mathcal{T}$*, the test coverage metric* $M(\mathfrak{R}, \mathcal{T})$ *is as follows:*

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

The coverage is used as a proxy metric for the confidence in the safety of the DNN under test.

5 Specific Coverage Requirements

In this section, we give DR^+ formulas for several important coverage criteria for DNNs, including Lipschitz continuity [2, 10–13] and test coverage criteria from the literature [3, 5, 6]. The criteria we consider have syntactical similarity with structural test coverage criteria in conventional software testing. Lipschitz continuity is semantic, specific to DNNs, and has been shown to be closely related to the theoretical understanding of convolutional DNNs [10] and the robustness of both DNNs [2, 13] and Generative Adversarial Networks [11]. These criteria have been studied in the literature using a variety of formalisms and approaches.

Each test coverage criterion gives rise to a set of test coverage requirements. In the following, we discuss the three coverage criteria from [3, 5, 6], respectively. We use $||t_1 - t_2||_q$ to denote the distance between two inputs t_1 and t_2 with respect to a given distance metric $||\cdot||_q$. The metric $||\cdot||_q$ can be, e.g., a norm-based metric such as the L_0-norm (the Hamming distance), the L_2-norm (the Euclidean distance), or the L_∞-norm (the 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 10 elaborates on the particular metrics we use for our 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.

$$\bigwedge_{i=1}^{s_1} x(i) - u \leq 0 \wedge x(i) - l \geq 0 \tag{8}$$

5.1 Lipschitz Continuity

In [9, 13], Lipschitz continuity has been shown to hold for a large class of DNNs, including DNNs for image classification.

Definition 5 (Lipschitz Continuity) *A network $\mathcal{N}$ is said to be* Lipschitz continuous *if there exists a real constant $c \geq 0$ such that, for all $x_1, x_2 \in D_{L_1}$:*

$$||v[x_1]_1 - v[x_2]_1|| \leq c \cdot ||x_1 - x_2|| \tag{9}$$

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

Since the computation of c_{best} is an NP-hard problem and a smaller c can significantly improve the performance of verification algorithms [2, 13, 29], it is interesting to determine whether a given c is a Lipschitz constant, either for the entire input space D_{L_1} or for some subspace. Testing for Lipschitz continuity can be guided using the following requirements.

Definition 6 (Lipschitz Coverage) *Given a real $c > 0$ and an integer $b > 0$, the set $\mathfrak{R}_{Lip}(b, c)$ of requirements for Lipschitz coverage is*

$$\begin{array}{c}\{\exists x_1, x_2.(||v[x_1]_1 - v[x_2]_1|| - c \cdot ||x_1 - x_2|| > 0) \\ \wedge x_1, x_2 \in X \mid X \in \mathcal{S}(D_{L_1}, b)\}\end{array} \tag{10}$$

where the $\mathcal{S}(D_{L_1}, b)$ are given input subspaces.

Intuitively, for each $X \in \mathcal{S}(D_{L_1}, b)$, this requirement expresses the existence of two inputs x_1 and x_2 that refute that c is a Lipschitz constant for $\mathcal{N}$. It is typically impossible to obtain full Lipschitz coverage, because there may exist inconsistent $r \in \mathfrak{R}_{Lip}(b, c)$. Thus, the goal for a test case generation algorithm is to produce a test suite $\mathcal{T}$ that satisfies the criterion as much as possible.

5.2 Neuron Coverage

Neuron Coverage (NC) [3] is an adaptation of statement coverage in conventional software testing to DNNs. It is defined as follows.

Definition 7 *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 $ap[t]_{k,i} = \mathit{true}$.*

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

Definition 8 (NC Requirements) *The set $\mathfrak{R}_{NC}$ of coverage requirements for Neuron Coverage is*

$$\{\exists x.ap[x]_{k,i} = \mathit{true} \mid 2 \leq k \leq K-1, 1 \leq i \leq s_k\} \tag{11}$$

5.3 Modified Condition/Decision (MC/DC) Coverage

In [5], a family of four test criteria is proposed, inspired by MC/DC coverage in conventional software testing. We will restrict the discussion here to Sign-Sign Coverage (SSC). According to [5], each neuron $n_{k+1,j}$ can be seen as a *decision* where the 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, we must show that all condition neurons can determine the outcome of the decision neuron independently. In the case of SSC coverage we say that the value of a decision or condition neuron changes if the sign of its activation function changes. Consequently, the requirements for SSC coverage are defined as follows.

