Testing Deep Neural Networks

Youcheng Sun[1], Xiaowei Huang[2], and Daniel Kroening[1]

[1] Department of Computer Science, University of Oxford, UK
[2] Department of Computer Science, University of Liverpool, UK

Abstract. Deep neural networks (DNNs) have a wide range of applications, and software employing them must be thoroughly tested, especially in safety critical domains. However, traditional software testing methodology, including test coverage criteria and test case generation algorithms, cannot be applied directly to DNNs. This paper bridges this gap. First, inspired by the traditional MC/DC coverage criterion, we propose a set of four test criteria that are tailored to the distinct features of DNNs. Our novel criteria are incomparable and complement each other. Second, for each criterion, we give an algorithm for generating test cases based on linear programming (LP). The algorithms produce a new test case (i.e., an input to the DNN) by perturbing a given one. They encode the test requirement and a fragment of the DNN by fixing the activation pattern obtained from the given input example, and then minimize the difference between the new and the current inputs. Finally, we validate our method on a set of networks trained on the MNIST dataset. The utility of our method is shown experimentally with four objectives: (1) bug finding; (2) DNN safety statistics; (3) testing efficiency and (4) DNN internal structure analysis.

Keywords: neural networks, test criteria, test case generation

1 Introduction

Artificial intelligence, and specifically algorithms that implement deep neural networks (DNNs), can deliver human-level results in some specialist tasks such as full-information two-player games [1], image classification [2] and some areas of natural language processing [3]. There is now a prospect of a wide-scale deployment of DNNs, including in safety-critical applications such as self-driving cars. This naturally raises the question how software implementing this technology would be tested, validated and ultimately certified to meet the requirements of the relevant safety standards.

The software industry relies on testing as primary means to provide stakeholders with information about the quality of the software product or service under test [4]. Research in software engineering has resulted in a broad range of approaches to test software ([5,6,7] give comprehensive reviews). In white-box testing, the structure of a program is exploited to (perhaps automatically) generate test cases according to test requirements. Code coverage criteria (or metrics) have been designed to guide the generation of test cases, and evaluate the completeness of a test suite. E.g., a test suite with 100% statement coverage exercises all instructions at least once. While it is arguable whether this ensures correct functionality, high coverage is able to increase users' confidence (or trust) in the

program [5]. Structural coverage metrics are used as a means of assessment in a number of high-tier safety standards.

Artificial intelligence systems are typically implemented in software. This includes AI that uses DNNs, either as a monolithic end-to-end system [8] or as a system component. However, (white-box) testing for traditional software cannot be directly applied to DNNs, because the software that implements DNNs does not have suitable structure. In particular, DNNs do not have traditional flow of control and thus it is not obvious how to define criteria such as branch coverage for them.

In this paper, we bridge this gap by proposing a novel (white-box) testing methodology for DNNs, including both test coverage criteria and test case generation algorithms. Technically, DNNs contain not only an architecture, which bear some similarity with traditional software programs, but also a large set of parameters, which are calculated by the training procedure. Any approach to testing DNNs needs to consider the distinct features of DNNs, such as the syntactic connections between neurons in adjacent layers (neurons in a given layer interact with each other and then pass information to higher layers), the ReLU activation functions, and the semantic relationship between layers (e.g., neurons in deeper layers represent more complex features [9,10]).

The contributions of this paper are three-fold. First, we propose four test criteria, inspired by the MC/DC test criteria [11] from traditional software testing, that fit the distinct features of DNNs mentioned above. MC/DC was developed by NASA and has been widely adopted. It is used in avionics software development guidance to ensure adequate testing of applications with the highest criticality. There exist two coverage criteria for DNNs: neuron coverage [12] and safety coverage [13], both of which have been proposed recently. Our experiments show that neuron coverage is too coarse: 100% coverage can be achieved by a simple test suite comprised of few input vectors from the training dataset. On the other hand, safety coverage is black-box, too fine, and it is computationally too expensive to compute a test suite in reasonable time. Moreover, our four proposed criteria are incomparable with each other, and complement each other in guiding the generation of test cases.

Second, we develop an automatic test case generation algorithm for each of our criteria. The algorithms produce a new test case (i.e., an input vector) by perturbing a given one using linear programming (LP). They encode the test requirement and a fragment of the DNN by fixing the activation pattern obtained from the given input vector, and then optimize over an objective that is to minimize the difference between the new and the current input vector. LP can be solved efficiently in practice, and thus, our test case generation algorithms can generate a test suite with low computational cost.

Finally, we implement our testing approaches in a software tool named *DeepCover*[3], and validate it by conducting experiments on a set of DNNs obtained by training on the MNIST dataset. We observe that (1) the generated test suites are effective in detecting safety bugs (i.e., adversarial examples) of the DNNs; (2) our approach can provide metrics, including coverage, adversarial ratio and adversarial quality, to quantify the safety/robustness of a DNN and are shown to be efficient; (3) the testing is efficient enough and (4) internal behaviors of DNNs have been also investigated through the experiments. Overall, the method is able to handle networks of non-trivial size (>10,000

[3] Available from github link `https://github.com/theyoucheng/deepcover`

hidden neurons). This compares very favourably with current approaches based on SMT, MILP and SAT, which can only handle networks with a few hundred hidden neurons.

2 Preliminaries: Deep Neural Networks

A (feedforward and deep) neural network, or DNN, is a tuple $\mathcal{N} = (L, T, \Phi)$, where each of its elements is defined as follows.

- $L = \{L_k | k \in \{1, \ldots, K\}\}$ is a set of layers such that L_1 is the *input* layer, L_K is the *output* layer, and layers other than input and output layers are called *hidden layers*.
  - Each layer L_k consists of s_k nodes, which are also called *neurons*.
  - The l-th neuron of layer k is denoted by $n_{k,l}$.
- $T \subseteq L \times L$ is a set of connections between layers such that, except for the input and output layers, each layer has an incoming connection and an outgoing connection.
- $\Phi = \{\phi_k | k \in \{2, \ldots, K\}\}$ is a set of *activation functions* $\phi_k : D_{L_{k-1}} \to D_{L_k}$, one for each non-input layer.
  - We use $v_{k,l}$ to denote the value of $n_{k,l}$.
  - Except for inputs, every node is connected to nodes in the preceding layer by pre-defined weights such that for all k and l with $2 \leq k \leq K$ and $1 \leq l \leq s_k$

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

    where $w_{k-1,h,l}$ is the pre-trained weight for the connection between $n_{k-1,h}$ (i.e., the h-th node of layer $k-1$) and $n_{k,l}$ (i.e., the l-th node of layer k) and $\delta_{k,l}$ the the so-called *bias* for node $n_{k,l}$. We note that this definition can express both fully-connected functions and convolutional functions.
  - Values of neurons in hidden layers need to pass through a Rectified Linear Unit (ReLU) [14], where the *final activation value* of each neuron of hidden layers is defined as

$$v_{k,l} = ReLU(v_{k,l}) = \begin{cases} v_{k,l} & \text{if } v_{k,l} > 0 \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

    ReLU is by far the most popular and effective activation function for neural networks.
  - Finally, for any input, the neural network assigns a *label*, that is, the index of the node of output layer with the largest value:

$$label = \text{argmax}_{1 \leq l \leq s_K} \{v_{K,l}\} \tag{3}$$

    Let $\mathcal{L}$ be the set of labels.

Example 1. Figure 1 is a simple neural network with four layers. Its input space is $D_{L_1} = \mathbb{R}^2$ where $\mathbb{R}$ the set of real numbers.

Owing to the use of the ReLU as in (2), the behavior of a neural network is highly non-linear. Given a neuron $n_{k,l}$, its ReLU is said to be *activated* iff its value $v_{k,l}$ is strictly positive; otherwise, when $ReLU(v_{k,l}) = 0$, the ReLU is deactivated.

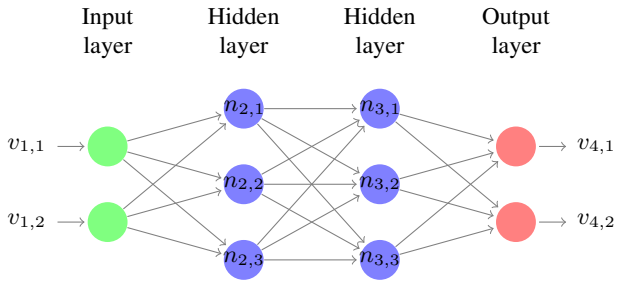


Fig. 1: A simple neural network

Neural network instance Given one particular input x, we say that the neural work $\mathcal{N}$ is *instantiated* and we use $\mathcal{N}[x]$ to denote this instance of the network.

- Given a network instance $\mathcal{N}[x]$, the activation value of each node $n_{k,l}$ of the network and the final classification label are fixed and they are denoted as $v_{k,l}[x]$ and $label[x]$ respectively.
- When the input of a neural network is given, the activation or deactivation of each ReLU operator in the network is also determined. We write $u_{k,l}[x]$ for the value before applying the ReLU and $v_{k,l}[x]$ for the value after applying the ReLU. Moreover, we write

$$sign(v_{k,l}[x]) = \begin{cases} +1 & \text{if } u_{k,l}[x] = v_{k,l}[x] \\ -1 & \text{otherwise} \end{cases} \tag{4}$$

When there is no ReLU operator for a neuron, $sign$ simply returns the sign (positive or negative) of its activation value.

