DeepFault: Fault Localization for Deep Neural Networks*

Hasan Ferit Eniser[1][0000−0002−3259−8794], Simos Gerasimou[2][0000−0002−2706−5272], and Alper Sen[1][0000−0002−5508−6484]

[1] Bogazici University, Istanbul, Turkey
{hasan.eniser,alper.sen}@boun.edu.tr
[2] University of York, York, UK
simos.gerasimou@york.ac.uk

Abstract. Deep Neural Networks (DNNs) are increasingly deployed in safety-critical applications including autonomous vehicles and medical diagnostics. To reduce the residual risk for unexpected DNN behaviour and provide evidence for their trustworthy operation, DNNs should be thoroughly tested. The DeepFault whitebox DNN testing approach presented in our paper addresses this challenge by employing suspiciousness measures inspired by fault localization to establish the hit spectrum of neurons and identify suspicious neurons whose weights have not been calibrated correctly and thus are considered responsible for inadequate DNN performance. DeepFault also uses a suspiciousness-guided algorithm to synthesize new inputs, from correctly classified inputs, that increase the activation values of suspicious neurons. Our empirical evaluation on several DNN instances trained on MNIST and CIFAR-10 datasets shows that DeepFault is effective in identifying suspicious neurons. Also, the inputs synthesized by DeepFault closely resemble the original inputs, exercise the identified suspicious neurons and are highly adversarial.

Keywords: Deep Neural Networks · Fault Localization · Test Input Generation

1 Introduction

Deep Neural Networks (DNNs) [33] have demonstrated human-level capabilities in several intractable machine learning tasks including image classification [10], natural language processing [56] and speech recognition [19]. These impressive achievements raised the expectations for deploying DNNs in real-world applications, especially in safety-critical domains. Early-stage applications include air traffic control [25], medical diagnostics [34] and autonomous vehicles [5]. The responsibilities of DNNs in these applications vary from carrying out well-defined tasks (e.g., detecting abnormal network activity [11]) to controlling the entire behaviour system (e.g., end-to-end learning in autonomous vehicles [5]).

Despite the anticipated benefits from a widespread adoption of DNNs, their deployment in safety-critical systems must be characterized by a high degree of dependability. Deviations from the expected behaviour or correct operation, as expected in safety-critical domains, can endanger human lives or cause significant financial loss. Arguably, DNN-based systems should be granted permission for use in the public domain only after exhibiting high levels of trustworthiness [6].

* This research was supported in part by Bogazici University; Research Fund 13662.

Software testing is the de facto instrument for analysing and evaluating the quality of a software system [24]. Testing enables at one hand to reduce the risk by proactively finding and eliminating problems (*bugs*), and on the other hand to evidence, through using the testing results, that the system actually achieves the required levels of safety. Research contributions and advice on best practices for testing conventional software systems are plentiful; [63], for instance, provides a comprehensive review of the state-of-the-art testing approaches.

Nevertheless, there are significant challenges in applying traditional software testing techniques for assessing the quality of DNN-based software [54]. Most importantly, the little correlation between the behaviour of a DNN and the software used for its implementation means that the behaviour of the DNN cannot be explicitly encoded in the control flow structures of the software [51]. Furthermore, DNNs have very complex architectures, typically comprising thousand or millions of parameters, making it difficult, if not impossible, to determine a parameter's contribution to achieving a task. Likewise, since the behaviour of a DNN is heavily influenced by the data used during training, collecting enough data that enables exercising all potential DNN behaviour under all possible scenarios becomes a very challenging task. Hence, there is a need for systematic and effective testing frameworks for evaluating the quality of DNN-based software [6].

Recent research in the DNN testing area introduces novel white-box and black-box techniques for testing DNNs [20,28,36,37,48,54,55]. Some techniques transform valid training data into adversarial through mutation-based heuristics [65], apply symbolic execution [15], combinatorial [37] or concolic testing [55], while others propose new DNN-specific coverage criteria, e.g., neuron coverage [48] and its variants [35] or MC/DC-inspired criteria [52]. We review related work in Section 6. These recent advances provide evidence that, while traditional software testing techniques are not directly applicable to testing DNNs, the sophisticated concepts and principles behind these techniques, if adapted appropriately, could be useful to the machine learning domain. Nevertheless, none of the proposed techniques uses *fault localisation* [4,47,63], which can identify parts of a system that are most responsible for incorrect behaviour.

In this paper, we introduce *DeepFault*, the first fault localization-based white-box testing approach for DNNs. The objectives of DeepFault are twofold: (i) *identification* of *suspicious* neurons, i.e., neurons likely to be more responsible for incorrect DNN behaviour; and (ii) *synthesis* of new inputs, using correctly classified inputs, that exercise the identified suspicious neurons. Similar to conventional fault localization, which receives as input a faulty software and outputs a ranked list of suspicious code locations where the software may be defective [63], DeepFault *analyzes* the behaviour of neurons of a DNN after training to establish their hit spectrum and *identifies* suspicious neurons by employing suspiciousness measures. DeepFault employs a suspiciousness-guided algorithm to *synthesize* new inputs, that achieve high activation values for suspicious neurons, by modifying correctly classified inputs. Our empirical evaluation on the popular publicly available datasets MNIST [32] and CIFAR-10 [1] provides evidence that DeepFault can identify neurons which can be held responsible for insufficient network performance. DeepFault can also synthesize new inputs, which closely resemble the original inputs, are highly adversarial and increase the activation values of the identified suspicious neurons. To the best of our knowledge, DeepFault is the first research attempt that introduces *fault localization* for DNNs to identify suspicious neurons and synthesize new, likely adversarial, inputs.

Overall, the main contributions of this paper are:

– The DeepFault approach for whitebox testing of DNNs driven by fault localization;
– An algorithm for identifying suspicious neurons that adapts suspiciousness measures from the domain of spectrum-based fault localization;
– A suspiciousness-guided algorithm to synthesize inputs that achieve high activation values of potentially suspicious neurons;
– A comprehensive evaluation of DeepFault on two public datasets (MNIST and CIFAR-10) demonstrating its feasibility and effectiveness;

The reminder of the paper is structured as follows. Section 2 presents briefly DNNs and fault localization in traditional software testing. Section 3 introduces *DeepFault* and Section 4 presents its open-source implementation. Section 5 describes the experimental setup, research questions and evaluation carried out. Sections 6 and 7 discuss related work and conclude the paper, respectively.

2 Background

2.1 Deep Neural Networks

We consider Deep Learning software systems in which one or more system modules is controlled by DNNs [13]. A typical feed-forward DNN comprises multiple interconnected neurons organised into several layers: the *input* layer, the *output* layer and at least one *hidden* layer (Fig. 1). Each DNN layer comprises a sequence of neurons. A *neuron* denotes a computing unit that applies a *nonlinear activation function* to its inputs and transmits the result to neurons in the successive layer. Commonly used activation functions are sigmoid, hyperbolic tangent, ReLU (Rectified Linear Unit) and leaky ReLU [13]. Except from the input layer, every neuron is connected to neurons in the successive layer with *weights*, i.e., edges, whose values signify the strength of a connection between neuron pairs. Once the DNN architecture is defined, i.e., the number of layers, neurons per layer and activation functions, the DNN undergoes a *training process* using a large amount of labelled training data to find weight values that minimise a *cost function*.

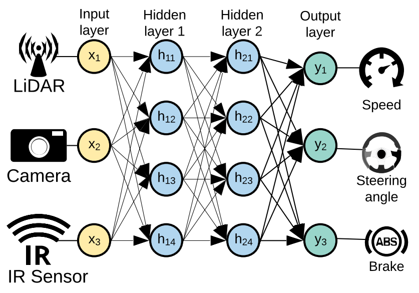

Fig. 1. A four layer fully-connected DNN that receives inputs from vehicle sensors (camera, LiDAR, infrared) and outputs a decision for speed, steering angle and brake.

In general, a DNN could be considered as a parametric multidimensional function that consumes input data (e.g, raw image pixels) in its input layer, extracts *features*, i.e., semantic concepts, by performing a series of nonlinear transformations in its *hidden layers*, and, finally, produces a decision that matches the effect of these computations in its *output layer*.