Definition 9 (SSC Requirements) *For SCC coverage, we first define a requirement $\mathfrak{R}_{SSC}(\alpha)$ for a pair of neurons $\alpha = (n_{k,i}, n_{k+1,j})$:*

$$\begin{array}{c}\{\exists x_1, x_2.\ ap[x_1]_{k,i} \neq ap[x_2]_{k,i} \wedge ap[x_1]_{k+1,j} \neq ap[x_2]_{k+1,j} \wedge \\ \bigwedge_{1 \leq l \leq s_k, l \neq i} ap[x_1]_{k,l} - ap[x_2]_{k,l} = 0\}\end{array} \tag{12}$$

and we get

$$\mathfrak{R}_{SSC} = \bigcup_{2\leq k\leq K-2, 1\leq i\leq s_k, 1\leq j\leq s_{k+1}} \mathfrak{R}_{SSC}((n_{k,i}, n_{k+1,j})) \tag{13}$$

That is, for each pair $(n_{k,i}, n_{k+1,j})$ of neurons in two adjacent layers k and $k+1$, 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 that the change is independent. The idea of SS Coverage (and all other criteria in [5]) is to ensure that not only the existence 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 Coverage

Neuron Boundary Coverage (NBC) [6] aims at covering neuron activation values that exceed a given bound. It can be formulated as follows.

Definition 10 (NBC Requirements) *Given two sets of bounds* $h = \{h_{k,i} | 2 \leq k \leq K-1, 1 \leq i \leq s_k\}$ *and* $l = \{l_{k,i} | 2 \leq k \leq K-1, 1 \leq i \leq s_k\}$*, the requirements* $\mathfrak{R}_{NBC}(h, l)$ *are*

$$\{\exists x.\ u[x]_{k,i} - h_{k,i} > 0,\ \ \exists x.\ u[x]_{k,i} - l_{k,i} < 0 \mid 2 \leq k \leq K-1, 1 \leq i \leq s_k\} \tag{14}$$

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 Overview of our Approach

This section gives an overview of our method for generating a test suite for a given DNN. Our method alternates between concrete evaluation of the activation patterns of the DNN and symbolic generation of new inputs. The pseudocode for our method is given as Algorithm 1. It is visualised in Figure 1.

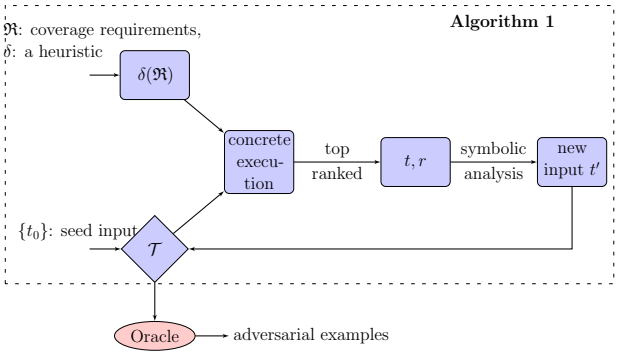


Figure 1: Overview of our concolic testing method

Algorithm 1 takes as inputs a DNN $\mathcal{N}$, an input t_0 for the DNN, a heuristic δ, and a set $\mathfrak{R}$ of coverage requirements, and produces a test suite $\mathcal{T}$ as output. The test suite $\mathcal{T}$ initially only contains the given test input t_0. The algorithm removes a requirement $r \in \mathfrak{R}$ from $\mathfrak{R}$ once it is satisfied by $\mathcal{T}$, i.e., $\mathcal{T} \models r$.

Algorithm 1 Concolic Testing for DNNs

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

```
1:  T ← {t_0} and F = {}
2:  t ← t_0
3:  while R \ S ≠ ∅ 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 validity_check(t') = true then
10:       T ← T ∪ {t'}
11:       break
12:     else if cost exceeded then
13:       F ← F ∪ {r}
14:       break
15: return T
```

The function *requirement_evaluation* (Line 7), whose details are given in Section 7, looks for a pair (t, r) [2] of input and requirement that, according to our concrete evaluation, is the most promising candidate for a new test case t' that satisfies the requirement r. The heuristic δ is a transformation function that maps a formula r with operator $\exists$ to an optimisation problem. This step relies on concrete execution.