Example 2. Let $\mathcal{N}$ be a network whose architecture is given in Figure 1. Assume that the weights for the first three layers are as follows

$$W_{1,2} = \begin{bmatrix} 4 & 0 & -1 \\ 1 & -2 & 1 \end{bmatrix}, \; W_{2,3} = \begin{bmatrix} 2 & 3 & -1 \\ -7 & 6 & 4 \\ 1 & -5 & 9 \end{bmatrix}$$

and that all biases are 0. When given an input $x = [0, 1]$, we get $sign(v_{2,1}[x]) = +1$, since $u_{2,1} = v_{2,1} = 1$, and $sign(v_{2,2}[x]) = -1$, since $u_{2,2} = -2 \neq 0 = v_{2,2}$.

3 Adequacy Criteria for Testing Deep Neural Networks

3.1 Test Coverage and MC/DC

Let N be a set of neural networks, R the set of requirements, and T the set of test suites.

Definition 1 ([5]). *A test adequacy criterion, or a test coverage metric, is a function* $M : \mathsf{N} \times \mathsf{R} \times \mathsf{T} \to [0, 1]$.

Intuitively, $M(\mathcal{N}, \mathcal{R}, \mathcal{T})$ quantifies the degree of adequacy to which a network $\mathcal{N}$ is tested by a test suite $\mathcal{T}$ with respect to a requirement $\mathcal{R}$. Usually, the greater the number $M(\mathcal{N}, \mathcal{R}, \mathcal{T})$, the more adequate the testing. Throughout this paper, we may use criterion and metric interchangeably. In this section, we will develop a set of test requirements $\mathcal{R}$ and their corresponding criteria $M\mathcal{R}$. The generation of test suite $\mathcal{TR}$ from the requirement $\mathcal{R}$ will be discussed in the next section.

Our new criteria for DNNs are inspired by established practices in software testing, in particular MC/DC test coverage, but are designed for the specific features of neural networks. *Modified Condition/Decision Coverage* (MC/DC) [11] is a method of ensuring adequate testing for safety-critical software. At its core is the idea that if a choice can be made, all the possible factors (conditions) that contribute to that choice (decision) must be tested. For traditional software, both conditions and the decision are usually Boolean variables or Boolean expressions. For example, the decision

$$d \iff ((a > 3) \vee (b = 0)) \wedge (c \neq 4) \tag{5}$$

contains the three conditions $(a > 3)$, $(b = 0)$ and $(c \neq 4)$. The following six test cases provide 100% MC/DC coverage:

1. $(a > 3)$=true, $(b = 0)$=true, $(c \neq 4)$=true
2. $(a > 3)$=false, $(b = 0)$=false, $(c \neq 4)$=false
3. $(a > 3)$=false, $(b = 0)$=false, $(c \neq 4)$=true
4. $(a > 3)$=false, $(b = 0)$=true, $(c \neq 4)$=true
5. $(a > 3)$=false, $(b = 0)$=true, $(c \neq 4)$=false
6. $(a > 3)$=true, $(b = 0)$=false, $(c \neq 4)$=true

The first two test cases already satisfy both the *condition coverage* (i.e., all possibilities of the conditions are exploited) and the *decision coverage* (i.e., all possibilities of the decision d are exploited). The last four cases are needed because for MC/DC each condition should evaluate to true and false at least once and should also affect the decision outcome.

3.2 Decisions and Conditions in DNNs

Our instantiation of the concepts "decision" and "condition" for DNNs is inspired by the similarity between Equation (1) and Equation (5) and the distinct features of DNNs. For example, it has been claimed that neurons in a deeper layer represent more complex features [9,10]. Therefore, the information represented by a neuron in the next layer can be seen as a *summary* (implemented by the layer function, the weights, and the bias) of the information in the current layer. *The core idea of our criteria 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.* More specifically, we consider every neuron $n_{k,l}$ for $2 \leq k \leq K$ and $1 \leq l \leq s_k$ a *decision* and say that its *conditions* are those neurons in the layer $k-1$, i.e., $\{n_{k-1,l'} \mid 1 \leq l' \leq s_{k-1}\}$.

Definition 2. *A neuron pair* $(n_{k,i}, n_{k+1,j})$ *are two neurons in adjacent layers* k *and* $k+1$ *such that* $1 \leq k \leq K-1$, $1 \leq i \leq s_k$, *and* $1 \leq j \leq s_{k+1}$. *Given a network* $\mathcal{N}$, *we write* $\mathcal{O}(\mathcal{N})$ *for the set of its neuron pairs.*

Our new criteria are obtained by giving different ways of instantiating the changes of the conditions and the decision. Unlike Boolean variables or expressions, where it is trivial to define change, i.e., true $\rightarrow$ false or false $\rightarrow$ true, for neurons, there are many different ways of defining that a decision is *affected* by the changes of the conditions. Before giving definitions for "affected" in Section 3.3, we clarify when a neuron "changes".

First, the change observed on a neuron activation value $v_{k,l}$ can be either a sign change or a value change.

Definition 3 (Sign Change). *Given a neuron $n_{k,l}$ and two test cases x_1 and x_2, we say that the sign change of $n_{k,l}$ is exploited by x_1 and x_2, denoted as $sc(n_{k,l}, x_1, x_2)$, if $sign(v_{k,l}[x_1]) \neq sign(v_{k,l}[x_2])$. We write $\neg sc(n_{k,l}, x_1, x_2)$ when the condition is not satisfied.*

Definition 4 (Value Change). *Given a neuron $n_{k,l}$ and two test cases x_1 and x_2, we say that the value change of $n_{k,l}$ is exploited with respect to a value function g by x_1 and x_2, denoted as $vc(g, n_{k,l}, x_1, x_2)$, if $g(u_{k,l}[x_1], u_{k,l}[x_2])$=true and $\neg sc(n_{k,l}, x_1, x_2)$. Moreover, we write $\neg vc(g, n_{k,l}, x_1, x_2)$ when the condition is not satisfied.*

The function g represents the change between two values $u_{k,l}[x_1]$ and $u_{k,l}[x_2]$. It can be instantiated as, e.g., $|u_{k,l}[x_1] - u_{k,l}[x_2]| \geq d$ (absolute change), or $\frac{u_{k,l}[x_1]}{u_{k,l}[x_2]} > d \vee \frac{u_{k,l}[x_1]}{u_{k,l}[x_2]} < 1/d$ (relative change), for some real number d.

Second, for the set of neurons in a layer k, we can quantify the degree of their changes in terms of *distance*.

Definition 5 (Distance Change). *Given the set of neurons $P_k = \{n_{k,l} \mid 1 \leq l \leq s_k\}$ in layer k and two test cases x_1 and x_2, we say that the distance change of P_k is exploited with respect to a distance function h by x_1 and x_2, denoted as $dc(h, k, x_1, x_2)$, if*

- $h(u_k[x_1], u_k[x_2])$ = true*, and*
- *for all $n_{k,l} \in P_k$, $sign(v_{k,l}[x_1]) = sign(v_{k,l}[x_2])$.*

We write $\neg dc(h, k, x_1, x_2)$ when any of the conditions is not satisfied.

The distance function $h(u_k[x_1], u_k[x_2])$ can be instantiated as, e.g., norm-based distances $||u_k[x_1] - u_k[x_s]||_p \leq d$ for a real number d and the distance measure L^p, or structural similarity distances, such as SSIM [15]. The distance measure L^p could be L^1 (Manhattan distance), L^2 (Euclidean distance), L^∞ (Chebyshev distance), etc.

3.3 Covering Methods

In this section, we present four covering methods for a given neuron pair and two test cases.

Definition 6 (Sign-Sign Cover, or SS Cover). *A neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is SS-covered by two test cases x_1, x_2, denoted as $cov_{SS}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$:*

- $sc(n_{k,i}, x_1, x_2)$;
- $\neg sc(n_{k,l}, x_1, x_2)$ *for all* $p_{k,l} \in P_k \setminus \{i\}$;
- $sc(n_{k+1,j}, x_1, x_2)$.

The SS Cover is designed to provide evidence that the change of a condition neuron $n_{k,l}$'s activation sign independently affects the sign of the decision neuron $n_{k+1,j}$ in the next layer. Intuitively, the first condition says that the sign change of neuron $n_{k,i}$ is exploited in x_1 and x_2. The second says that, except for $n_{k,i}$, at the k-th layer, the signs of all neurons do not change in x_1 and x_2, and the third says that the sign change of neuron $n_{k+1,j}$ is exploited in x_1 and x_2.