2.2 Software Fault Localization

Fault localization (FL) is a white box testing technique that focuses on identifying source code elements (e.g., statements, declarations) that are more likely to contain faults. The general FL process [63] for traditional software (Fig. 5) uses as inputs a program P, corresponding to the system under test, and a test suite T, and employs an FL technique to test P against T and establish subsets that represent the passed and failed tests. Using these sets and information regarding program elements $p \in P$, the FL technique extracts fault localization data which is then employed by an FL measure to establish the "suspiciousness" of each program element p. Spectrum-based FL, the most studied class of FL techniques, uses program traces (called program spectra) of successful and failed test executions to establish for program element p the tuple (e_s,e_f,n_s,n_f). Members e_s and e_f (n_s and n_f) represent the number of times the corresponding program element has been (has not been) executed by tests, with success and fail, respectively. A spectrum-based FL measure consumes this list of tuples and ranks the program elements in decreasing order of suspiciousness enabling software engineers to inspect program elements and find faults effectively. For a comprehensive survey of state-of-the-art FL techniques, see [63].

3 DeepFault

In this section, we introduce our DeepFault whitebox approach that enables to systematically test DNNs by identifying and localizing highly erroneous neurons across a DNN. Given a pre-trained DNN, DeepFault, whose workflow is shown in Figure 2, performs a series of *analysis*, *identification* and *synthesis* steps to identify highly erroneous DNN neurons and synthesize new inputs that exercise erroneous neurons. We describe the DeepFault steps in Sections 3.1–3.3.

We use the following notations to describe DeepFault. Let $\mathcal{N}$ be a DNN with l layers. Each layer $L_i, 1 \leq i \leq l$, consists of s_i neurons and the total number of neurons in $\mathcal{N}$ is given by $s = \sum_{i=1}^{l} s_i$. Let also $n_{i,j}$ be the j-th neuron in the i-th layer. When the context is clear, we use $n \in \mathcal{N}$ to denote any neuron which is part of the DNN $\mathcal{N}$ irrespective of its layer. Likewise, we use N_H to denote the neurons which belong to the hidden layers of N, i.e., $N_H = \{n_{ij} | 1 < i < l, 1 \leq j \leq s_j\}$. We use $\mathcal{T}$ to denote the set of test inputs from the input domain of $\mathcal{N}$, $t \in \mathcal{T}$ to denote a concrete input, and $u \in t$ for an element of t. Finally, we use the function $\phi(t, n)$ to signify the output of the activation function of neuron $n \in \mathcal{N}$.

3.1 Neuron Spectrum Analysis

The first step of DeepFault involves the analysis of neurons within a DNN to establish suitable neuron-based attributes that will drive the detection and localization of faulty neurons. As highlighted in recent research [18, 48], the adoption of whitebox testing techniques provides additional useful insights regarding internal neuron activity and network behaviour. These insights cannot be easily extracted through black-box DNN testing, i.e., assessing the performance of a DNN considering only the decisions made given a set of test inputs $\mathcal{T}$.

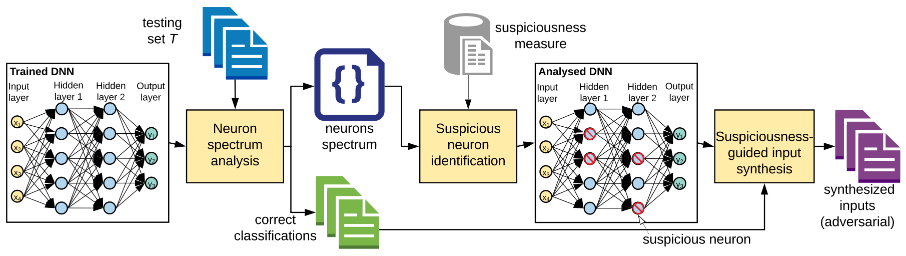

Fig. 2. DeepFault workflow.

DeepFault initiates the identification of suspicious neurons by establishing attributes that capture a neuron's execution pattern. These attributes are defined as follows. Attributes $attr_n^{\mathrm{as}}$ and $attr_n^{\mathrm{af}}$ signify the number of times neuron n was active (i.e., the result of the activation function $\phi(t, n)$ was above the predefined threshold) and the network made a successful or failed decision, respectively. Similarly, attributes $attr_n^{\mathrm{ns}}$ and $attr_n^{\mathrm{nf}}$ cover the case in which neuron n is not active. DeepFault analyses the behaviour of neurons in the DNN hidden layers, under a specific test set $\mathcal{T}$, to assemble a *Hit Spectrum (HS)* for each neuron, i.e., a tuple describing its dynamic behaviour. We define formally the HS as follows.

Definition 1. *Given a DNN $\mathcal{N}$ and a test set $\mathcal{T}$, we say that for any neuron $n \in \mathcal{N}_H$ its hit spectrum is given by the tuple $HS_n = (attr_n^{as}, attr_n^{af}, attr_n^{ns}, attr_n^{nf})$.*

Note that the sum of each neuron's HS should be equal to the size of $\mathcal{T}$.

Clearly, the interpretation of a hit spectrum (cf. Def. 1) is meaningful only for neurons in the hidden layers of a DNN. Since neurons within the input layer L_1 correspond to elements from the input domain (e.g., pixels from an image captured by a camera in Fig. 1), we consider them to be "correct-by-construction". Hence, these neurons cannot be credited or held responsible for a successful or failed decision made by the network. Furthermore, input neurons are always active and thus propagate one way or another their values to neurons in the following layer. Likewise, neurons within the output layer L_l simply aggregate values from neurons in the penultimate layer L_{l-1}, multiplied by the corresponding weights, and thus have limited influence in the overall network behaviour and, accordingly, to decision making.

3.2 Suspicious Neurons Identification

During this step, DeepFault consumes the set of hit spectrums, derived from DNN analysis, and identifies *suspicious* neurons which are likely to have made significant contributions in achieving inadequate DNN performance (low accuracy/high loss). To achieve this identification, DeepFault employs a spectrum-based suspiciousness measure which computes a suspiciousness score per neuron using spectrum-related information. Neurons with the highest suspiciousness score are more likely to have been trained unsatisfactorily and, hence, contributing more to incorrect DNN decisions. This indicates that the weights of these neurons need further calibration [13]. We define neuron suspiciousness as follows.

Table 1. Suspiciousness measures used in DeepFault

Suspiciousness Measure	**Algebraic Formula**
Tarantula [23]:	$\frac{attr_n^{\mathrm{af}}/(attr_n^{\mathrm{af}}+attr_n^{\mathrm{nf}})}{attr_n^{\mathrm{af}}/(attr_n^{\mathrm{af}}+attr_n^{\mathrm{nf}})+attr_n^{\mathrm{as}}/(attr_n^{\mathrm{as}}+attr_n^{\mathrm{ns}})}$
Ochiai [42]:	$\frac{attr_n^{\mathrm{af}}}{\sqrt{(attr_n^{\mathrm{af}}+attr_n^{\mathrm{nf}})\cdot(attr_n^{\mathrm{af}}+attr_n^{\mathrm{as}})}}$
D* [62]:	$\frac{{attr_n^{\mathrm{af}}}^{*}}{attr_n^{\mathrm{as}}+attr_n^{\mathrm{nf}}}$

$* > 0$ is a variable. We used $* = 3$, among the most widely explore values [47, 63].

Algorithm 1 Identification of suspicious neurons

```
1: function SuspiciousNeuronsIdentification($\mathcal{N}, \mathcal{T}, k$)
2:     $S \leftarrow \emptyset$                                  ▷ suspiciousness vector
3:     for all $n \in N$ do
4:         $HS_n \leftarrow \emptyset$                           ▷ $n$-th neuron hit spectrum vector
5:         for all $p \in \{as, af, ns, nf\}$ do
6:             $a_n^p$ =attr($\mathcal{T}, p$)                   ▷ establish attribute for property $p$
7:             $HS_n = HS_n \cup \{a_n^p\}$                      ▷ construct hit spectrum (cf. Def. 1)
8:         $S = S \cup \{$Susp$(HS_n)\}$                         ▷ determine neuron suspiciousness (cf. Def. 2)
9:     SN = $\{n |$Susp$(HS_n) \in$ Select$(S, k)\}$            ▷ select the $k$ most suspicious neurons
10:    return SN
```

Definition 2. *Given a neuron $n \in \mathcal{N}_H$ with HS_n being its hit spectrum, a neuron's spectrum-based suspiciousness is given by the function* $\mathrm{Susp}_n : HS_n \rightarrow \mathbb{R}$.