After obtaining (t, r), *symbolic_analysis* (Line 8), whose details are in Section 8, is applied to obtain a new concrete input t'. Then a function *validity_check* (Line 9), whose details are given in Section 9, is applied to check whether the new input is valid or not. If so, the test is added to the test suite. Otherwise, ranking and symbolic input generation are repeated until a given computational cost is exceeded, after which test generation for the requirement is deemed to have failed. This is recorded in the set F.

The algorithm terminates when either all test requirements have been satisfied, i.e., $\mathfrak{R} = \emptyset$, or no further requirement in $\mathfrak{R}$ can be satisfied, i.e., $F = \mathfrak{R}$. It then returns the current test suite $\mathcal{T}$.

Finally, as illustrated in Figure 1, the test suite $\mathcal{T}$ generated by Algorithm 1,

[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.

is passed to an oracle in order to evaluate the robustness of the DNN. The details of the oracle are in Section 9.

7 Ranking Coverage Requirements

This section presents our approach for Line 7 of Algorithm 1. Given a set of requirements $\mathfrak{R}$ that have not yet been satisfied, a heuristic δ, and the current set $\mathcal{T}$ of test inputs, 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 compute the next concrete input t' (to be given in Section 8). The selection of t and r is done by means of a series of concrete executions.

The general idea 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

$$r = \arg\max_{r}\{val(t, \delta(r)) \mid r \in \mathfrak{R}\}. \tag{15}$$

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 $\delta(r)$ is 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 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.) determines the input x among those satisfying the Boolean formula e that maximises (minimises) 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 3, the semantics is for a set $\mathcal{T}$ of test cases and we can adapt it to a continuous input subspace $X \subseteq D_{L_1}$.

7.1 Heuristics

We present the heuristics δ we use the coverage requirements discussed in Section 5. We remark that, since δ is a heuristic, there exist alternatives. The following definitions work well in our experiments.

7.1.1 Lipschitz Continuity

When a Lipschitz requirement r as in Equation (10) is not satisfied by $\mathcal{T}$, we transform it into $\delta(r)$ as follows:

$$\arg\max_{x_1,x_2} .||v[x_1]_1 - v[x_2]_1|| - c * ||x_1 - x_2|| : x_1, x_2 \in X \tag{16}$$

I.e., the aim is to find the best t_1 and t_2 in $\mathcal{T}$ to make $||v[t_1]_1 - v[t_2]_1|| - c \cdot ||t_1 - t_2||$ as large as possible. As described, we also need to compute $val(t_1, t_2, r) = ||v[t_1]_1 - v[t_2]_1|| - c \cdot ||t_1 - t_2||$.

7.1.2 Neuron Cover

When a requirement r as in Equation (11) is not satisfied by $\mathcal{T}$, we transform it into the following requirement $\delta(r)$:

$$\arg\max_{x} c_k \cdot u_{k,i}[x] : \mathsf{true} \tag{17}$$

We obtain 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 per-layer constant. It motivated by 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 that normalises the average activation valuations of neurons at different layers into the same magnitude level. It is estimated by sampling a sufficiently large input dataset.

7.1.3 SS Coverage

In SS Coverage, 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 the test input that is generated negates the signs of $n_{k,i}$ and $n_{k+1,j}$ while the remainder of $n_{k+1,j}$'s condition neurons preserve their respective signs. This is achieved by the following $\delta(r)$:

$$\arg\max_{x} -c_k \cdot |u[x]_{k,i}| : \mathsf{true} \tag{18}$$

Intuitively, given the decision neuron $n_{k+1,j}$, Equation (18) selects the condition that is closest to the change of activation sign (i.e., yields the smallest $|u[x]_{k,i}|$).