Example 3. (Continuation of Example 2) Given two inputs $x_1 = (0.1, 0)$ and $x_2 = (0, -1)$, we can compute the activation values for each neuron as given in Table 1. Therefore, we have $sc(n_{2,1}, x_1, x_2)$, $\neg sc(n_{2,2}, x_1, x_2)$, $\neg sc(n_{2,3}, x_1, x_2)$, and $sc(n_{3,1}, x_1, x_2)$. By Definition 6, the neuron pair $(n_{2,1}, n_{3,1})$ is SS-covered by x_1 and x_2.

input[4]	$n_{2,1}$	$n_{2,2}$	$n_{2,3}$	$n_{3,1}$	$n_{3,2}$	$n_{3,3}$
$(0.1, 0)$	0.4	0	-0.1 (0)	0.8	1.2	-0.4 (0)
$(0, -1)$	-1(0)	2	-1 (0)	-14 (0)	12	8
sign ch.	sc	¬sc	¬sc	sc	¬sc	sc
$(0, 1)$	1	-2 (0)	1	3	-2 (0)	8
$(0.1, 0.1)$	0.5	-0.2 (0)	0	1	1.5	-0.5 (0)
sign ch.	¬sc	¬sc	¬sc	¬sc	sc	sc
$(0, -1)$	-1 (0)	2	-1 (0)	-14 (0)	12	8
$(0.1, -0.1)$	0.3	0.2	-0.2 (0)	-0.8 (0)	2.1	0.5
sign ch.	sc	¬sc	¬sc	¬sc	¬sc	sc
$(0, 1)$	1	-2 (0)	1	3	-2 (0)	8
$(0.1, 0.5)$	0.9	-1 (0)	0.4	2.2	0.7	2.7
sign ch.	¬sc	¬sc	¬sc	¬sc	sc	¬sc

Table 1: Activation values and sign changes for the input examples in Examples 3, 4, 5, 6. An entry can be of the form v, in which $v \geq 0$ and ReLU is not activated, or $v_1(v_2)$, in which $v_1 < 0$, $v_2 = 0$, and the ReLU is activated, or sc, denoting that the sign has been changed, or $\neg sc$, denoting that there is no sign change.

The SS Cover is very close to MC/DC: instead of observing the change of a Boolean variable (i.e., true $\rightarrow$ false or false $\rightarrow$ true), we observe a sign change of a neuron. However, the behavior of a neural network has additional complexity. A direct adoption of the MC/DC-style coverage to a neural network does not necessarily catch all the

[4] The inputs in this table are simple numbers for the purpose of exposition; the test generation algorithms that we introduce later can find much better test pairs for the required sign or value changes.

important information in the network. The following three additional coverage criteria are designed to complement SS Cover.

First, the sign of $v_{k+1,j}$ can be altered between two test cases, even when none of the neuron values $v_{k,i}$ in layer k changes its sign.

Definition 7 (Distance-Sign Cover, or DS Cover). *Given a distance function h, a neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is DS-covered by two test cases x_1, x_2, denoted as $cov^h_{DS}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$:*

- $dc(h, k, x_1, x_2)$, *and*
- $sc(n_{k+1,j}, x_1, x_2)$.

Intuitively, the first condition describes the distance change of neurons in layer k and the second condition requests the sign change of the neuron $n_{k+1,j}$. Note that, in $dc(h, k, x_1, x_2)$, according to Definition 5, we have another condition $h(u_k[x_1], u_k[x_2])$ = true related to the distance between activation vectors $u_k[x_1]$ and $u_k[x_2]$. This is to ensure that the change in layer k is small (and therefore x_1 and x_2 have the same label): there might exist a completely different test case x_2' for x_1 in which $sc(n_{k+1,j}, x_1, x_2')$ is satisfied, but such a pair of test cases is not meaningful for testing purpose. Note that, for two pairs $(n_{k,i}, n_{k+1,j})$ and $(n_{k,i'}, n_{k+1,j})$ with the same second component $n_{k+1,j}$, the coverage of any of them implies the coverage of the other one.

Example 4. (Continuation of Example 2) Given two inputs $x_1 = (0, 1)$ and $x_2 = (0.1, 0.1)$, by the computed activation values in Table 1, we have $sc(n_{3,3}, x_1, x_2)$ and all neurons in layer 2 do not change their activation signs, i.e., $\forall i \in \{1, \ldots, 3\} : \neg sc(n_{2,i}, x_1, x_2)$. Thus, by Definition 7, x_1 and x_2 (subject to a certain distance metric h) can be used to DS-cover the neuron pairs $(n_{2,i}, n_{3,3})$ for any $i \in \{1, \ldots, 3\}$.

Until now, we have seen the sign change of a decision neuron $n_{k+1,j}$ as the equivalent of a change of a decision in MC/DC. This view may still be limited. For DNNs, a key safety problem [16] related to their high non-linearity is that an insignificant (or imperceptible) change to the input (e.g., an image) may lead to a significant change to the output (e.g., its label). We expect our criteria can provide guidance to the test case generation algorithms for discovering un-safe cases, by working with two adjacent layers, which are finer than the input-output relation. We notice that the label change in the output layer is the direct result of the changes to the activation values in the penultimate layer. Therefore, in addition to the sign change, the change of the value of the decision neuron $n_{k+1,j}$ is also significant.

Definition 8 (Sign-Value Cover, or SV Cover). *Given a value function g, a neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is SV-covered by two test cases x_1, x_2, denoted as $cov^g_{SV}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$:*

- $sc(n_{k,i}, x_1, x_2)$;
- $\neg sc(n_{k,l}, x_1, x_2)$ *for all* $p_{k,l} \in P_k \setminus \{i\}$;
- $vc(g, n_{k+1,j}, x_1, x_2)$.

The first and second conditions are the same as those in Definition 6. The difference is in the third condition, in which instead of considering the sign change $sc(n_{k+1,j}, x_1, x_2)$, we consider the value change $vc(g, n_{k+1,j}, x_1, x_2)$ with respect to a value function g. Intuitively, the SV Cover observes significant change of a decision neuron's value, by independently modifying one its condition neuron's sign.

Example 5. (Continuation of Example 2) Consider the neuron pair $(n_{2,1}, n_{3,2})$, given two inputs $x_1 = (0, -1)$ and $x_2 = (0.1, -0.1)$, by the computed activation values in Table 1, we have $sc(n_{2,1}, x_1, x_2)$, $\neg sc(n_{2,2}, x_1, x_2)$, and $\neg sc(n_{2,3}, x_1, x_2)$. If, according to the function g, $\frac{v_{3,2}[x_1]}{v_{3,2}[x_2]} \approx 5.71$ is a significant change, i.e., $g(v_{3,2}[x_1], v_{3,2}[x_2])$=true, then the pair $(n_{2,1}, n_{3,2})$ is SV-covered by x_1 and x_2.

Similarly, we have the following definition by replacing the sign change of the decision in Definition 7 with the value change.

Definition 9 (Distance-Value Cover, or DV Cover). *Given a distance function h and a value function g, a neuron pair $\alpha = (n_{k,i}, n_{k+1,j})$ is DV-covered by two test cases x_1, x_2, denoted as $cov^{h,g}_{DV}(\alpha, x_1, x_2)$, if the following conditions are satisfied by the network instances $\mathcal{N}[x_1]$ and $\mathcal{N}[x_2]$:*

- $dc(h, k, x_1, x_2)$;
- $vc(g, n_{k+1,j}, x_1, x_2)$.

Similar to DS cover, the DV cover of a pair $(n_{k,i}, n_{k+1,j})$ implies the DV cover of any other pair $(n_{k,i'}, n_{k+1,j})$ for $i' \in \{1, \ldots, s_k\}$ and $i' \neq i$. Intuitively, a DV cover targets the scenario that there is no sign change for a neuron pair, but the decision neuron's value is changed significantly.

Example 6. (Continuation of Example 2) For any $i \in \{1, \ldots, 3\}$, neurons pairs $(n_{2,i}, n_{3,3})$ are DV-covered by the inputs $x_1 = (0, 1)$ and $x_2 = (0.1, 0.5)$, subject to the functions h and g, since $\frac{v_{3,2}[x_1]}{v_{3,2}[x_2]} \approx 2.96$, $\forall i \in \{1, \ldots, 3\} : \neg sc(n_{2,i}, x_1, x_2)$, and $\neg sc(n_{3,3}, x_1, x_2)$.

3.4 Test Requirements and Criteria

We now give the requirements and criteria in Definition 1 by utilizing the covering methods defined in Section 3.3. Let $F = \{cov_{SS}, cov^{d}_{DS}, cov^{g}_{SV}, cov^{d,g}_{DV}\}$ be a set of covering methods. First of all, the test requirement is as follows.

Definition 10 (Test Requirement). *Given a network $\mathcal{N}$ and a covering method $f \in F$, a test requirement $\mathcal{R}f$ is to find a test suite $\mathcal{TR}f$ such that it covers all neuron pairs in $\mathcal{O}(\mathcal{N})$ with respect to f, i.e.,*

$$\forall \alpha \in \mathcal{O}(\mathcal{N}) \exists x_1, x_2 \in \mathcal{TR}f : f(\alpha, x_1, x_2) \tag{6}$$

Due to the one-to-one correspondence between f and $\mathcal{TR}f$, we may write $\mathcal{TR}f$ as $\mathcal{T}f$.