Intuitively, a suspiciousness measure facilitates the derivation of correlations between a neuron's behaviour given a test set $\mathcal{T}$ and the failure pattern of $\mathcal{T}$ as determined by the overall network behaviour. Neurons whose behaviour pattern is *close* to the failure pattern of $\mathcal{T}$ are more likely to operate unreliably, and consequently, they should be assigned higher suspiciousness. Likewise, neurons whose behaviour pattern is *dissimilar* to the failure pattern of $\mathcal{T}$ are considered more trustworthy and their suspiciousness values should be low.

In this paper, we instantiate DeepFault with three different suspiciousness measures, i.e., Tarantula [23], Ochiai [42] and D* [62] whose algebraic formulae are shown in Table 1. The general principle underlying these suspiciousness measures is that the more often a neuron is activated by test inputs for which the DNN made an incorrect decision, and the less often the neuron is activated by test inputs for which the DNN made a correct decision, the more suspicious the neuron is. These suspiciousness measures have been adapted from the domain of fault localization in software engineering [63] in which they have achieved competitive results in automated software debugging by isolating the root causes of software failures while reducing human input. To the best of our knowledge, DeepFault is the first approach that proposes to incorporate these suspiciousness measures into the DNN domain for the identification of defective neurons.

The use of suspiciousness measures in DNNs targets the identification of a set of defective neurons rather than diagnosing an isolated defective neuron. Since the output of a DNN decision task is typically based on the aggregated effects of its neurons (computation units), with each neuron making its own contribution

Algorithm 2 New input synthesis guided by the identified suspicious neurons

```
Input: $SN \leftarrow$ suspicious neurons (Algorithm 1), $step \leftarrow$ step size in gradient ascent
$T_s \leftarrow$ test inputs correctly classified by $\mathcal{N}$, $d \leftarrow$ new inputs maximum allowed distance
1: function SuspiciousnessGuidedInputSynthesis($SN, \mathcal{T}_s, d, step$)
2:     $NT \leftarrow \emptyset$                              ▷ set of synthesized inputs
3:     for all $t \in T_s$ do
4:         $G_t \leftarrow \emptyset$                         ▷ gradient collection of suspicious neurons
5:         for all $n \in SN$ do
6:             $n^v = \phi(t, n)$                             ▷ determine output of neuron
7:             $G = \partial n^v / \partial t$                ▷ establish gradient of neuron for $t$
8:             $G_t = G_t \cup \{G\}$                         ▷ collect gradients of suspicious neurons for $t$
9:         $t' \leftarrow \emptyset$                          ▷ initialisation of input to be synthesised
10:        for all $u \in t$ do
11:            $u_{gradient} = \sum_{G \in G_t} G / |G_t|$    ▷ determine average gradient of $u$
12:            $u_{gradient} =$ GradientConstraint($u_{gradient}, d, step$)
13:            $t' = t' \frown \{$DomainConstraints($u + u_{gradient}$)$\}$
14:        $NT = NT \cup \{t'\}$
15:    return $NT$
```

to the whole computation procedure [13], identifying a single point of failure (i.e., a single defective neuron) has limited value. Thus, after establishing the suspiciousness of neurons in the hidden layers of a DNN, the neurons are ordered in decreasing order of suspiciousness and the $k, 1 \leq l \leq s$, most probably defective (i.e., "undertrained") neurons are selected. Algorithm 1 presents the high-level steps for identifying and selecting the k most suspicious neurons. When multiple neurons achieve the same suspiciousness score, DeepFault resolves ties by prioritising neurons that belong to deeper hidden layers (i.e., they are closer to the output layer). The rationale for this decision lies in fact that neurons in deeper layers are able to learn more meaningful representations of the input space [69].

3.3 Suspiciousness-Guided Input Synthesis

DeepFault uses the selected k most suspicious neurons (cf. Section 3.2) to synthesize inputs that exercise these neurons and could be adversarial (see Section 5). The premise underlying the synthesis is that increasing the activation values of suspicious neurons will cause the propagation of degenerate information, computed by these neurons, across the network, thus, shifting the decision boundaries in the output layer. To achieve this, DeepFault applies targeted modification of test inputs from the test set $\mathcal{T}$ for which the DNN made correct decisions (e.g., for a classification task, the DNN determined correctly their ground truth classes) aiming to steer the DNN decision to a different region (see Fig. 2).

Algorithm 2 shows the high-level process for synthesising new inputs based on the identified suspicious neurons. The synthesis task is underpinned by a gradient ascent algorithm that aims at determining the extent to which a correctly classified input should be modified to increase the activation values of suspicious neurons. For any test input $t \in T_s$ correctly classified by the DNN, we extract the value of each suspicious neuron and its gradient in lines 6 and 7, respectively. Then, by iterating over each input dimension $u \in t$, we determine the gradient value $u_{gradient}$ by which u will be perturbed (lines 11-12). The value of $u_{gradient}$ is based on the mean gradient of u across the suspicious neurons controlled by the function GRADIENTCONSTRAINTS. This function uses a test set specific *step* parameter and a distance d parameter to facilitate the synthesis of realistic test inputs that are sufficiently *close*, according to L_∞-norm, to the original inputs. We demonstrate later in the evaluation of DeepFault (cf. Table 4) that these parameters enable the synthesis of inputs similar to the original. The function DOMAINCONSTRAINTS applies domain-specific constraints thus ensuring that u changes due to gradient ascent result in realistic and physically reproducible test inputs as in [48]. For instance, a domain-specific constraint for an image classification dataset involves bounding the pixel values of synthesized images to be within a certain range (e.g., 0–1 for the MNIST dataset [32]). Finally, we append the updated u to construct a new test input t' (line 13).

As we experimentally show in Section 5, the suspiciousness measures used by DeepFault can synthesize adversarial inputs that cause the DNN to misclassify previously correctly classified inputs. Thus, the identified suspicious neurons can be attributed a degree of responsibility for the inadequate network performance meaning that their weights have not been optimised. This reduces the DNN's ability for high generalisability and correct operation in untrained data.

4 Implementation

To ease the evaluation and adoption of the DeepFault approach (cf. Fig. 2), we have implemented a prototype tool on top of the open-source machine learning framework Keras (v2.2.2) [9] with Tensorflow(v1.10.1) backend [2]. The full experimental results are summarised in the following section.

5 Evaluation

5.1 Experimental Setup

We evaluate DeepFault on two popular publicly available datasets. MNIST [32] is a handwritten digit dataset with 60,000 training samples and 10,000 testing samples; each input is a 28x28 pixel image with a class label from 0 to 9. CIFAR-10 [1]) is an image dataset with 50,000 training samples and 10,000 testing samples; each input is a 32x32 image in ten different classes (e.g., dog, bird, car).

For each dataset, we study three DNNs that have been used in previous research [1,60] (Table 2). All DNNs have different architecture and number of trainable parameters. For MNIST, we use fully connected neural networks (dense) and for CIFAR-10 we use convolutional neural networks with max-pooling and dropout layers that have been trained to achieve at least 95% and 70% accuracy on the provided test sets, respectively. The column 'Architecture' shows the number of fully connected hidden layers and the number of neurons per layer. Each DNN uses a leaky ReLU [38] as its activation function ($\alpha = 0.01$), which has been shown to achieve competitive accuracy results [67].

We instantiate DeepFault using the suspiciousness measures Tarantula [23], Ochiai [42] and D* [62] (Table 1). We analyse the effectiveness of DeepFault instances using different number of suspicious neurons, i.e., $k \in \{1, 2, 3, 5, 10\}$

Table 2. Details of MNIST and CIFAR-10 DNNs used in the evaluation.

Dataset	Model Name	# Trainable Params	Architecture	Accuracy
MNIST	MNIST_1	27,420	<5 × 30>	96.6%
	MNIST_2	22,975	<6 × 25>	95.8%
	MNIST_3	18,680	<8 × 20>	95%
CIFAR-10	CIFAR_1	411,434	<4 × 128>	70.1%
	CIFAR_2	724,010	<2 × 256>	72.6%
	CIFAR_3	1,250,858	<1 × 512>	76.1%

and $k \in \{10, 20, 30, 40, 50\}$ for MNIST and CIFAR models, respectively. We also ran preliminary experiments for each model from Table 2 to tune the hyper-parameters of Algorithm 2 and facilitate replication of our findings. Since gradient values are model and input specific, the perturbation magnitude should reflect these values and reinforce their impact. We determined empirically that $step = 1$ and $step = 10$ are good values, for MNIST and CIFAR models, respectively, that enable our algorithm to perturb inputs. We also set the maximum allowed distance d to be at most 10% (L_∞) with regards to the range of each input dimension (maximum pixel value). As shown in Table 4, the synthesized inputs are very similar to the original inputs and are rarely constrained by d. Studying other *step* and d values is part of our future work. All experiments were run on an Ubuntu server with 16 GB memory and Intel Xeon E5-2698 2.20GHz.