7.1.4 Neuron Boundary Coverage

We transform the requirement r in Equation (19) into the following $\delta(r)$ when it is not satisfied by $\mathcal{T}$; it selects the neuron that is closest to either the higher or lower boundary.

$$\begin{array}{l} \arg\max_x c_k \cdot (u[x]_{k,i} - h_{k,i}) : \mathsf{true} \\ \arg\max_x c_k \cdot (l_{k,i} - u[x]_{k,i}) : \mathsf{true} \end{array} \tag{19}$$

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 (Line 10 of Algorithm 1). The symbolic analysis techniques to be considered include the linear programming in [5], global optimisation for the L_0 norm in [30], and a new optimisation algorithm that will 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.

To simplify the presentation, the following description may, for each algorithm, focus on some specific coverage 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 [5] yield a set of inputs that exhibit the same ReLU behaviour as x:

$$\{\mathbf{u_{k,i}} = \sum_{1\leq j\leq s_{k-1}} \{w_{k-1,j,i} \cdot \mathbf{v_{k-1,j}}\} + b_{k,i} \mid k \in [2, K], i \in [1..s_k]\} \tag{20}$$

$$\begin{array}{l} \{\mathbf{u_{k,i}} \geq 0 \wedge \mathbf{u_{k,i}} = \mathbf{v_{k,i}} \mid ap[x]_{k,i} = \mathsf{true}, k \in [2, K), i \in [1..s_k]\} \\ \cup\{\mathbf{u_{k,i}} < 0 \wedge \mathbf{v_{k,i}} = 0 \mid ap[x]_{k,i} = \mathsf{false}, k \in [2, K), i \in [1..s_k]\} \end{array} \tag{21}$$

Continuous variables in the LP model are emphasized in **bold**.

- The activation value of each neuron is encoded by the linear constraint in (20), which is a symbolic version of Equation (2) that calculates a neuron's activation value.
- Given a particular input x, the activation pattern (Definition 1) $ap[x]$ is known: $ap[x]_{k,i}$ is either true or false, which indicates whether the ReLU is activated or not for the neuron $n_{k,i}$. Following (3) and the definition of ReLU in (1), for every neuron $n_{k,i}$, the linear constraints in (21) encode ReLU activation (when $ap[x]_{k,i} = \mathsf{true}$) or deactivation (when $ap[x]_{k,i} = \mathsf{false}$).

The linear model (denoted as $\mathcal{C}$) given by (20) and (21) 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, an objective is added to the LP model that minimizes the distance between t and t'.* Thus, the use of LP requires that the distance metric is linear. For instance, this applies to the L_∞-norm in (6), but not to the L_2-norm.

8.1.1 Neuron Coverage

The symbolic analysis of neuron coverage takes the input test case t and requirement r on the activation of neuron $n_{k,i}$, and returns a new test t' such that the test requirement is satisfied by the network instance $\mathcal{N}[t']$. We have the activation pattern $ap[t]$ of the given $\mathcal{N}[t]$, and can build up a new activation pattern ap' such that

$$\{ap'_{k,i} = \neg ap[t]_{k,i} \wedge \forall k_1 < k : \bigwedge_{0 \leq i_1 \leq s_{k_1}} ap'_{k_1,i_1} = ap[t]_{k_1,i_1}\} \tag{22}$$

This activation pattern specifies the following conditions.

- $n_{k,i}$'s activation sign is negated: this encodes the goal 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 multiple 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 (22) is encoded by the LP model $\mathcal{C}$ using (20) and (21), and if there exists a feasible solution, then the new test input t', which satisfies the requirement r, can be extracted from that solution.

8.1.2 SS Coverage

To satisfy an SS Coverage requirement r, we need to find a new test case such that, with respect to the input t, the activation signs of $n_{k+1,j}$ and $n_{k,i}$ are negated, while other signs of other neurons at layer k are equal to those for input t.

To achieve this, the following activation pattern ap' is constructed.

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

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 activation value of neuron $n_{k,i}$ exceeds either its higher bound $h_{k,i}$ or its lower bound $l_{k,i}$.

To achieve this, while preserving the DNN activation pattern $ap[t]$, we add one of the following constraints to the LP program.

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

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 [30]. That is, by specifying the test requirement as an optimisation objective, we apply global optimisation to compute a test case that satisfies the test coverage requirement.