Intuitively, a test requirement $\mathcal{R}f$ asks that all neuron pairs α are covered by at least two test cases in $\mathcal{T}f$ with respect to the covering method f. When the requirement cannot be satisfied completely by a given test suite $\mathcal{T}$, we might want to compute the degree to which this requirement is satisfied.

Definition 11 (Test Criterion). *Given a network $\mathcal{N}$ and a covering method $f \in F$, the test criterion $M\mathcal{R}f$ for a test suite $\mathcal{T}$ is as follows:*

$$M\mathcal{R}f(\mathcal{N}, \mathcal{R}f, \mathcal{T}) = \frac{|\{\alpha \in \mathcal{O}(\mathcal{N}) \mid \exists x_1, x_2 \in T : f(\alpha, x_1, x_2)\}|}{|\mathcal{O}(\mathcal{N})|} \tag{7}$$

Due to the one-to-one correspondence between $M\mathcal{R}f(\mathcal{N}, \mathcal{R}f, \mathcal{T})$ and f, we may write $M\mathcal{R}f(\mathcal{N}, \mathcal{R}f, \mathcal{T})$ as $Mf(\mathcal{N}, \mathcal{T})$.

Intuitively, it computes the percentage of the neuron pairs that are covered by test cases in $\mathcal{T}$ with respect to the covering method f.

	sign change (Def. 3)	distance change (Def. 5)
sign change (Def. 3)	$Mcov_{SS}(\mathcal{N}, \mathcal{T})$	$Mcov^{d}_{DS}(\mathcal{N}, \mathcal{T})$
value change (Def. 4)	$Mcov^{g}_{SV}(\mathcal{N}, \mathcal{T})$	$Mcov^{d,g}_{DV}(\mathcal{N}, \mathcal{T})$

Table 2: A set of adequacy criteria for testing the DNNs, inspired by the MC/DC code coverage criteria. The columns are changes to the conditions (i.e., the activations of neurons in layer k), and the rows are changes to the decision (i.e., the activation of a neuron in layer $k+1$)

Finally, instantiating f with covering methods in F, we obtain four test criteria $Mcov_{SS}(\mathcal{N}, \mathcal{T})$, $Mcov^{d}_{DS}(\mathcal{N}, \mathcal{T})$, $Mcov^{g}_{SV}(\mathcal{N}, \mathcal{T})$, and $Mcov^{d,g}_{DV}(\mathcal{N}, \mathcal{T})$. Table 2 summaries the criteria with respect to their key ingredients, i.e., changes to the conditions and the decision.

4 Automated Test Case Generation

We introduce for each test requirement $\mathcal{R}f$ for $f \in F$ an automatic test case generation algorithm **Test-Gen**$(\mathcal{N}, \mathcal{R}f)$ to compute $\mathcal{T}f$. As we will explain later in Section 5.2, the generation of test suites for the criteria is non-trivial: a random testing approach will not achieve a test suite with high adequacy. In this paper, we consider approaches based on constraint solving. The algorithms are based on Linear Programming (LP). We remark that, as usual in conventional software testing, our test requirements/criteria are broader than any particular test case generation algorithm, and there may exist alternative algorithms for a given requirement/criterion.

Definition 12 (Activation Pattern). *Given a network $\mathcal{N}$ and an input x, the activation pattern of $\mathcal{N}[x]$ is a function $ap[\mathcal{N}, x]$, mapping from the set of hidden neurons to the signs $\{+1, -1\}$. We may write $ap[\mathcal{N}, x]$ as $ap[x]$ if $\mathcal{N}$ is clear from the context.*

For an activation pattern $ap[x]$, we use $ap[x]_{k,i}$ to denote the sign of the i-th neuron at layer k.

Example 7. (Continuation of Example 2) For input $x = (0, -1)$, by the activation values in Table 1, we have $ap[x]_{2,1} = -1$, $ap[x]_{2,2} = +1$, $ap[x]_{2,3} = -1$, $ap[x]_{2,2} = -1$, $ap[x]_{2,2} = +1$.

The function $\hat{f}$ represented by a DNN is highly non-linear and cannot be encoded with linear programming (LP) in general. Therefore, other constraint solvers such as SMT [17,18,19], MILP [20,21,19,22], SAT [23] have been considered. However, it is computationally hard, if not impossible, for such direct encodings to handle large networks, because their underlying constraint solving problems are intractable (at least NP-hard). In this paper, for the efficient generation of a test case x', we consider (1) an LP-based approach by fixing the activation pattern $ap[x]$ according to a given input x, and (2) encoding a prefix of the network, instead of the entire network, with respect to a given neuron pair.

In the following, we explain the LP encoding of a DNN instance $\mathcal{N}[x]$ (Section 4.1), introduce a few operations on the given activation pattern (Section 4.2), discuss a safety requirement (Section 4.3), and then present the algorithms based on them (Section 4.4).

4.1 LP Model of a DNN Instance

The variables used in the LP model are distinguished in **bold**. All variables are real-valued. Given an input x, the input variable $\mathbf{x}$, whose value is to be synthesized with LP, is required to have the identical activation pattern as x, i.e., $ap[\mathbf{x}] = ap[x]$.

We use $\mathbf{u_{k,i}}$ and $\mathbf{v_{k,i}}$ to denote the valuations of a neuron $n_{k,i}$ before and after the application of ReLU, respectively. Then, we have the following set $\mathcal{C}_1[x]$ of constraints

$$\begin{array}{l}\{\mathbf{u_{k,i}} \geq 0 \wedge \mathbf{v_{k,i}} = \mathbf{u_{k,i}} \mid ap[x]_{k,i} \geq 0, k \in [2, K), i \in [1..s_k]\} \\ \cup\{\mathbf{u_{k,i}} < 0 \wedge \mathbf{v_{k,i}} = 0 \mid ap[x]_{k,i} < 0, k \in [2, K), i \in [1..s_k]\}\end{array} \tag{8}$$

Moreover, the activation valuation $\mathbf{u}_{k,i}$ of each neuron is decided by the activation values $\mathbf{v}_{k-1,j}$ of those neurons in the prior layer. Therefore, we add the following set $\mathcal{C}_2[x]$ of constraints

$$\{\mathbf{u}_{k,i} = \sum_{1 \leq j \leq s_{k-1}} \{w_{k-1,j,i} \cdot \mathbf{v}_{k-1,j}\} + \delta_{k,i} \mid k \in [2, K), i \in [1..s_k]\} \tag{9}$$

Please note that the resulting LP model $\mathcal{C}[x] = \mathcal{C}_1[x] \cup \mathcal{C}_2[x]$ represents *a symbolic set of inputs* that have the identical activation pattern as x. Further, we can specify some optimization objective obj, and call an LP solver to find the optimal $\mathbf{x}$ (if one exists).

4.2 Operations on activation patterns

We define a few operations on the set $\mathcal{C}[x]$ of constraints. First,

Definition 13. *Let $\mathcal{C}[1 \ldots k']$ be a partial encoding of $\mathcal{C}$ up to layer k'. More precisely, we write $\mathcal{C}[1 \ldots k']$ if we substitute K in Expression (8) and (9) by k'.*

Note that, for each neuron $n_{k,i}$, in Expression 8, we have either $\mathbf{u}_{k,i} \geq 0 \wedge \mathbf{v}_{k,i} = \mathbf{u}_{k,i}$ or $\mathbf{u_{k,i}} < 0 \wedge \mathbf{v}_{k,i} = 0$, but not both. We write $\mathcal{C}_1[x](k,i)$ to denote the expression that is taken, and write $\mathcal{C}_1[x](k,i)_{-1}$ to denote the other expression that is not taken.

Definition 14. *Let $\mathcal{C}[(k,i)]$ be the same set of constraints as $\mathcal{C}$, except that the constraint $\mathcal{C}_1[x](k,i)$ is substituted by $\mathcal{C}_1[x](k,i)_{-1}$.*

Assume that the value function g and the distance function h are linearly encodable. Without loss of generality, we still use g and h to denote their linear encodings, respectively.

Definition 15. *Let $\mathcal{C}[+g] = \mathcal{C} \cup \{g\}$ and $\mathcal{C}[+h] = \mathcal{C} \cup \{h\}$.*

For example, when requiring a neuron $n_{k,i}$ to be activated with values bounded by $[lb_{k,i}, ub_{k,i}]$, we can use the following operation $\mathcal{C}[+(\mathbf{u_{k,i}} \in [lb_{k,i}, ub_{k,i}])]$.

Furthermore, we can generalize the above notation and define $\mathcal{C}[ops]$ for ops being a set of operations. It is obvious that the ordering of the operations does not matter if there does not exist a neuron whose value is affected by more than one operations in ops.