5.2 Research Questions

Our experimental evaluation aims to answer the following research questions.

RQ1 (Validation): Can DeepFault find suspicious neurons effectively? If suspicious neurons do exist, suspiciousness measures used by DeepFault should comfortably outperform a random suspiciousness selection strategy.

RQ2 (Comparison): How do DeepFault instances using different suspiciousness measures compare against each other? Since DeepFault can work with multiple suspiciousness measures, we examined the results produced by DeepFault instances using Tarantula [23], Ochiai [42] and D* [62].

RQ3 (Suspiciousness Distribution): How are suspicious neurons found by DeepFault distributed across a DNN? With this research question, we analyse the distribution of suspicious neurons in hidden DNN layers using different suspiciousness measures.

RQ4 (Similarity): How realistic are inputs synthesized by DeepFault? We analysed the distance between synthesized and original inputs to examine the extent to which DeepFault synthesizes realistic inputs.

RQ5 (Increased Activations): Do synthesized inputs increase activation values of suspicious neurons? We assess whether the suspiciousness-guided input synthesis algorithm produces inputs that reinforce the influence of suspicious neurons across a DNN.

RQ6 (Performance): How efficiently can DeepFault synthesize new inputs? We analysed the time consumed by DeepFault to synthesize new inputs and the effect of suspiciousness measures used in DeepFault instances.

5.3 Results and Discussion

RQ1 (Validation). We apply the DeepFault workflow to the DNNs from Table 2. To this end, we instantiate DeepFault with a suspiciousness measure, *analyse* a pre-trained DNN given the dataset's test set $\mathcal{T}$, *identify* k neurons with the highest suspiciousness scores and *synthesize* new inputs, from *correctly classified* inputs, that exercise these suspicious neurons. Then, we measure the prediction performance of the DNN on the synthesized inputs using the standard performance metrics: cross-entropy *loss*, i.e., the divergence between output and target distribution, and *accuracy*, i.e., the percentage of correctly classified inputs over all given inputs. Note that DNN analysis is done per class, since the activation pattern of inputs from the same class is similar to each other [69].

Table 3 shows the average loss and accuracy for inputs synthesized by DeepFault instances using Tarantula (T), Ochiai (O), DStar (D) and a random selection strategy (R) for different number of suspicious neurons k on the MNIST (top) and CIFAR-10 (bottom) models from Table 2. Each cell value in Table 3, except from random R, is averaged over 100 synthesized inputs (10 per class). For R, we collected 500 synthesized inputs (50 per class) over five independent runs, thus, reducing the risk that our findings may have been obtained by chance.

As expected (see Table 3), DeepFault using any suspiciousness measure (T, O, D) obtained considerably lower prediction performance than R on MNIST models. The suspiciousness measures T and O are also effective on CIFAR-10 model, whereas the performance between D and R is similar. These results show that the identified k neurons are actually *suspicious* and, hence, their weights are insufficiently trained. Also, we have sufficient evidence that increasing the activation value of suspicious neurons by slightly perturbing inputs that have been classified correctly by the DNN could transform them into adversarial.

We applied the non-parametric statistical test Mann-Whitney with 95% confidence level [61] to check for statistically significant performance difference between the various DeepFault instances and random. We confirmed the significant difference among T-R and O-R (p-value< 0.05) for all MNIST and CIFAR-10 models and for all k values. We also confirmed the interesting observation that significant difference between D-R exists only for MNIST models (all k values). We plan to investigate this observation further in our future work.

Another interesting observation from Table 3 is the small performance difference of DeepFault instances for different k values. We investigated this further by analyzing the activation values of the next k' most suspicious neurons according to the suspiciousness order given by Algorithm 1. For instance, if $k = 2$ we analysed the activation values of the next $k' \in \{3, , 5, 10\}$ most suspicious neurons. We observed that the synthesized inputs frequently increase the activation values of the k' neurons whose suspiciousness scores are also high, in addition to increasing the values of the top k suspicious neurons.

Considering these results, we have empirical evidence about the existence of *suspicious* neurons which can be responsible for inadequate DNN performance. Also, we confirmed that DeepFault instances using sophisticated suspiciousness measures significantly outperform a random strategy for most of the studied DNN models (except from the D-R case on CIFAR models; see RQ3).

Table 3. Accuracy and loss of inputs synthesized by DeepFault on MNIST (top) and CIFAR-10 (bottom) datasets. The best results per suspiciousness measure are shown in bold. (k:#suspicious neurons, T:Tarantula, O:Ochiai, D:D*, R:Random)

k	**Measure**	**MNIST_1**				**MNIST_2**				**MNIST_3**			
		T	O	D	R	T	O	D	R	T	O	D	R
1	Loss	3.55	**6.19**	4.03	2.42	3.48	3.53	**3.97**	2.78	7.35	**8.23**	6.36	3.66
	Accuracy	0.26	**0.16**	0.2	0.59	0.3	**0.2**	0.5	0.49	0.16	**0.1**	0.13	0.39
2	Loss	3.73	**6.08**	3.18	2.67	3.12	3.76	**4.08**	0.9	4.27	**6.81**	6.5	3.06
	Accuracy	**0.16**	0.23	0.4	0.58	0.23	0.23	**0.13**	0.77	0.29	**0.13**	0.26	0.56
3	Loss	4.1	6.19	**6.25**	1.14	2.39	**3.94**	3.04	1.61	3.33	**7.59**	6.98	2.91
	Accuracy	**0.23**	**0.23**	0.33	0.77	0.46	0.26	**0.23**	0.67	0.26	**0.06**	0.16	0.61
5	Loss	4.63	6.68	**6.97**	1.1	2.49	**3.64**	3.48	0.94	4.15	**7.22**	6.47	1.22
	Accuracy	0.23	0.23	**0.13**	0.79	0.26	0.26	**0.2**	0.73	0.16	**0.1**	0.26	0.77
10	Loss	4.97	6.95	**7.4**	1.3	2.08	3.06	**3.82**	0.49	4.45	**7.16**	5.9	0.57
	Accuracy	0.23	**0.2**	0.23	0.75	0.4	**0.23**	0.26	0.86	**0.13**	**0.13**	**0.13**	0.87

k	**Measure**	**CIFAR_1**				**CIFAR_2**				**CIFAR_3**			
		T	O	D	R	T	O	D	R	T	O	D	R
10	Loss	12.75	**13.49**	1.33	3.25	**8.42**	8.41	0	2.49	**6.12**	1.77	1.12	1.21
	Accuracy	0.2	**0.16**	0.9	0.79	**0.47**	**0.47**	1.0	0.84	**0.62**	0.88	0.92	0.91
20	Loss	**12.79**	12.43	0.45	1.8	**8.81**	6.92	0.32	1.67	**6.12**	1.12	0.96	0.64
	Accuracy	**0.2**	0.22	0.96	0.88	**0.44**	0.55	0.97	0.89	**0.62**	0.92	0.93	0.95
30	Loss	**13.19**	13.13	0.38	1.43	**8.35**	6.32	0.55	0.86	**5.64**	0.76	0.42	0.41
	Accuracy	**0.18**	**0.18**	0.95	0.9	**0.48**	0.6	0.95	0.94	**0.64**	0.93	0.96	0.97
40	Loss	**13.69**	11.92	0.8	1.29	**9.4**	5.01	0.32	0.61	**4.51**	1.12	0.22	0.54
	Accuracy	**0.14**	0.26	0.92	0.91	**0.41**	0.68	0.97	0.95	**0.72**	0.92	0.97	0.96
50	Loss	12.1	**13.37**	0.36	0.9	**9.59**	3.38	0	0.56	**4.67**	0.04	0.64	0.48
	Accuracy	0.24	**0.17**	0.96	0.94	**0.4**	0.78	1.0	0.96	**0.71**	0.98	0.96	0.96

RQ2 (Comparison). We compare DeepFault instances using different suspiciousness measures and carried out pairwise comparisons using the Mann-Whitney test to check for significant difference between T, O, and D*. We show the results of these comparisons in Appendix. We observe that Ochiai achieves better results on MNIST_1 and MNIST_3 models for various k values. This result suggests that the suspicious neurons reported by Ochiai are more responsible for insufficient DNN performance. D* performs competitively on MNIST_1 and MNIST_3 for $k \in \{3, 5, 10\}$, but its performance on CIFAR-10 models is significantly inferior to Tarantula and Ochiai. The best performing suspiciousness measure in CIFAR models for most k values is, by a great amount, Tarantula.