- For Neuron Coverage, the objective is 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

$$ap[t']_{k,i} \neq ap[t]_{k,i} \wedge ap[t']_{k+1,j} \neq ap[t]_{k+1,j} \wedge \bigwedge_{i' \neq i} ap[t']_{k,i'} = ap[t]_{k,i}$$

- 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' is either $u[t']_{k,i} > h_{k,i}$ or $u[t']_{k,i} < l_{k,i}$.

Readers are referred to [30] for the details of the algorithm.

8.3 Lipschitz Test Case Generation

Given a coverage requirement as in Equation (10) 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

$$\begin{aligned} &||v[t_1]_1 - v[t_2]_1||_{D_1} - c \cdot ||t_1 - t_2||_{D_1} > 0 \\ &\text{s.t. } \; ||t_1 - t_0||_{D_2} \leq \Delta, \; ||t_2 - t_0||_{D_2} \leq \Delta \end{aligned} \tag{23}$$

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

The above problem can be efficiently solved by a novel *alternating compass search* scheme. Specifically, we alternate between solving the following two optimisation problems through relaxation [31], i.e., maximizing the lower bound of the original Lipschitz constant instead of directly maximizing the Lipschitz constant itself. To do so, we reformulate the original non-linear proportional optimisation as a linear problem when both norm metrics $||*||_{D_1}$ and $||*||_{D_2}$ are the L_∞-norm.

8.3.1 Stage One

We solve

$$\begin{aligned} \min_{t_1} F(t_1, t_0) = &-||v[t_1]_1 - v[t_0]_1||_{D_1} \\ &\text{s.t. } \; ||t_1 - t_0||_{D_2} \leq \Delta \end{aligned} \tag{24}$$

The objective above 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, maximising the norm distance of $v[t_1]_1$ and $v[t_0]_1$. The constraint implies that $\sup_{||t_1-t_0||_{D_2}\leq\Delta} ||t_1 - t_0||_{D_2} = \Delta$. Thus, a smaller $F(t_1, t_0)$ yields 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 search for a trace that minimises $F(t_1, t_0)$ increases the Lipschitz constant.

To solve the problem above we use the *compass search method* [32], which is efficient, derivative-free, and guaranteed to provide first-order global convergence. Because we aim to find an input pair that refutes the given Lipschitz constant c instead of finding the largest possible 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 (24) is convergent and we can find an optimal t_1 as

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

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

8.3.2 Stage Two

We solve

$$\begin{aligned} \min_{t_2} F(t_1^*, t_2) = -||v[t_2]_1 - v[t_1^*]_1||_{D_1} \\ \text{s.t. } \; ||t_2 - t_0||_{D_2} \leq \Delta \end{aligned} \tag{25}$$

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 (25) 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 (25) and continue the search and satisfiability checking procedure.

8.3.3 Stage Three

If the function $\mathbf{Lip}(t_1^*, t_2^*)$ fails to make progress in Stage Two, we treat the whole search procedure as convergent and have failed to find an input pair that can refute the given Lipschitz constant c. In this case, we return the best input pair we found so far, i.e., t_1^* and t_2^*, and the largest Lipschitz constant $\mathbf{Lip}(t_1^*, t_2)$ observed. Note that the returned constant is smaller than c.

In summary, the proposed method is an alternating optimisation scheme based on 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 run) to initiate another optimisation trajectory on the space, i.e., $S(t_1^*, \Delta(t_0))$. This process is repeated until we have gradually covered the entire space $S(\Delta(t_0))$ of the norm ball.

9 Test Oracle

We provide details about the validity checking performed for the generated test inputs (Line 9 of Algorithm 1) and how the test suite is finally used to quantify the safety of the DNN.

Definition 11 (Valid test input) *We are given a set O of inputs for which we assume to have a correct classification (e.g., the training dataset). Given a real number b, a test input $t' \in \mathcal{T}$ is said to be* valid *if*

$$\exists t \in O : ||t - t'|| \leq b\,. \tag{26}$$

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

To quantify the quality of the DNN using a test suite $\mathcal{T}$, we use the following robustness criterion.

Definition 12 (Robustness Oracle) *Given a set O of classified inputs, a test case t' passes the robustness oracle if*

$$\arg\max_j v[t']_{K,j} = \arg\max_j v[O(t')]_{K,j} \tag{27}$$

Whenever we identify a test input t' that fails to pass this oracle, then it serves as evidence that the DNN lacks robustness.