4.3 Safety Requirement

In this section, we discuss a *safety* requirement that is independent of the test criteria. This is to check *automatically* whether a given test case x is a bug. The neural network $\mathcal{N}$ represents a function $\hat{f}(x)$, which approximates $f(x) : D_{L_1} \rightarrow \mathcal{L}$, a function that models the human perception capability in labeling input examples. Therefore, a straightforward safety requirement is to require that for all test cases $x \in D_{L_1}$, we have $\hat{f}(x) = f(x)$. However, a direct use of such requirement is not practical because the number of inputs in D_{L_1} can be too large, if not infinite, to be labeled by human.

A pragmatic way, as done in many other works including [16,17], is to study the following safety requirement.

Definition 16 (Safety Requirement). *Given a finite set X of correctly labeled inputs, the safety requirement is to ensure that for all inputs x' that are* close *to an input $x \in X$, we have $\hat{f}(x') \neq f(x)$.*

Ideally, the *closeness* between two inputs x and x' is to be measured with respect to the human perception capability. In practice, this has been approximated by various approaches, including norm-based distance measures. In this paper, the closeness of two inputs x_1 and x_2 is concretised as the norm L^∞, i.e., $||x_1 - x_2||_\infty \leq b$ for some bound $b \in \mathbb{R}$. Specifically, we define the predicate

$$close(x_1, x_2) = ||x_1 - x_2||_\infty \leq b \tag{10}$$

We remark that our algorithms can work with any definition of closeness as long as it can be encoded as a set of linear constraints. The test generation algorithms in this paper enable the computation of x_1 and x_2 such that $||x_1 - x_2||_\infty$ can be upper bounded by a small number. A pair of inputs that satisfy the closeness definition are called *adversarial examples if only one of them is correctly labeled by the DNN*. In our experiments, to exhibit the behavior of adversarial examples, we instantiate b with a large enough number, and study the percentage of adversarial examples with respect to the number b (as illustrated in Figure 2 for one of the criteria).

4.4 Automatic Test Generation Algorithms

Our testing criteria defined in Section 3 feature that the (sign or value) change of every decision neuron must be supported by (sign or distance) changes of its condition neurons. Given a neuron pair and a testing criterion, we are going to find two activation patterns, and the two patterns together shall exhibit the changes required by the corresponding testing criterion. The inputs matching these patterns will be added to the final test suite.

At first, we define a routine to call an LP solver, which takes as input a set $\mathcal{C}[x_1]$ of constraints and an optimization objective obj, and returns a valuation x_2 for input variable **x** if one exists. For instance, a solver call can be written as follows.

$$x_2 = lp\_call(\mathcal{C}[x_1], obj) \tag{11}$$

If returned successfully, x_2 satisfies the linear constraints of $\mathcal{C}[x_1]$, with respect to the optimization objective obj.

The test suite generation framework is given as in Algorithm 1: for every $(n_{k,i}, n_{k+1,j})$ of $\mathcal{O}(\mathcal{N})$, it calls the $get\_input\_pair(f, n_{k,i}, n_{k+1,j})$ method to find an input pair that satisfies the test requirement, according to the covering method $f \in \{cov_{SS}, cov^{d}_{DS}, cov^{g}_{SV}, cov^{d,g}_{DV}\}$. We remind that $\mathcal{O}(\mathcal{N})$ is the set of neuron pairs of a DNN (see Definition 2).

Algorithm 1 Test-Gen

INPUT: DNN $\mathcal{N}$, covering method f
OUTPUT: test suite $\mathcal{T}$, adversarial examples $\mathcal{T}'$

1: **for** each neuron pair $n_{k,i}, n_{k+1,j} \in \mathcal{O}(\mathcal{N})$ **do**
2: $x_1, x_2 = get\_input\_pair(f, n_{k,i}, n_{k+1,j})$
3: **if** x_1, x_2 are valid inputs **then**
4: $\mathcal{T} \leftarrow \mathcal{T} \cup \{x_1, x_2\}$
5: **if** $\mathcal{N}[x_1].label \neq \mathcal{N}[x_2].label$ and $close(x_1, x_2)$ **then**
6: $\mathcal{T}' \leftarrow \mathcal{T}' \cup \{x_1, x_2\}$
7: **return** $\mathcal{T}, \mathcal{T}'$

To get an input pair of tests for each corresponding neuron pair, we assume the existence of a $data\_set$ of correctly labeled inputs. An input x_1 in $data\_set$ serves for the reference activation pattern, by modifying which we obtain another activation pattern such that the two together shall be able to support testing conditions specified by the covering method for a neuron pair $n_{k,i}$ and $n_{k+1,j}$.

Algorithm 2 tries to find a pair of inputs that satisfy requirements of SS Cover for a neuron pair, which specify that the sign change of $n_{k,i}$ must independently change also the sign of $n_{k+1,j}$. In particular, the LP call in Algorithm 2 comes with the constraints that a new activation pattern must share the partial encoding of x_1's activation pattern until layer k (by Definition 13), but with signs of neuron $n_{k,i}$ and $n_{k+1,j}$'s activation being negated (by Definition 14). As a matter of fact, this encoding is even stronger than the requirement in SS Cover, which does not specify constraints for neurons in

prior to layer k. The encoding in the lp_call is a compromise for efficiency, as it is computationally un-affordable to consider combinations of even a subset of neuron activations. Consequently, the LP call may fail because the proposed activation pattern is infeasible, and this explains why we need to prepare a $data\_set$ for potentially multiple LP calls.

Algorithm 2 get_input_pair with the first argument being cov_{SS}

INPUT: cov_{SS}, neuron pair $n_{k,i}, n_{k+1,j}$
OUTPUT: input pair x_1, x_2

1: **for** each $x_1 \in data\_set$ **do**
2: $x_2 = lp\_call(\mathcal{C}[x_1][1..k][\{(k,i),(k+1,j)\}], \min ||x_2 - x_1||_\infty)$
3: **if** x_2 is a valid input **then return** x_1, x_2
4: **return** _, _

The functions that find input pairs subject to the other test requirements largely follow the same structure as the SS Cover case, with the addition of distance change or value change constraints. Note that when implementing test generation for cov^d_{DS} and $cov^{d,h}_{DV}$, there is no need to go through every neuron pair, as these metrics do not require the individual change of a particular condition neuron as long as the overall behavior of all condition neurons of the decision neuron fall into certain distance change (and they do not exhibit sign change).

Algorithm 3 get_input_pair with the first argument being cov^d_{DS}

INPUT: cov^d_{DS}, neuron $n_{k,i}$
OUTPUT: input pair t_1, t_2

1: **for** each $x_1 \in data\_set$ **do**
2: $x_2 = lp\_call(\mathcal{C}[x_1][1..k-1](\{(k,i),h\}), \min ||x_2 - x_1||_\infty)$
3: **if** x_2 is a valid input **then return** x_1, x_2
4: **return** _, _

In our implementation, every time the get_input_pair is called we randomly select 40 correctly labeled inputs to construct the $data\_set$. The objective $\min ||x_2 - x_1||_\infty$ is used in all LP calls, to find good adversarial examples with respect to the safety requirement. Moreover, we use

$$g = \frac{v_{k+1,j}[x_2]}{v_{k+1,j}[x_1]} \geq \sigma \tag{12}$$

with $\sigma = 2$ for cov^g_{SV} and $\sigma = 5$ for $cov^{d,g}_{DV}$. We admit that such choices are experimental. Meanwhile, for generality and to speed up the experiments, we leave the distance function

Algorithm 4 get_input_pair with the first argument being cov^g_{SV}

INPUT: cov^g_{SV}, neuron pair $n_{k,i}, n_{k+1,j}$
OUTPUT: input pair x_1, x_2

1: **for** each $x_1 \in data\_set$ **do**
2: $x_2 = lp\_call(\mathcal{C}[x_1][1..k](\{(k,i),(k+1,j),g\}), \min ||x_2 - x_1||_\infty)$
3: **if** x_2 is a valid input **then return** x_1, x_2
4: **return** _, _

Algorithm 5 get_input_pair with the first argument being $cov^{d,g}_{DV}$

INPUT: $cov^{d,g}_{DV}$, neuron $p_{k,i}$
OUTPUT: input pair t_1, t_2

1: **for** each $x_1 \in data\_set$ **do**
2: $x_2 = lp\_call(\mathcal{C}[x_1][1..k](\{(k+1,j),h\}), \min ||x_2 - x_1||_\infty)$
3: **if** x_2 is a valid input **then return** x_1, x_2
4: **return** _, _

h unspecified. Providing a specific h may require more LP calls to find a x_2 (because h is an additional constraint) but the resulting x_2 can be better with respect to h.

4.5 $Mcov_{SS}$ and $Mcov^g_{SV}$ with Top Weights

For the two criteria $Mcov_{SS}$ and $Mcov^g_{SV}$, we need to call the function get_input_pair $|\mathcal{O}(\mathcal{N})|$ times. We note that $|\mathcal{O}(\mathcal{N})| = \sum_{k=2}^{K} s_k \cdot s_{k-1}$. Therefore, when the size of the network is large, the generation of a test suite may take a long time. To work around this, we may consider an alternative definition of $\mathcal{O}(\mathcal{N})$ by letting $(n_{k,i}, n_{k+1,j}) \in \mathcal{O}(\mathcal{N})$ only when the weight $w_{k+1,i,j}$ is one of the κ largest among $\{|w_{k+1,i',j}| \mid i' \in [1 \ldots s_k]\}$. The rationale behind is that condition neurons do not equally affect their decision, and those with higher (absolute) weights are likely to have a larger influence.