These findings show that multiple suspiciousness measures could be used for instantiating DeepFault with competitive performance. We also have evidence that DeepFault using D* is ineffective for some complex networks (e.g., CIFAR-10), but there is insufficient evidence for the best performing DeepFault instance. Our findings conform to the latest research on software fault localization which claims that there is no single best spectrum-based suspiciousness measure [47].

RQ3 (Suspiciousness Distribution). We analysed the distribution of suspicious neurons identified by DeepFault instances across the hidden DNN layers. Fig. 3 shows the distribution of suspicious neurons on MNIST_3 and CIFAR_3 models with $k = 10$ and $k = 50$, respectively. Considering MNIST_3, the majority of suspicious neurons are located at the deeper hidden layers (Dense 4-Dense 8)

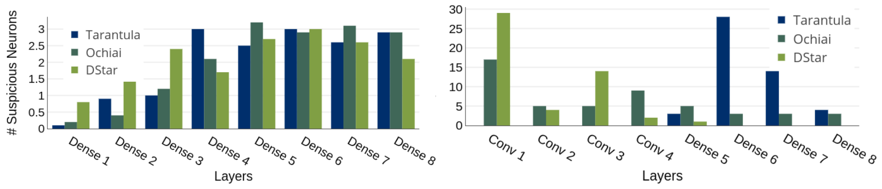

Fig. 3. Suspicious neurons distribution on MNIST_3 (left) and CIFAR_3 (right) models.

irrespective of the suspiciousness measure used by DeepFault. This observation holds for the other MNIST models and k values. On CIFAR_3, however, we can clearly see variation in the distributions across the suspiciousness measures. In fact, D* suggests that most of the suspicious neurons belong to initial hidden layers which is in contrast with Tarantula's recommendations. As reported in RQ2, the inputs synthesized by DeepFault using Tarantula achieved the best results on CIFAR models, thus showing that the identified neurons are actually suspicious. This difference in the distribution of suspicious neurons explains the inferior inputs synthesized by D* on CIFAR models (Table 3).

Another interesting finding concerns the relation between the suspicious neurons distribution and the "adversarialness" of synthesized inputs. When suspicious neurons belong to deeper hidden layers, the likelihood of the synthesized input being adversarial increases (cf. Table 3 and Fig. 3). This finding is explained by the fact that initial hidden layers transform input features (e.g., pixel values) into abstract features, while deeper hidden layers extract more semantically meaningful features and, thus, have higher influence in the final decision [13].

RQ4 (Similarity). We examined the distance between original, correctly classified, inputs and those synthesized by DeepFault, to establish DeepFault's ability to synthesize realistic inputs. Table 4 (left) shows the distance between original and synthesized inputs for various distance metrics (L_1 Manhattan, L_2 Euclidean, $L\infty$ Chebyshev) for different k values (# suspicious neurons). The distance values, averaged over inputs synthesized using the DeepFault suspiciousness measures (T, O and D*), demonstrate that the degree of perturbation is similar irrespective of k for MNIST models, whereas for CIFAR models the distance decreases as k increases. Given that a MNIST input consists of 784 pixels, with each pixel taking values in $[0, 1]$, the average perturbation per input is less than 5.28% of the total possible perturbation (L_1 distance). Similarly, for a CIFAR input that comprises 3072 pixels, with each pixel taking values in {0,1,...,255}, the average perturbation per input is less that 0.03% of the total possible perturbation (L_1 distance). Thus, for both datasets, the difference of synthesized inputs to their original versions is very small. We qualitatively support our findings by showing in Fig. 4 the synthesized images and their originals for an example set of inputs from the MNIST and CIFAR-10 datasets.

We also compare the distances between original and synthesized inputs based on the suspiciousness measures (Table 4 right). The inputs synthesized by DeepFault instances using T, O or D* are very close to the inputs of the random selection strategy (L_1 distance). Considering these results, we can conclude that

Table 4. Distance between synthesized and original inputs. The values shown represent minimal perturbation to the original inputs ($<5\%$ for MNIST and $<1\%$ for CIFAR-10).

k	MNIST			CIFAR		
MNIST(CIFAR)	L_1	L_2	L_∞	L_1	L_2	L_∞
1(10)	41.4	2.0	0.1	179.07	7216.6	15.46
2(20)	41.2	1.99	0.1	144.95	5897.4	12.45
3(30)	40.9	1.98	0.1	124.61	5073.9	10.67
5(40)	40.7	1.97	0.1	113.45	4579.2	9.89
10(50)	40.3	1.96	0.1	104.72	4273	9.24

Susp.	**MNIST**			**CIFAR**		
measure	L_1	L_2	L_∞	L_1	L_2	L_∞
Tarantula	40.3	1.97	0.1	180.23	6575.6	19.41
Ochiai	41.0	1.98	0.1	110.45	4825.3	7.84
DStar	41.5	1.99	0.1	109.4	4823.2	7.39
Random	39.2	1.92	0.1	121.73	4988.1	11.63

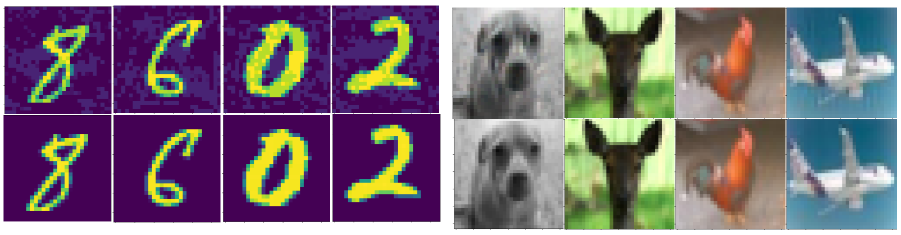

Fig. 4. Synthesized images (top) and their originals (bottom). For each dataset, suspicious neurons are found using (from left to right) Tarantula, Ochiai, Dstar and Random.

DeepFault is effective in synthesizing highly adversarial inputs (cf. Table 3) that closely resemble their original counterparts.

RQ5 (Increasing Activations). We studied the activation values of suspicious neurons identified by DeepFault to examine whether the synthesized inputs increase the values of these neurons. The gradients of suspicious neurons used in our suspiciousness-guided input synthesis algorithm might be conflicting and a global increase in all suspicious neurons' values might not be feasible. This can occur if some neurons' gradients are negative, indicating a decrease in an input feature's value, whereas other gradients are positive and require to increase the value of the same feature. Table 5 shows the percentage of suspicious neurons k, averaged over all suspiciousness measures for all considered MNIST and CIFAR-10 models from Table 2, whose values were increased by the inputs synthesized by DeepFault. For MNIST models, DeepFault synthesized inputs that increase the suspicious neurons' values with success at least 97% for $k \in \{1,2,3,5\}$, while the average effectiveness for CIFAR models is 90%. These results show the effectiveness of our suspiciousness-guided input synthesis algorithm in generating inputs that increase the activation values of suspicious neurons (see Appendix).

Table 5. Effectiveness of *suspiciousness-guided input synthesis* algorithm to increase activations values of suspicious neurons.

	k: MNIST(CIFAR)				
Datasets	**1(10)**	**2(20)**	**3(30)**	**5(40)**	**10(50)**
MNIST	98%	99%	97%	97%	91%
CIFAR	91%	92%	90%	89%	88%

RQ6 (Performance). We measured the performance of Algorithm 2 to synthesize new inputs (Table 10 in Appendix). The average time required to synthesize a single input for MNIST and CIFAR models is $1s$ and $24.3s$, respectively. As expected, the performance of the algorithm depends on the number of suspicious neurons (k), the distribution of those neurons over the DNN and its architecture.

For CIFAR models, for instance, the execution time per input ranges between $3s$ ($k = 10$) and $48s$ ($k = 50$). We also confirmed empirically that more time is taken to synthesize an input if the suspicious neurons are in deeper hidden layers.