10 Experimental Results

We have implemented the concolic testing approach presented in this paper in a tool we have named DeepConcolic[3]. We compare it with other tools for testing DNNs. The experiments are run on a machine with 24 core Intel(R) Xeon(R) CPU E5-2620 v3 and 2.4 GHz and 125 GB memory. We use a timeout of 12 h. All coverage results are averaged over 10 runs or more.

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

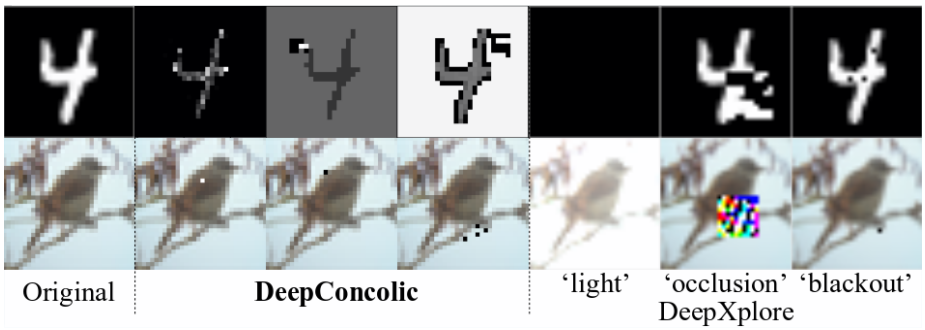


Figure 2: Adversarial images, with L_∞-norm for MNIST (top row) and L_0-norm for CIFAR-10 (bottom row), generated by DeepConcolic and DeepXplore, the latter with image constraints 'light', 'occlusion', and 'blackout'.

10.1 Comparison with DeepXplore

We now compare DeepConcolic and DeepXplore [3] on DNNs obtained from the MNIST and CIFAR-10 datasets. We remark that DeepXplore has been applied to 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 coverage obtained by 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

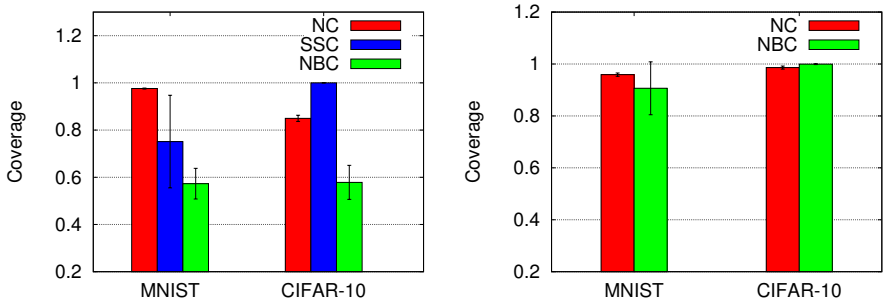
	DeepConcolic		DeepXplore		
	L_∞-norm	L_0-norm	light	occlusion	blackout
MNIST	97.60%	95.91%	80.77%	82.68%	81.61%
CIFAR-10	84.98%	98.63%	77.56%	81.48%	83.25%

Figure 2 presents several adversarial examples found by DeepConcolic (with L_∞-norm and L_0-norm) and DeepXplore. Although DeepConcolic does not impose particular domain-specific constraints on the original image as DeepXplore does, concolic testing generates images 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.

10.2 Results for NC, SCC, and NBC

We give the results obtained with DeepConcolic using the coverage criteria NC, SSC, and NBC. DeepConcolic starts NC testing with one single seed input. For SSC and NBC, to improve the performance, an initial set of 1000 images are sampled. Furthermore, 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 with global optimisation (Section 8.2) is too high. Thus, the SSC result with L_0-norm is excluded.