5 Experiments

We use the well-known MNIST *Handwritten Image Dataset* to train a set of 10 fully connected DNNs to perform classification. Each DNN has an input layer of $28 \times 28 = 784$ neurons and an output layer of 10 neurons. The number of hidden layers for each DNN is randomly sampled from the set of $\{3, 4, 5\}$ and at each hidden layer, the number of neurons are uniformly selected from 20 to 100. Every DNN is trained until an accuracy of at least 97.0% is reached on the MNIST validation data. Details on the structure of the DNNs are given in Table 3.

All implementations and the data discussed in this section are available online[5]. Most experiments were conducted on a MacBook with 2.5 GHz Intel Core i5 and 8 GB of memory.

5.1 Hypothesis 1: Neuron Coverage is Easy

Neuron coverage is a simple test coverage metric proposed in [12]. It was applied as one of the dual optimization objectives to compute adversarial examples for DNNs. Unfortunately, as a testing criterion, neuron coverage is inadequate. In particular, a test suite with high neuron coverage is not sufficient to increase confidence in the neural network in safety-critical domains.

To demonstrate this, for each DNN tested, we randomly pick 25 images from the MNIST test dataset. For each selected image, to maximize the neuron coverage, if an input neuron is not activated (i.e., its activation value is equal to 0), we sample its value from $[0, 0.1]$. Then we measure the neuron coverage of the DNN by using the generated test suite of 25 images. As a result, for all 10 DNNs, we obtain almost 100% neuron coverage. Our simple experiment here demonstrates that it is straight-forward to obtain a trivial test suite that has high neuron coverage but does not provide any adversarial examples.

As stated in [5], the role of test adequacy is a contributory factor in building confidence in the repeated cycle of testing, debugging, modifying program, and then testing again. The above experimental result shows that neuron coverage does not serve well in this role.

5.2 Hypothesis 2: Random Testing is Inefficient

Previous research [5] has observed that, while easy to perform, random testing does not yield high confidence in software reliability, or more formally, test adequacy is typically low. Our experiments confirm this for the specific case of DNNs.

For each DNN, we first choose an image from the MNIST test dataset, denoted by x. Subsequently, we randomly sample 10^5 inputs in the region bounded by $x \pm 0.1$, and we check whether an adversarial example exists for the original image x. Using this process, we have obtained adversarial examples for only a single one of the 10 DNNs; for the other nine DNNs, we did not observe any adversarial examples among the 10^5 randomly generated images.

More importantly, random testing does not provide information on the adequacy level achieved. Though it can be used to generate test cases for simple criteria such as neuron coverage, it is not trivial to apply random testing for complex metrics, such as the ones given in this paper, which require independent sign change or value change of particular neuron combinations.[6]

[5] Available from github link `https://github.com/theyoucheng/deepcover`

[6] However, we have no intention to discourage the use of dynamic testing for DNNs, which has been under-investigated. On the contrary, we tend to expect that a more carefully designed dynamic testing method, together with white-box techniques like algorithms in our paper, can improve the performance and efficiency for testing DNNs.

5.3 SS, SV, DS, and DV Cover

We apply the test generation algorithms given in Section 4, and we record the testing results for the SS, SV, DS, and DV covering methods. Besides the coverage Mf for each method, we also measure the percentage of adversarial examples among all test pairs in the test suite, which is denoted as AEf with $f \in \{cov_{SS}, cov^{d}_{DS}, cov^{g}_{SV}, cov^{d,g}_{DV}\}$. We further investigate the use of the "top 10 weights" simplification of SS Cover, results of which are represented by $Mcov^{w10}_{SS}$ and $AEcov^{w10}_{SS}$. For clarity, our experiments are classified into four classes: ① *bug finding* ② *DNN safety statistics* ③ *testing efficiency* ④ *DNN internal structure analysis*, and results will be labeled correspondingly.

DNN Bug finding ① The full results from our testing approaches are reported in Table 3. In our set-up, the input layer may contribute a disproportionately large number of covered neuron pairs. Thus, in Table 3, we separate them from condition neurons in the input layer. As we are going to see later, coverage of neuron pairs at front layers is easier than covering their deeper counterparts.

	hidden layers	$Mcov_{SS}$	$AEcov_{SS}$	$Mcov^{d}_{DS}$	$AEcov^{d}_{DS}$	$Mcov^{g}_{SV}$	$AEcov^{g}_{SV}$	$Mcov^{d,g}_{DV}$	$AEcov^{d,g}_{DV}$
$\mathcal{N}_1$	67x22x63	99.7%	18.9%	100%	15.8%	100%	6.7%	100%	21.1%
$\mathcal{N}_2$	59x94x56x45	98.5%	9.5%	100%	6.8%	99.9%	3.7%	100%	11.2%
$\mathcal{N}_3$	72x61x70x77	99.4%	7.1%	100%	5.0%	99.9%	3.7%	98.6%	11.0%
$\mathcal{N}_4$	65x99x87x23x31	98.4%	7.1%	100%	7.2%	99.8%	3.7%	98.4%	11.2%
$\mathcal{N}_5$	49x61x90x21x48	89.1%	11.4%	99.1%	9.6%	99.4%	4.9%	98.7%	9.1%
$\mathcal{N}_6$	97x83x32	100.0%	9.4%	100%	5.6%	100%	3.7%	100%	8.0%
$\mathcal{N}_7$	33x95x67x43x76	86.9%	8.8%	100%	7.2%	99.2%	3.8%	96%	12.0%
$\mathcal{N}_8$	78x62x73x47	99.8%	8.4%	100%	9.4%	100%	4.0%	100%	7.3%
$\mathcal{N}_9$	87x33x62	100.0%	12.0%	100%	10.5%	100%	5.0%	100%	6.7%
$\mathcal{N}_{10}$	76x55x74x98x75	86.7%	5.8%	100%	6.1%	98.3%	2.4%	93.9%	4.5%

Table 3: Results on 10 DNNs with the SS, DS, SV and SV covering methods

The results in Table 3 are promising.

- Our test generation algorithms are effective, as they reach high coverage for all covering criteria.
- The covering methods designed are useful. This is supported by the fact that a significant portion of adversarial examples have been identified.

DNN safety analysis ② The coverage Mf and adversarial percentage AEf together provide quantitative statistics to evaluate a DNN. Generally speaking, a DNN that has a high coverage (Mf) test suite with low adversarial example percentage (AEf) is considered more robust. At the same time, we can quantify the *adversarial quality*.

Because our test generation minimizes the distance when computing new input pairs (from which adversarial examples are distinguished), to evaluate the quality of obtained

	$\sum_{1<k\leq K} s_k$	$\lvert\mathcal{O}(\mathcal{N})\rvert$	$\lvert\mathcal{T}\rvert$				
			cov_{SS}	(cov_{SS}^{w10})	cov_{DS}^{d}	cov_{SV}^{g}	$cov_{DV}^{d,g}$
$\mathcal{N}_1$	162	3490	3478	(949)	95	3490	95
$\mathcal{N}_2$	264	13780	13569	(2037)	205	13779	205
$\mathcal{N}_3$	290	14822	14739	(2170)	218	14820	215
$\mathcal{N}_4$	315	18072	17779	(2409)	250	18028	246
$\mathcal{N}_5$	279	11857	10567	(2120)	228	11780	227
$\mathcal{N}_6$	222	11027	11027	(1250)	125	11027	125
$\mathcal{N}_7$	324	16409	14264	(2593)	291	16275	280
$\mathcal{N}_8$	270	13263	13235	(1917)	192	13263	192
$\mathcal{N}_9$	192	5537	5537	(1050)	105	5537	105
$\mathcal{N}_{10}$	388	23602	20459	(2806)	312	23203	293

Table 4: Test suites generated for DNNs using different criteria

adversarial examples, we can plot a distance curve to see how close the adversarial example is to the correct input. Let us take a further look at the results of SS Cover for the last three DNNs in Table 3. As illustrated in Figure 2, the horizontal axis measures the L^{∞} distance, and the vertical axis reports the accumulated percentage of adversarial examples fall into this distance. A more robust DNN will have its shape in the small distance end (towards 0) lower, as the reported adversarial examples are relatively farther from their original correct inputs. Intuitively, this means that more effort needs to be made in order to fool a DNN from correct classification to a wrong output label. Figure 3 exhibits several adversarial examples found during the testing with different distances. The value of b is used to indicate the closeness and instantiate the safety property we define in Section 4.3.