5.4 Threats to Validity

Construct validity threats might be due to the adopted experimental methodology including the selected datasets and DNN models. To mitigate this threat, we used widely studied public datasets (MNIST [32] and CIFAR-10 [1]), and applied DeepFault to multiple DNN models of different architectures with competitive prediction accuracies (cf. Table 2). Also, we mitigate threats related to the identification of suspicious neurons (Algorithm 1) by adapting suspiciousness measures from the fault localisation domain in software engineering [63].
Internal validity threats might occur when establishing the ability of DeepFault to synthesize new inputs that exercise the identified suspicious neurons. To mitigate this threat, we used various distance metrics to confirm that the synthesized inputs are close to the original inputs and similar to the inputs synthesized by a random strategy. Another threat could be that the suspiciousness measures employed by DeepFault accidentally outperform the random strategy. To mitigate this threat, we reported the results of the random strategy over five independent runs per experiment. Also, we ensured that the distribution of the randomly selected suspicious neurons resembles the distribution of neurons identified by DeepFault suspiciousness measures. We also used the non-parametric statistical test Mann-Whitney to check for significant difference in the performance of DeepFault instances and random with a 95% confidence level.
External validity threats might exist if DeepFault cannot access the internal DNN structure to assemble the hit spectrums of neurons and establish their suspiciousness. We limit this threat by developing DeepFault using the open-source frameworks Keras and Tensorflow which enable whitebox DNN analysis. We also examined various spectrum-based suspiciousness measures, but other measures can be investigated [63]. We further reduce the risk that DeepFault might be difficult to use in practice by validating it against several DNN instances trained on two widely-used datasets. However, more experiments are needed to assess the applicability of DeepFault in domains and networks with characteristics different from those used in our evaluation (e.g., LSTM and Capsules networks [50]).

6 Related Work

DNN Testing and Verification. The inability of blackbox DNN testing to provide insights about the internal neuron activity and enable identification of corner-case inputs that expose unexpected network behaviour [14], urged researchers to leverage whitebox testing techniques from software engineering [28, 35,43,48,54]. DeepXplore [48] uses a differential algorithm to generate inputs that increase neuron coverage. DeepGauge [35] introduces multi-granularity coverage criteria for effective test synthesis. Other research proposes testing criteria and techniques inspired by metamorphic testing [58], combinatorial testing [37], mutation testing [36], MC/DC [54], symbolic execution [15] and concolic testing [55].

Formal DNN verification aims at providing guarantees for trustworthy DNN operation [20]. Abstraction refinement is used in [49] to verify safety properties of small neural networks with sigmoid activation functions, while AI^2 [12] employs abstract interpretation to verify similar properties. Reluplex [26] is an SMT-based approach that verifies safety and robustness of DNNs with ReLUs, and DeepSafe [16] uses Reluplex to identify safe regions in the input space. DLV [60] can verify local DNN robustness given a set of user-defined manipulations.

DeepFault adopts spectrum-based fault localisation techniques to systematically identify suspicious neurons and uses these neurons to synthesize new inputs, which is mostly orthogonal to existing research on DNN testing and verification.

Adversarial Deep Learning. Recent studies have shown that DNNs are vulnerable to adversarial examples [57] and proposed search algorithms [8,40,41,44], based on gradient descent or optimisation techniques, for generating adversarial inputs that have a minimal difference to their original versions and force the DNN to exhibit erroneous behaviour. These types of adversarial examples have been shown to exist in the physical world too [29]. The identification of and protection against these adversarial attacks, is another active area of research [45,59]. DeepFault is similar to these approaches since it uses the identified suspicious neurons to synthesize perturbed inputs which as we have demonstrated in Section 5 are adversarial. Extending DeepFault to support the synthesis of adversarial inputs using these adversarial search algorithms is part of our future work.

Fault Localization in Traditional Software. Fault localization is widely studied in many software engineering areas including including software debugging [46], program repair [17] and failure reproduction [21, 22]. The research focus in fault localization is the development of identification methods and suspiciousness measures that isolate the root causes of software failures with reduced engineering effort [47]. The most notable fault localization methods are spectrum-based [3,23,30,31,62], slice-based [64] and model-based [39]. Threats to the value of empirical evaluations of spectrum-based fault localisation are studied in [53], while the theoretical analyses in [66,68] set a formal foundation about desirable formal properties that suspiciousness measures should have. We refer interested readers to a recent comprehensive survey on fault localization [63].

7 Conclusion

The potential deployment of DNNs in safety-critical applications introduces unacceptable risks. To reduce these risks to acceptable levels, DNNs should be tested thoroughly. We contribute in this effort, by introducing DeepFault, the first fault localization-based whitebox testing approach for DNNs. DeepFault *analyzes* pre-trained DNNs, given a specific test set, to establish the hit spectrum of each neuron, *identifies suspicious neurons* by employing suspiciousness measures and *synthesizes* new inputs that increase the activation values of the suspicious neurons. Our empirical evaluation on the widely-used MNIST and CIFAR-10 datasets shows that DeepFault can identify neurons which can be held responsible for inadequate performance. DeepFault can also synthesize new inputs, which closely resemble the original inputs, are highly adversarial and exercise the identified suspicious neurons. In future work, we plan to evaluate DeepFault on other DNNs and datasets, to improve the suspiciousness-guided synthesis algorithm and to extend the synthesis of adversarial inputs [44]. We will also explore techniques to repair the identified suspicious neurons, thus enabling to reason about the safety of DNNs and support safety case generation [7, 27].

Appendix

Fault Localization in Software Engineering

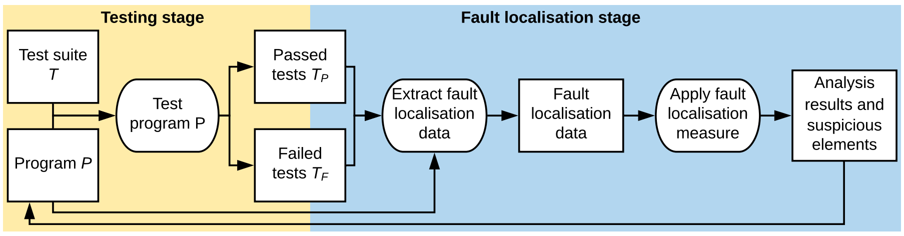

Fig. 5. General fault localization for traditional software systems.

Additional Material for RQ5

Table 6. Ratios of increases in activations of k' suspicious neurons with inputs that are synthesized to increase activations of k neurons where $k' \geq k$ in MNIST networks. Synthesized inputs, most of the time, increase the activation values of the top k' neurons in addition to increasing the values of the top k suspicious neurons.

	Tarantula					**Ochiai**					**D***				
k \ k'	**1**	**2**	**3**	**5**	**10**	**1**	**2**	**3**	**5**	**10**	**1**	**2**	**3**	**5**	**10**
1	1.0	0.98	0.94	0.90	0.78	1.0	1.0	0.96	0.97	0.90	1.0	0.95	0.96	0.97	0.84
2	■	1.0	0.98	0.91	0.81	■	1.0	0.99	0.99	0.89	■	1.0	0.98	0.98	0.87
3	■	■	0.98	0.97	0.87	■	■	1.0	0.99	0.90	■	■	1.0	1.0	0.91
5	■	■	■	0.98	0.89	■	■	■	0.98	0.91	■	■	■	1.0	0.90
10	■	■	■	■	0.91	■	■	■	■	0.97	■	■	■	■	0.94

Table 7. Ratios of increases in activations of k' suspicious neurons with inputs that are synthesized to increase activations of k neurons where $k' \geq k$ in CIFAR networks. Synthesized inputs, most of the time (except from D* case), increase the activation values of the top k' neurons in addition to increasing the values of the top k suspicious neurons.

	Tarantula					**Ochiai**					**D***				
k \ k'	**10**	**20**	**30**	**40**	**50**	**10**	**20**	**30**	**40**	**50**	**10**	**20**	**30**	**40**	**50**
10	0.95	0.91	0.92	0.90	0.89	0.95	0.92	0.92	0.91	0.89	0.81	0.61	0.56	0.32	0.15
20	■	0.96	0.94	0.92	0.90	■	0.97	0.95	0.94	0.91	■	0.78	0.49	0.31	0.15
30	■	■	0.94	0.92	0.91	■	■	0.92	0.91	0.90	■	■	0.79	0.33	0.15
40	■	■	■	0.95	0.93	■	■	■	0.97	0.93	■	■	■	0.65	0.17
50	■	■	■	■	0.94	■	■	■	■	0.91	■	■	■	■	0.7

Additional Material for RQ2

Table 8. Statistical significance evaluation of techniques when compared to each other on MNIST networks. A checkmark is put when a technique on the row has a significantly better loss/accuracy value than the technique on the column for given network and k parameter (k stands for number of suspicious neurons).