(a) L_∞-norm (b) L_0-norm

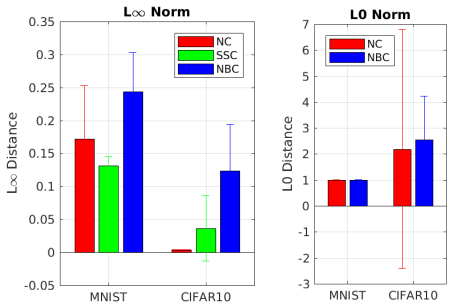
Figure 3: Coverage results for different criteria

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

	L_∞-norm				L_0-norm			
	MNIST		**CIFAR-10**		**MNIST**		**CIFAR-10**	
	adversary %	minimum dist.	adversary %	minimum dist.	adversary %	minimum dist.	adversary %	minimum dist.
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, using the robustness check (Definition 12), detects a significant number of adversarial examples. However, coverage of corner-case activation values (i.e., NBC) is limited.

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). Remarkably, for the same network, the number of adversarial examples found with NC can vary substantially when the distance metric is changed. This observation suggests that, when designing coverage criteria for DNNs, they need to be examined using a variety of distance metrics.



(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.

10.3 Results for Lipschitz Constant Testing

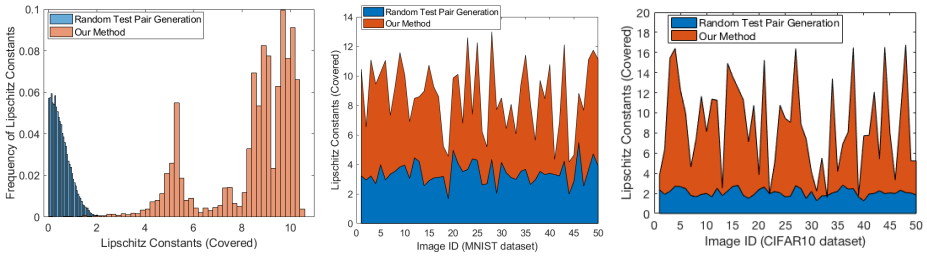
This section reports experimental results for the Lipschitz constant testing on DNNs. We test Lipschitz constants ranging over $\{0.01 : 0.01 : 20\}$ on 50 MNIST images and 50 CIFAR-10 images respectively. Every image represents a subspace in D_{L_1} and thus a requirement in Equation (10).

10.3.1 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.

10.3.2 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



(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.

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 [13, 30]. 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.

11 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 the robustness of well-known DNNs. The generation of the test inputs can be guided by a variety of coverage metrics, including Lipschitz continuity. Our experimental results confirm that the combination of concrete execution and symbolic analysis delivers both coverage and automates the discovery of adversarial examples.

Acknowledgements

This document is an overview of UK MOD (part) sponsored research and is released for informational purposes only. The contents of this document should not be interpreted as representing the views of the UK MOD, nor should it be assumed that they reflect any current or future UK MOD policy. The information contained in this document cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

References

[1] C. Kaner, "Exploratory testing," in *Quality Assurance Institute Worldwide Annual Software Testing Conference*, 2006.

[2] M. Wicker, X. Huang, and M. Kwiatkowska, "Feature-guided black-box safety testing of deep neural networks," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS*. Springer, 2018, pp. 408–426.

[3] K. Pei, Y. Cao, J. Yang, and S. Jana, "DeepXplore: Automated whitebox testing of deep learning systems," in *Proceedings of the 26th Symposium on Operating Systems Principles*. ACM, 2017, pp. 1–18.

[4] Y. Tian, K. Pei, S. Jana, and B. Ray, "DeepTest: Automated testing of deep-neural-network-driven autonomous cars," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 303–314.

[5] Y. Sun, X. Huang, and D. Kroening, "Testing deep neural networks," *arXiv preprint arXiv:1803.04792*, 2018.

[6] L. Ma, F. Juefei-Xu, J. Sun, C. Chen, T. Su, F. Zhang, M. Xue, B. Li, L. Li, Y. Liu *et al.*, "DeepGauge: Comprehensive and multi-granularity testing criteria for gauging the robustness of deep learning systems," *arXiv preprint arXiv:1803.07519*, 2018.

[7] P. Godefroid, N. Klarlund, and K. Sen, "DART: directed automated random testing," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2005, pp. 213–223.

[8] K. Sen, D. Marinov, and G. Agha, "CUTE: A concolic unit testing engine for C," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 263–272, 2005.

[9] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," in *International Conference on Learning Representations (ICLR)*, 2014.