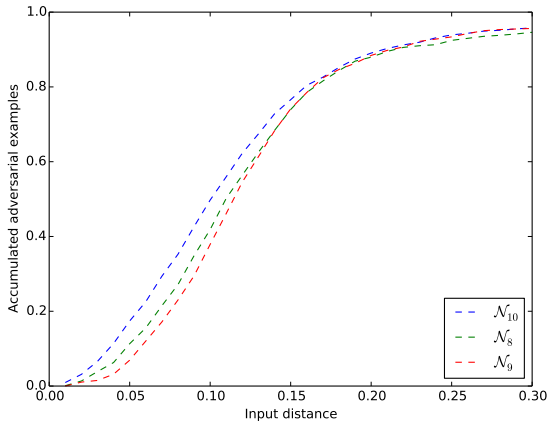
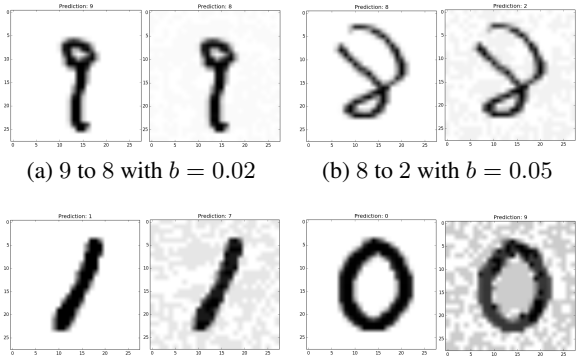


Fig. 2: SS Cover distance curves of 3 DNNs



(a) 9 to 8 with $b = 0.02$ (b) 8 to 2 with $b = 0.05$

(c) 1 to 7 with $b = 0.1$ (d) 0 to 9 with $b = 0.2$

Fig. 3: Examples of adversarial examples

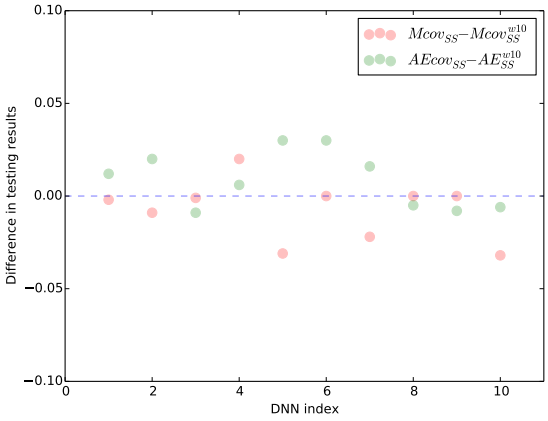
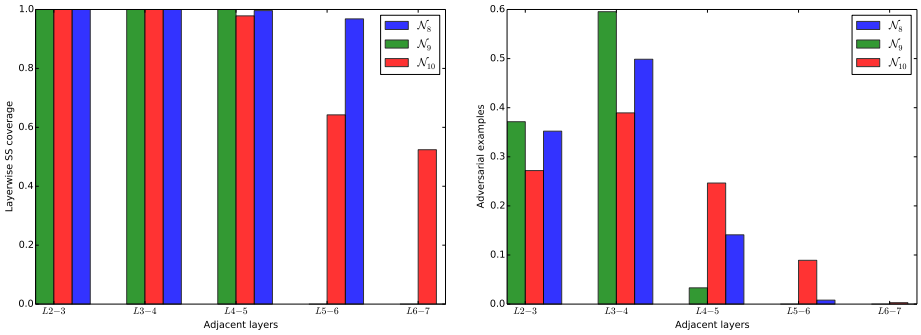


Fig. 4: SS Cover vs SS Cover with top 10 weights

SS Cover with top weights ③ Detailed information for the generated test suite for each covering method is reported in Table 4, which lists the number of neurons and neuron pairs in a DNN, and the number of test pairs inside the test suite under different criteria. As expected, covering methods that request independent sign change of condition neurons need to examine a much larger set of test cases.

Figure 4 shows the difference, on coverage and adversarial percentage, between SS Cover and its simplification for the testing of 10 DNNs. In general, the two are comparable. This is very useful in practice, as the "top weights" simplification mitigates

the size of test suite resulted, and it is thus able to behave as a faster pre-processing phase and even alternative with comparable results for SS Cover.



(a) Layerwise coverage (b) Adversarial examples at different layers

Fig. 5: SS Cover: layerwise results

Layerwise behavior ④ Through our experiments, we believe it is also worth showing that different layers of a DNN exhibit different behaviors in testing. Figure 5 reports the SS Cover results, collected in adjacent layers. In particular, Figure 5a gives the percentage of covered neuron pairs within individual adjacent layers. As we can see, when going deeper into the DNN, it can become harder for the cover of neuron pairs. Under such circumstances, to improve the coverage performance, the use of larger $data\_set$ when generating test pairs is needed. Figure 5b gives the percentage of adversarial examples found at different layers. Interestingly, it seems that most adversarial examples be found around the middle layers of all DNNs tested.

	$\mathcal{N}_8$			$\mathcal{N}_9$			$\mathcal{N}_{10}$		
	#vars	$\|\mathcal{C}\|$	t	#vars	$\|\mathcal{C}\|$	t	#vars	$\|\mathcal{C}\|$	t
$L2$-3	864	3294	0.58	873	3312	0.57	862	3290	0.49
$L3$-4	926	3418	0.84	906	3378	0.61	917	3400	0.71
$L4$-5	999	3564	0.87	968	3502	0.86	991	3548	0.75
$L5$-6	1046	3658	0.91	–	–	–	1089	3744	0.82
$L6$-7	–	–	–	–	–	–	1164	3894	0.94

Table 5: Number of variables and constraints, and time cost of each LP call in test generation

Cost of LP call ③ Since LP encoding of the DNN (partial) activation pattern plays a key part in our test generation, in this part we give details of the LP call cost, even

though linear programming is widely accepted as an efficient method. For every DNN, we select a set of neuron pairs, where each decision neuron is at a different layer. Then, we measure the number of variables and constraints, and the time t in seconds (averaged over 100 runs) spent on solving each LP call. Results in Table 5 confirm that the LP model of a partial activation pattern is indeed lightweight, and its complexity increases in the linear manner as traversing into deeper layers of a DNN.

5.4 Convolutional Neural Networks ④

The testing approach in this paper can be generalized to Convolutional Neural Networks (CNNs). In a CNN, a convolutional layer consists of multiple *feature maps*. The weights and bias that define a feature map are often said to be *filters*. Given a decision neuron inside some feature map, its activation is computed by a subset of precedent neurons that are processed by the associated filter. To investigate the testing of CNNs, we trained two networks $\mathcal{N}_1^c$ and $\mathcal{N}_2^c$, each having two convolutional layers followed by one fully connected layer of 128 neurons. $\mathcal{N}_1^c$ (resp. $\mathcal{N}_2^c$) has 20 (resp. 2) filters for the first convolutional layer and 40 (resp. 4) filters for the second convolutional layer. The filters have size 5×5 and every convolutional layer is augmented with a so-called max-pooling layer of size 2×2. Overall, $\mathcal{N}_1^c$ has 14208 hidden neurons and $\mathcal{N}_2^c$ is much smaller with 1536 hidden neurons. This is designed on purpose to conduct our testing approaches with different scalabilities.

The emphasis of this part is on understanding the impact of filters in CNN through testing, and results of SS Cover are analysed at feature map level. We use $f_{k,i}$ to denote the i-th feature map in the k-th convolutional layer, and $Mcov_{SS}{}^{f_{k,i}}_{f_{k+1,j}}$ represents the SS coverage among neuron pairs such that the condition neuron and decision neuron are from $f_{k,i}$ and $f_{k+1,j}$ respectively. Correspondingly, $AEcov_{SS}{}^{f_{k,i}}_{f_{i+1,j}}$ is for the adversarial percentage. Results in Table 6 show the detailed testing information on several feature maps of the two CNNs. The use of extra filters is justified by our testing, as a much smaller number of adversarial examples were found for $\mathcal{N}_1^c$.

	$Mcov_{SS}{}^{f_{1,1}}_{f_{2,1}}$	$AEcov_{SS}{}^{f_{1,1}}_{f_{2,1}}$	$Mcov_{SS}{}^{f_{1,1}}_{f_{2,2}}$	$AEcov_{SS}{}^{f_{1,1}}_{f_{2,2}}$
$\mathcal{N}_1^c$	99.7%	1.6%	99.7%	1.5%
$\mathcal{N}_2^c$	100%	7.6%	100%	6.8%

Table 6: Feature map wise SS Cover results

6 Related Work

For traditional software, there is a broad range of test coverage criteria defined either at the syntactic level (or code-level), such as statement coverage, branch coverage, MC/DC coverage, path coverage, etc. [5,6,7], or semantic level (or model level), such as state

and transition coverage [24,25]. In the following, we briefly review existing work on how to validate safety properties of DNNs.