			Tarantula		**Ochiai**		**D***	
Technique	**Network**	k	**Loss**	**Accuracy**	**Loss**	**Accuracy**	**Loss**	**Accuracy**
Tarantula	**MNIST_1**	**1**	■	■				
		2	■	■		✓		
		3	■	■				
		5	■	■				
		10	■	■				
	MNIST_2	**1**	■	■				
		2	■	■				
		3	■	■				
		5	■	■				
		10	■	■				
	MNIST_3	**1**	■	■				
		2	■	■				
		3	■	■				
		5	■	■				
		10	■	■				
Ochiai	**MNIST_1**	**1**	✓	✓	■	■	✓	
		2	✓		■	■	✓	
		3	✓		■	■		
		5	✓		■	■		
		10	✓		■	■		
	MNIST_2	**1**			■	■		✓
		2			■	■		
		3	✓	✓	■	■		
		5	✓		■	■		
		10		✓	■	■		
	MNIST_3	**1**			■	■	✓	
		2			■	■		
		3			■	■		
		5			■	■		✓
		10			■	■	✓	
D*	**MNIST_1**	**1**					■	■
		2					■	■
		3	✓				■	■
		5	✓				■	■
		10	✓				■	■
	MNIST_2	**1**					■	■
		2					■	■
		3		✓			■	■
		5					■	■
		10	✓	✓			■	■
	MNIST_3	**1**					■	■
		2	✓				■	■
		3	✓				■	■
		5	✓				■	■
		10	✓				■	■

Table 9. Statistical significance evaluation of techniques when compared to each other on CIFAR networks. A checkmark is put when a technique on the row has a significantly better loss/accuracy value than the technique on the column for given network and k parameter (k stands for number of suspicious neurons).

			Tarantula		**Ochiai**		**D***	
Technique	**Network**	k	**Loss**	**Accuracy**	**Loss**	**Accuracy**	**Loss**	**Accuracy**
Tarantula	**CIFAR_1**	**10**	■	■			✓	✓
		20	■	■			✓	✓
		30	■	■			✓	✓
		40	■	■		✓	✓	✓
		50	■	■			✓	✓
	CIFAR_2	**10**	■	■			✓	✓
		20	■	■	✓		✓	✓
		30	■	■	✓		✓	✓
		40	■	■	✓	✓	✓	✓
		50	■	■	✓	✓	✓	✓
	CIFAR_3	**10**	■	■	✓	✓	✓	✓
		20	■	■	✓	✓	✓	✓
		30	■	■	✓	✓	✓	✓
		40	■	■	✓	✓	✓	✓
		50	■	■	✓	✓	✓	✓
Ochiai	**CIFAR_1**	**10**			■	■	✓	✓
		20			■	■	✓	✓
		30			■	■	✓	✓
		40			■	■	✓	✓
		50			■	■	✓	✓
	CIFAR_2	**10**			■	■	✓	✓
		20			■	■	✓	✓
		30			■	■	✓	✓
		40			■	■	✓	✓
		50			■	■	✓	✓
	CIFAR_3	**10**			■	■		
		20			■	■		
		30			■	■		
		40			■	■		
		50			■	■		
D*	**CIFAR_1**	**10**					■	■
		20					■	■
		30					■	■
		40					■	■
		50					■	■
	CIFAR_2	**10**					■	■
		20					■	■
		30					■	■
		40					■	■
		50					■	■
	CIFAR_3	**10**					■	■
		20					■	■
		30					■	■
		40					■	■
		50					■	■

Additional Material for RQ6

Table 10. Performance measurement of input perturbation in DeepFault. (mm:ss)

DNNs	**Susp. Measure**	k**=1(10)**	k**=2(20)**	k**=3(30)**	k**=5(40)**	k**=10(50)**
MNIST	T	< 00:01	< 00:01	< 00:01	00:01	00:03
	O	< 00:01	< 00:01	< 00:01	00:01	00:03
	D	< 00:01	< 00:01	< 00:01	< 00:01	00:02
CIFAR	T	00:04	00:14	00:29	00:52	01:20
	O	00:03	00:09	00:17	00:27	00:41
	D	00:02	00:04	00:08	00:14	00:23

References

1. Cifar10 model in keras. `https://github.com/keras-team/keras/blob/master/examples/cifar10_cnn.py`, accessed: 08-10-2018
2. Abadi, M., Barham, P., Chen, J., et al.: Tensorflow: A system for large-scale machine learning. In: 12th USENIX Symposium on Operating Systems Design and Implementation. pp. 265–283 (2016)
3. Abreu, R., Zoeteweij, P., Golsteijn, R., Van Gemund, A.J.: A practical evaluation of spectrum-based fault localization. Journal of Systems and Software **82**(11), 1780–1792 (2009)
4. Artzi, S., Dolby, J., Tip, F., Pistoia, M.: Directed test generation for effective fault localization. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 49–60 (2010)
5. Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., et al.: End to end learning for self-driving cars (2016)
6. Burton, S., Gauerhof, L., Heinzemann, C.: Making the case for safety of machine learning in highly automated driving. In: Computer Safety, Reliability, and Security. pp. 5–16 (2017)
7. Calinescu, R., Weyns, D., Gerasimou, S., Iftikhar, M.U., Habli, I., Kelly, T.: Engineering trustworthy self-adaptive software with dynamic assurance cases. IEEE Transactions on Software Engineering **44**(11), 1039–1069 (2018)
8. Carlini, N., Wagner, D.: Towards evaluating the robustness of neural networks. In: IEEE Symposium on Security and Privacy (S&P). pp. 39–57 (2017)
9. Chollet, F., et al.: Keras. `https://keras.io` (2015)
10. Cireşan, D., Meier, U., Schmidhuber, J.: Multi-column deep neural networks for image classification. In: Conference on Computer Vision and Pattern Recognition (CVPR). pp. 3642–3649 (2012)
11. Cui, Z., Xue, F., Cai, X., Cao, Y., et al.: Detection of malicious code variants based on deep learning. IEEE Transactions on Industrial Informatics **14**(7), 3187–3196 (2018)
12. Gehr, T., Mirman, M., Drachsler-Cohen, D., Tsankov, P., et al.: AI2: Safety and robustness certification of neural networks with abstract interpretation. In: IEEE Symposium on Security and Privacy (S&P). pp. 1–16 (2018)
13. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016), `http://www.deeplearningbook.org`
14. Goodfellow, I., Papernot, N.: The challenge of verification and testing of machine learning (2017)