[10] T. Wiatowski and H. Bölcskei, "A mathematical theory of deep convolutional neural networks for feature extraction," *IEEE Transactions on Information Theory*, vol. 64, no. 3, pp. 1845–1866, 2018.

[11] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein GAN," *arXiv preprint arXiv:1701.07875*, 2017.

[12] R. Balan, M. Singh, and D. Zou, "Lipschitz properties for deep convolutional networks," *arXiv preprint arXiv:1701.05217*, 2017.

[13] W. Ruan, X. Huang, and M. Kwiatkowska, "Reachability analysis of deep neural networks with provable guarantees," in *The 27th International Joint Conference on Artificial Intelligence, IJCAI*, 2018, pp. 2651–2659.

[14] X. Huang, M. Kwiatkowska, S. Wang, and M. Wu, "Safety verification of deep neural networks," in *International Conference on Computer Aided Verification, CAV*. Springer, 2017, pp. 3–29.

[15] G. Katz, C. Barrett, D. L. Dill, K. Julian, and M. J. Kochenderfer, "Reluplex: An efficient SMT solver for verifying deep neural networks," in *International Conference on Computer Aided Verification*. Springer, 2017, pp. 97–117.

[16] T. Gehr, M. Mirman, D. Drachsler-Cohen, P. Tsankov, S. Chaudhuri, and M. Vechev, "Ai2: Safety and robustness certification of neural networks with abstract interpretation," in *Security and Privacy (SP), 2018 IEEE Symposium on*, 2018.

[17] M. Mirman, T. Gehr, and M. Vechev, "Differentiable abstract interpretation for provably robust neural networks," in *International Conference on Machine Learning*, 2018, pp. 3575–3583.

[18] K. Hayhurst, D. Veerhusen, J. Chilenski, and L. Rierson, "A practical tutorial on modified condition/decision coverage," NASA, Tech. Rep., 2001.

[19] C. Cadar, D. Dunbar, and D. R. Engler, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, vol. 8, 2008, pp. 209–224.

[20] T. Xie, D. Marinov, W. Schulte, and D. Notkin, "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, TACAS*, ser. LNCS, vol. 3440. Springer, 2005, pp. 365–381.

[21] W. Visser, C. S. Pasareanu, and S. Khurshid, "Test input generation with Java PathFinder," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 4, pp. 97–107, 2004.

[22] R. Kannavara, C. J. Havlicek, B. Chen, M. R. Tuttle, K. Cong, S. Ray, and F. Xie, "Challenges and opportunities with concolic testing," in *Aerospace and Electronics Conference (NAECON), 2015 National*. IEEE, 2015, pp. 374–378.

[23] J. Burnim and K. Sen, "Heuristics for scalable dynamic test generation," in *Automated Software Engineering, ASE. 23rd International Conference on*. IEEE, 2008, pp. 443–446.

[24] P. Godefroid, M. Y. Levin, D. A. Molnar *et al.*, "Automated whitebox fuzz testing," in *NDSS*, vol. 8, 2008, pp. 151–166.

[25] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proceedings of the 40th International Conference on Software Engineering*. ACM, 2018, pp. 291–302.

[26] S. Cha, S. Hong, J. Lee, and H. Oh, "Automatically generating search heuristics for concolic testing," in *Proceedings of the 40th International Conference on Software Engineering, ICSE*. ACM, 2018, pp. 1244–1254.

[27] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *International Conference on Machine Learning*, 2010, pp. 807–814.

[28] Z. Wang, E. P. Simoncelli, and A. C. Bovik, "Multiscale structural similarity for image quality assessment," in *Signals, Systems and Computers, Conference Record of the Thirty-Seventh Asilomar Conference on*, 2003.

[29] M. Wu, M. Wicker, W. Ruan, X. Huang, and M. Kwiatkowska, "A game-based approximate verification of deep neural networks with provable guarantees," *arXiv preprint arXiv:1807.03571*, 2018.

[30] W. Ruan, M. Wu, Y. Sun, X. Huang, D. Kroening, and M. Kwiatkowska, "Global robustness evaluation of deep neural networks with provable guarantees for L0 norm," *arXiv preprint arXiv:1804.05805v1*, 2018.

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

[32] C. Audet and W. Hare, *Derivative-Free and Blackbox Optimization*. Springer, 2017.