Existing Test Coverage Metrics Up to now, there are two proposals for test coverage criteria. In [12], *neuron coverage* is proposed, and it is applied in [26] to guide the testing of DNN-driven autonomous cars. Given an input x, we can observe the activations of the hidden neurons and compute the set $ReLU(x)$ of hidden neurons whose ReLU activation function is activated. Let P_H be the set of hidden neurons. Given a test suite $\mathcal{T}$, its neuron coverage is

$$Cov_{neuron}(X) = |\bigcup\{ReLU(x) \mid x \in \mathcal{T}\}|/|P_H| \tag{13}$$

While neuron coverage bears some similarity with statement coverage, as we explained in the paper, it is a very coarse criterion: it is easy to find a test suite that achieves 100% neuron coverage but the network still allows trivial adversarial examples. Thus, neuron coverage does not even satisfy the expectations we usually have for a test suite with high statement coverage for conventional software [12].

In [13], the input space is discretised with τ-hyper-rectangles, and then one test case is generated for each hyper-rectangle. It is shown that, if τ is small enough (precisely $\tau < 2\ell/\hbar$ for ℓ the minimum confidence gap and $\hbar$ the Lipschitz constant of the network), then this testing approach can provide a guarantee, i.e., it can find all adversarial examples, and is able to claim safety when no adversarial example can be found. Thus, safety coverage is a strong criterion, but the generation of a test suite can be very expensive; therefore, [13] uses a Monte-Carlo Tree Search based approach to generate test cases. Assume that we have a test suite $\mathcal{T}$ and for each $x \in \mathcal{T}$ it is associated with a hyper-rectangle η_x, then we have the safety coverage defined as follows:

$$Cov_{safety}(X) = ||\bigcup\{\eta_x \mid x \in \mathcal{T}\}||/||\eta|| \tag{14}$$

where $||\eta||$ measures the size of the region η.

Adversarial Example Generation of DNNs Most existing work, e.g., [16,27,28,29,30], applies various heuristic algorithms, generally using search algorithms based on gradient descent or evolutionary techniques. [31] construct a saliency map of the importance of the pixels based on gradient descent and then modify the pixels. These approaches may be able to find adversarial examples efficiently, but are *not able to provide any guarantee* (like verification) or *any certain level of confidence* (like testing) about the nonexistence of adversarial examples when the algorithm fails to find one.

Automated Verification of DNNs There are two ways to perform safety verification for DNNs. The first is to reduce the problem into a constraint solving problem. Notable work includes, e.g., [32,18]. Moreover, [20] determines whether an output value of a DNN is reachable from a given input subspace, and reduce the problem to a MILP problem. [33] considers the range of output values from a given input subspace. Their approach interleaves local search (based on gradient descent) with global search (based on reduction to MILP). These approaches can only work with small networks with a few hundred hidden neurons.

The second approach is to discretise the vector spaces of the input or hidden layers and then to apply exhaustive search algorithms or Monte-Carlo tree search algorithm on the discretised spaces. The guarantees are achieved by establishing local assumptions such as minimality of manipulations in [17] and minimum confidence gap for Lipschitz networks in [13]. This approach has been shown to be applicable to state-of-the-art networks.

7 Conclusions

We have proposed a set of novel test criteria for DNNs, developped for each criterion an algorithm for test suite generation, and implemented the algorithms in a software tool. Our experiments on a set of DNNs trained on the MNIST data set show promising results, indicating the effectiveness of both the test criteria and the test case generation algorithms for testing the DNNs. Future work includes even more efficient test case generation algorithms that do not require linear programming.

References

1. David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of Go without human knowledge. *Nature*, 550(354–359), 2017.
2. Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, and Li Fei-Fei. ImageNet large scale visual recognition challenge. *International Journal of Computer Vision*, 115(3):211–252, Dec 2015.
3. Yoav Goldberg. A primer on neural network models for natural language processing. *Journal of Artificial Intelligence Research*, 57(345–420), 2016.
4. Cem Kaner. Exploratory testing. In *Quality Assurance Institute Worldwide Annual Software Testing Conference*, 2006.
5. Hong Zhu, Patrick AV Hall, and John HR May. Software unit test coverage and adequacy. *ACM Computing Surveys*, 29(4):366–427, 1997.
6. Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, 37(5):649–678, 2011.
7. Ting Su, Ke Wu, Weikai Miao, Geguang Pu, Jifeng He, Yuting Chen, and Zhendong Su. A survey on data-flow testing. *ACM Computing Surveys*, 50(1):5:1–5:35, March 2017.
8. Mariusz Bojarski, Davide Del Testa, Daniel Dworakowski, Bernhard Firner, Beat Flepp, Prasoon Goyal, Lawrence D. Jackel, Mathew Monfort, Urs Muller, Jiakai Zhang, Xin Zhang, Jake Zhao, and Karol Zieba. End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*, 2016.
9. Jason Yosinski, Jeff Clune, Anh Nguyen, Thomas Fuchs, and Hod Lipson. Understanding neural networks through deep visualization. *arXiv preprint arXiv:1506.06579*, 2015.
10. Chris Olah, Arvind Satyanarayan, Ian Johnson, Shan Carter, Ludwig Schubert, Katherine Ye, and Alexander Mordvintsev. The building blocks of interpretability. *Distill*, 2018. https://distill.pub/2018/building-blocks.
11. Kelly Hayhurst, Dan Veerhusen, John Chilenski, and Leanna Rierson. A practical tutorial on modified condition/decision coverage. Technical report, NASA, 2001.

12. Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. DeepXplore: Automated whitebox testing of deep learning systems. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 1–18. ACM, 2017.
13. Matthew Wicker, Xiaowei Huang, and Marta 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, arXiv preprint arXiv:1710.07859*, 2018.
14. Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted Boltzmann machines. In *Proceedings of the 27th International Conference on Machine Learning (ICML)*, pages 807–814, 2010.
15. Zhou Wang, Eero P Simoncelli, and Alan C Bovik. Multiscale structural similarity for image quality assessment. In *Signals, Systems and Computers, Conference Record of the Thirty-Seventh Asilomar Conference on*, 2003.
16. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. In *International Conference on Learning Representations (ICLR)*, 2014.
17. Xiaowei Huang, Marta Kwiatkowska, Sen Wang, and Min Wu. Safety verification of deep neural networks. In *International Conference on Computer Aided Verification*, pages 3–29. Springer, 2017.
18. Guy Katz, Clark Barrett, David L Dill, Kyle Julian, and Mykel J Kochenderfer. Reluplex: An efficient SMT solver for verifying deep neural networks. In *International Conference on Computer Aided Verification*, pages 97–117. Springer, 2017.
19. Rudy Bunel, Ilker Turkaslan, Philip HS Torr, Pushmeet Kohli, and M Pawan Kumar. Piecewise linear neural network verification: A comparative study. *arXiv preprint arXiv:1711.00455*, 2017.
20. Alessio Lomuscio and Lalit Maganti. An approach to reachability analysis for feed-forward ReLU neural networks. *arXiv preprint arXiv:1706.07351*, 2017.
21. Chih-Hong Cheng, Georg Nührenberg, and Harald Ruess. Maximum resilience of artificial neural networks. In Deepak D'Souza and K. Narayan Kumar, editors, *Automated Technology for Verification and Analysis*, pages 251–268. Springer, 2017.
22. Weiming Xiang, Hoang-Dung Tran, and Taylor T Johnson. Output reachable set estimation and verification for multi-layer neural networks. *arXiv preprint arXiv:1708.03322*, 2017.
23. Nina Narodytska, Shiva Prasad Kasiviswanathan, Leonid Ryzhyk, Mooly Sagiv, and Toby Walsh. Verifying properties of binarized deep neural networks. *arXiv preprint arXiv:1709.06662*, 2017.
24. Paul Ammann and Jeff Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.
25. Arilo C Dias Neto, Rajesh Subramanyan, Marlon Vieira, and Guilherme H Travassos. A survey on model based testing approaches: A systematic review. In *1st ACM International Workshop on Empirical Assessment of Software Engineering Language and Technologies*, pages 31–36, 2007.
26. Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. DeepTest: Automated testing of deep-neural-network-driven autonomous cars. *arXiv preprint arXiv:1708.08559*, 2017.
27. Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.
28. Anh Nguyen, Jason Yosinski, and Jeff Clune. Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436, 2015.
29. Seyed-Mohsen Moosavi-Dezfooli, Alhussein Fawzi, Omar Fawzi, and Pascal Frossard. Universal adversarial perturbations. *arXiv preprint arXiv:1610.08401*, 2016.
30. Nicholas Carlini and David Wagner. Towards evaluating the robustness of neural networks. In *Security and Privacy (SP), IEEE Symposium on*, pages 39–57, 2017.

31. Nicolas Papernot, Patrick McDaniel, Somesh Jha, Matt Fredrikson, Z Berkay Celik, and Ananthram Swami. The limitations of deep learning in adversarial settings. In *Security and Privacy (EuroS&P), IEEE European Symposium on*, pages 372–387, 2016.
32. Luca Pulina and Armando Tacchella. An abstraction-refinement approach to verification of artificial neural networks. In *International Conference on Computer Aided Verification*, pages 243–257. Springer, 2010.
33. Souradeep Dutta, Susmit Jha, Sriram Sanakaranarayanan, and Ashish Tiwari. Output range analysis for deep neural networks. *arXiv preprint arXiv:1709.09130*, 2017.