15. Gopinath, D., Wang, K., Zhang, M., Pasareanu, C.S., Khurshid, S.: Symbolic execution for deep neural networks. In: arXiv preprint arXiv:1807.10439 (2018)
16. Gopinath, D., Katz, G., Pasareanu, C.S., Barrett, C.: DeepSafe: A data-driven approach for checking adversarial robustness in neural networks. arXiv preprint arXiv:1710.00486 (2017)
17. Goues, C.L., Nguyen, T., Forrest, S., Weimer, W.: GenProg: A generic method for automatic software repair. IEEE Transactions on Software Engineering **38**(1), 54–72 (2012)
18. Guo, J., Jiang, Y., Zhao, Y., Chen, Q., Sun, J.: DLFuzz: Differential fuzzing testing of deep learning systems. In: ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE). pp. 739–743 (2018)
19. Hinton, G., Deng, L., Yu, D., Dahl, G.E., et al.: Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Processing Magazine **29**(6), 82–97 (2012)
20. Huang, X., Kwiatkowska, M., Wang, S., Wu, M.: Safety verification of deep neural networks. In: International Conference on Computer Aided Verification (CAV). pp. 3–29 (2017)
21. Jin, W., Orso, A.: BugRedux: Reproducing field failures for in-house debugging. In: International Conference on Software Engineering (ICSE). pp. 474–484 (2012)
22. Jin, W., Orso, A.: F3: Fault localization for field failures. In: ACM International Symposium on Software Testing and Analysis (ISSTA). pp. 213–223 (2013)
23. Jones, J.A., Harrold, M.J.: Empirical evaluation of the tarantula automatic fault-localization technique. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 273–282 (2005)
24. Jorgensen, P.C.: Software testing: a craftsman's approach. Auerbach Publications (2013)
25. Julian, K.D., Lopez, J., Brush, J.S., Owen, M.P., Kochenderfer, M.J.: Policy compression for aircraft collision avoidance systems. In: IEEE Digital Avionics Systems Conference (DASC). pp. 1–10 (2016)
26. Katz, G., Barrett, C., Dill, D.L., Julian, K., Kochenderfer, M.J.: Reluplex: An efficient SMT solver for verifying deep neural networks. In: International Conference on Computer Aided Verification (CAV). pp. 97–117 (2017)
27. Kelly, T.P.: Arguing safety: a systematic approach to managing safety cases. Ph.D. thesis, University of York York (1999)
28. Kim, J., Feldt, R., Yoo, S.: Guiding deep learning system testing using surprise adequacy. In: arXiv preprint arXiv:1808.08444 (2018)
29. Kurakin, A., Goodfellow, I., Bengio, S.: Adversarial examples in the physical world. arXiv preprint arXiv:1607.02533 (2016)
30. Landsberg, D., Chockler, H., Kroening, D., Lewis, M.: Evaluation of measures for statistical fault localisation and an optimising scheme. In: International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 115–129 (2015)
31. Landsberg, D., Sun, Y., Kroening, D.: Optimising spectrum based fault localisation for single fault programs using specifications. In: International Conference on Fundamental Approaches to Software Engineering (FASE). pp. 246–263 (2018)
32. LeCun, Y.: The MNIST database of handwritten digits. http://yann. lecun. com/exdb/mnist (1998)
33. Lecun, Y., Bengio, Y., Hinton, G.: Deep learning. Nature **521**(7553), 436–444 (2015)
34. Litjens, G., Kooi, T., Bejnordi, B.E., et al.: A survey on deep learning in medical image analysis. Medical Image Analysis **42**, 60–88 (2017)

35. Ma, L., Juefei-Xu, F., Zhang, F., Sun, J., et al.: DeepGauge: Multi-granularity testing criteria for deep learning systems. In: IEEE/ACM International Conference on Automated Software Engineering (ASE) (2018)
36. Ma, L., Zhang, F., Sun, J., Xue, M., et al.: DeepMutation: Mutation testing of deep learning systems. In: IEEE International Symposium on Software Reliability Engineering (ISSRE) (2018)
37. Ma, L., Zhang, F., Xue, M., Li, B., et al.: Combinatorial testing for deep learning systems. In: arXiv preprint arXiv:1806.07723 (2018)
38. Maas, A.L., Hannun, A.Y., Ng, A.Y.: Rectifier nonlinearities improve neural network acoustic models. In: International Conference on Machine Learning (ICML). vol. 30, p. 3 (2013)
39. Mayer, W., Stumptner, M.: Evaluating models for model-based debugging. In: IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 128–137 (2008)
40. Moosavi-Dezfooli, S.M., Fawzi, A., Frossard, P.: Deepfool: A simple and accurate method to fool deep neural networks. In: IEEE Conference on Computer Vision and Pattern Recognition (CVPR). pp. 2574–2582 (2016)
41. Nguyen, A., Yosinski, J., Clune, J.: Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In: Conference on Computer Vision and Pattern Recognition (CVPR). pp. 427–436 (2015)
42. Ochiai, A.: Zoogeographic studies on the soleoid fishes found in japan and its neighbouring regions. Bulletin of Japanese Society of Scientific Fisheries **22**, 526–530 (1957)
43. Odena, A., Goodfellow, I.: Tensorfuzz: Debugging neural networks with coverage-guided fuzzing. In: arXiv preprint arXiv:1807.10875 (2018)
44. Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., et al.: The limitations of deep learning in adversarial settings. In: International Symposium on Security and Privacy (S&P). pp. 372–387 (2016)
45. Papernot, N., McDaniel, P., Wu, X., Jha, S., Swami, A.: Distillation as a defense to adversarial perturbations against deep neural networks. In: International Symposium on Security and Privacy (S&P). pp. 582–597 (2016)
46. Parnin, C., Orso, A.: Are automated debugging techniques actually helping programmers? In: International Symposium on Software Testing and Analysis (ISSTA). pp. 199–209 (2011)
47. Pearson, S., Campos, J., Just, R., Fraser, G., et al.: Evaluating and improving fault localization. In: International Conference on Software Engineering (ICSE). pp. 609–620 (2017)
48. Pei, K., Cao, Y., Yang, J., Jana, S.: DeepXplore: Automated whitebox testing of deep learning systems. In: Symposium on Operating Systems Principles (SOSP). pp. 1–18 (2017)
49. Pulina, L., Tacchella, A.: An abstraction-refinement approach to verification of artificial neural networks. In: International Conference on Computer Aided Verification (CAV). pp. 243–257 (2010)
50. Sabour, S., Frosst, N., Hinton, G.E.: Dynamic routing between capsules. In: Advances in Neural Information Processing Systems. pp. 3856–3866 (2017)
51. Salay, R., Queiroz, R., Czarnecki, K.: An analysis of ISO26262: Using machine learning safely in automotive software. arXiv preprint arXiv:1709.02435 (2017)
52. Seshia, S.A., Desai, A., Dreossi, T., Fremont, D.J., Ghosh, S., Kim, E., Shivakumar, S., Vazquez-Chanlatte, M., Yue, X.: Formal specification for deep neural networks. In: Technical Report, University of California at Berkeley (2018)

53. Steimann, F., Frenkel, M., Abreu, R.: Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In: International Symposium on Software Testing and Analysis (ISSTA). pp. 314–324 (2013)
54. Sun, Y., Huang, X., Kroening, D.: Testing deep neural networks. In: arXiv preprint arXiv:1803.04792 (2018)
55. Sun, Y., Wu, M., Ruan, W., Huang, X., et al.: Concolic testing for deep neural networks. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE). pp. 109–119 (2018)
56. Sutskever, I., Vinyals, O., Le, Q.V.: Sequence to sequence learning with neural networks. In: International Conference on Neural Information Processing Systems. pp. 3104–3112 (2014)
57. Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., et al.: Intriguing properties of neural networks. arXiv preprint arXiv:1312.6199 (2013)
58. Tian, Y., Pei, K., Jana, S., Ray, B.: Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In: International Conference on Software Engineering (ICSE). pp. 303–314 (2018)
59. Tramèr, F., Kurakin, A., Papernot, N., Goodfellow, I., et al.: Ensemble adversarial training: Attacks and defenses. arXiv preprint arXiv:1705.07204 (2017)
60. Wicker, M., Huang, X., Kwiatkowska, M.: 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). pp. 408–426 (2018)
61. Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., et al.: Experimentation in Software Engineering. Springer Science & Business Media (2012)
62. Wong, W.E., Debroy, V., Gao, R., Li, Y.: The DStar method for effective software fault localization. IEEE Transactions on Reliability **63**(1), 290–308 (2014)
63. Wong, W.E., Gao, R., Li, Y., Abreu, R., Wotawa, F.: A survey on software fault localization. IEEE Transactions on Software Engineering **42**(8), 707–740 (2016)
64. Wong, W.E., Qi, Y.: Effective program debugging based on execution slices and inter-block data dependency. Journal of Systems and Software **79**(7), 891–903 (2006)
65. Wu, M., Wicker, M., Ruan, W., Huang, X., Kwiatkowska, M.: A game-based approximate verification of deep neural networks with provable guarantees. In: arXiv preprint arXiv:1807.03571 (2018)
66. Xie, X., Chen, T.Y., Kuo, F.C., Xu, B.: A theoretical analysis of the risk evaluation formulas for spectrum-based fault localization. ACM Transactions on Software Engineering Methodology **22**(4), 31–40 (2013)
67. Xu, B., Wang, N., Chen, T., Li, M.: Empirical evaluation of rectified activations in convolutional network. arXiv preprint arXiv:1505.00853 (2015)
68. Yoo, S., Xie, X., Kuo, F.C., Chen, T.Y., Harman, M.: Human competitiveness of genetic programming in spectrum-based fault localisation: Theoretical and empirical analysis. ACM Transactions on Software Engineering Methodology **26**(1), 4–30 (2017)
69. Zeiler, M.D., Fergus, R.: Visualizing and understanding convolutional networks. In: European Conference on Computer Vision (ECCV). pp. 818–833 (2014)